2

Towards a Roadmap for Effective Handset Network Test Automation

Clauirton A. Siebra, Andre L. M. Santos & Fabio Q. B. Silva
CIN/Samsung Laboratory of Research and Development[1] - UFPE
Brazil

1. Introduction

In recent years, wireless networks have presented a significant evolution in their technology. While first-generation networks, based on analog signalling, were targeted primarily at voice and data communications occurring at low data rates, we have recently seen the evolution of second- and third-generation wireless systems that incorporate the features provided by broadband networks (Garg, 2001). In addition to supporting mobility, broadband networks also aims to support multimedia traffic, with quality of service (QoS) assurance. Therefore the evolution from 2G to 3G wireless technology bears the promise of offering a wide range of new multimedia services to mobile subscribers (De Vriendt et al., 2002).
In this context, handset devices are following this ongoing network evolution and taking advantage of this technological update to offer a broad variety of resources and applications to their users. In fact, handset development has evolved into a complex engineering process, mainly because of the recent network capabilities supported by new mobile communication and computer technology advances (memory speed and size, processing power, better resources for information delivery, etc.). This scenario has increased the demand on the test phase of handset development, which is required to apply more extensive and efficient evaluation procedures so that the final product meets the fast time-to-market goals and can compete in the global marketplace.
While the number and complexity of tests are increasing, test centers need to decrease their test execution time. The quicker a specific handset is evaluated and delivered to the market, the better will be its chances to compete with other models. Therefore we have a contradiction: we need to increase the number of tests and decrease the test time. Furthermore, this contradiction can lead us to reduce the quality of our test processes.
Test automation is one alternative to this emerging scenario, because it enables tests to be launched and executed without the need for user intervention. Thus, common delays and errors associated with the manipulation of test parameters by humans can be avoided.

[1] The results presented in this chapter have been developed as part of a collaborative project between Samsung Institute for Development of Informatics (Samsung/SIDI) and the Centre of Informatics at the Federal University of Pernambuco (CIn/UFPE), financed by Samsung Eletronica da Amazonia Ltda., under the auspices of the Brazilian Federal Law of Informatics no. 8248/91.

Furthermore, tests can be continuously run during hours or days, ensuring maximum test coverage.

Considering such issues, this chapter discusses the steps associated with the specification of an automation architecture for network tests of handsets. We begin in Section 2 by presenting the handset test domain and an initial test architecture to support simulation of real network conditions in laboratory. We then look at techniques for dealing with the automation requirements for this initial architecture. Section 3 details the *semantic definition* and standardization of test cases, so that their content can be used in a common way by decision-making processes. Section 4 discusses the use of Intelligent Planning as an option to create appropriate sequences of test cases, based on specific devices, environment features and semantics. Section 5 introduces the concept of *autonomic computing* and shows how *knowledge-based systems* can be used to create a self-managing test process that can deal with unexpected situations. During the test performance, it is important to monitor the test indicators to ensure the process quality and to identify failures and opportunities for improvements. Section 6 considers this issue and discusses our experience in applying DMAIC, a *Six Sigma* framework for statistical monitoring and control of processes, which is an important part of our automation and control architecture. Section 7 presents the idea of using test management tools and how they can aggregate value to the process of automation. Finally, Section 8 concludes this chapter, summarising its main objectives.

2. The Handset Test Domain

Network tests for handsets intend to verify if new devices are able to perform a set of operations pre-defined by carriers over the network. The majority of these operations, such as handovers and reselection of cells, are default operations of any network carrier. However, specific operations can be offered for just one carrier. Note that this kind of evaluation does not test the user experience features of the device, user interface, performance of data applications, battery life and others. Network tests are mainly focused on conformance (protocol performance, application enabler and radio components) and interoperability testing.

Each of the individual layers (Fig. 1) of a handset implementation requires a different battery of tests, which are performed in specific scenarios. A scenario is a collection of cell configurations and other options, which can represent parts of artificial or "real-life" networks. There are specific tools (e.g., Sagem, Nemo and Tems) for network information capture, which are able to read and save network configurations so that we can simulate scenarios of these networks in laboratory.

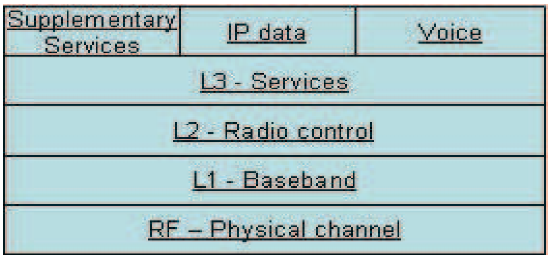


Fig. 1. Layers of a handset architecture

The scenarios that we are working with are based on the GSM standard. GSM is a cellular network, which means that handsets connect to it by searching for cells in their immediate vicinity. The main feature of this network is its operation frequency ranges. Most GSM networks operate in the 900 MHz or 1800 MHz bands. Some countries in the Americas (including the United States and Canada) use the 850 MHz and 1900 MHz bands because the 900 and 1800 MHz frequency bands were already allocated.
Handsets usually have to be evaluated in more than one scenario. However, some test cases are not appropriate to every scenario. For example, consider a scenario with two cells: $cell_1$ and $cell_2$. Using only the frequency ranges of 900 MHz and 1800 MHz, we could set three different scenarios: ($cell_1$= $cell_2$= 900MHz), ($cell_1$ = $cell_2$ = 1800MHz) and ($cell_1$ = 900MHz, $cell_2$ = 1800MHz). However frequency range is only one of the parameters that can be set to cells, therefore a considerable variation of scenarios can be specified.
Considering that we have hundreds of test cases to be performed in several different scenarios, the use of *field-testing*, which uses real carriers' networks, can be expensive, especially in terms of travel cost, time and potential disruption of real networks. The use of *field-testing* also makes it impossible to guarantee the reproducibility of the same test environment in different runs of the same test. Such facts have motivated the development of simulation environments for wireless networks, which support the performance of a significant variety of network tests. This section, in particular, describes our experience with the Anite/SAS network simulator (Anite, 1999).
The Anite/SAS environment enables the handset communication within any network configuration of up to eight base stations under laboratory conditions. A PC controller runs the network simulation software and controls the base station modules (in our case, two Agilent 8960 test sets). The test sets interact with the handset being tested via a normal RF interface. To test end-to-end communication, the PC can be connected to the Internet, allowing data to be transmitted to the handset. Using this configuration, for example (Fig. 2), we are able to emulate a two-cell scenario and perform tests that involve handovers, cell selection and cell reselection for circuit-switched, packet-based and dual transfer mode (DTM) services over GSM.

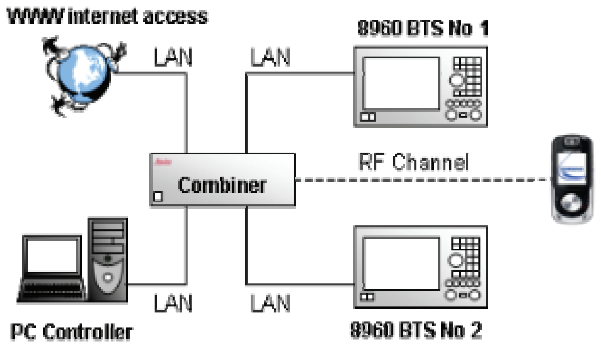


Fig. 2. Test simulation environment

An important feature of network simulation environments is their flexibility in configuring cells. Currently we are creating artificial scenarios via the simulator's cells editor. Using this editor, for example, we can set the cell identity, channel descriptions and frequency list. The collection of all these parameters defines a cell and the collection of all cells defines a scenario. Scenarios can be saved, loaded and several different scenarios can be used to test a handset. In an extension of this project, we intend to use "real-life" scenarios, creating a database with real network information of several Brazilian cities.,

3. A Semantic Standardization for Tests Cases

3.1 A Classification for Network Tests

The main sources of information to create a suite of tests are the carrier's requirements. Indeed, if carriers account for creating and maintaining wireless network services, it is natural that they also provide the requirements that a handset must implement to operate over their networks. However, the tests provided by different carriers are not uniform and present several variations related to their procedures. Thus, we needed to carry out a process of information extraction and consolidation to create a test suite proposal. There are some efforts in the direction of unifying the performance of network tests via the creation of certifications centers. These efforts vary with different technologies and geographies. For example, the procedure for evaluating the GSM technology (Rahnema, 1993) is managed by the PCS Type Certification Review Board in the U.S., while the Global Certification Forum does this in other countries. Furthermore, groups like that also elaborate documents, such as the 3GPPTS 51.010-1, which contains collections of test cases for handset validation. We have also used similar documents to support and aggregate content to our test suite.
In order to consolidate all the information from carriers and certification documents we have elaborated a classification that currently has 25 test batteries. The test cases were mainly classified according to technological aspects and kind of services that they are supposed to evaluate. Examples of test batteries are: Air Signaling Tests (RF), Enhanced Messaging Service, Supplementary Services, Data Connection, Multimedia Messaging Service and Wireless Application Protocol.

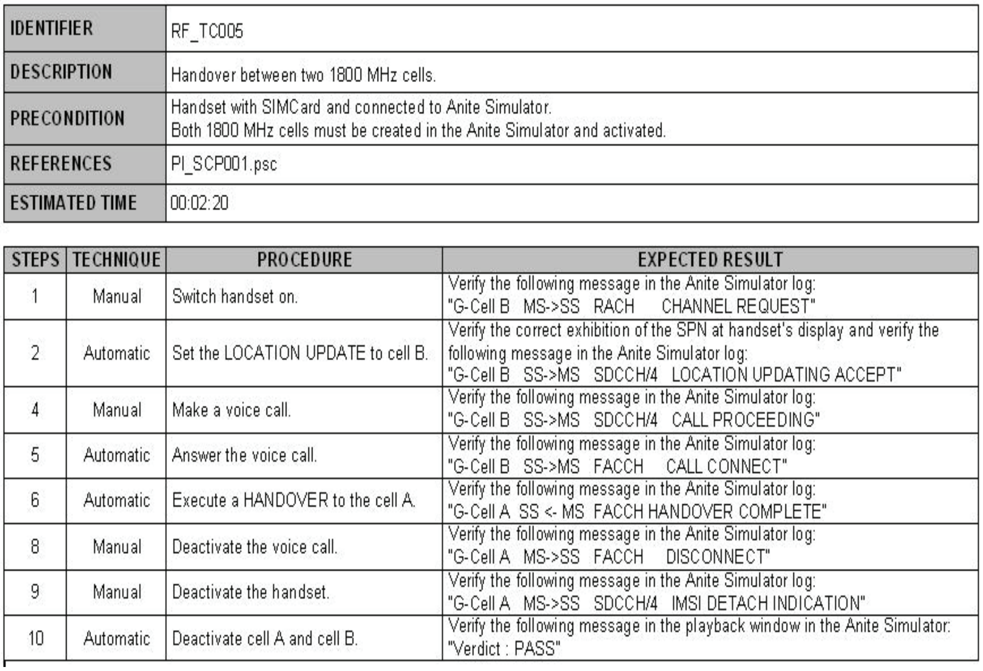
3.2 The Test Case Specification

The next step, after the test cases classification, is to identify and organize the common concepts related to test cases, so that they can be represented in a unified way. From our analysis, we have defined the following concepts (Fig. 3):

- Test Case Identifier – a unique key that identifies a test case (TC) among the entire test case collection. The pattern proposed in our project is <xx_TCyyy>, where "xx" represents the battery identifier and "yyy" represents the TC number. For example, SM_TC003 is the third test case from the Short Message (SM) battery;
- Description – a short description of the main function of a test case;
- Preconditions – required initial conditions to the execution of a test case;
- References – indicate the files, such as the test scripts, associated with a test case;
- Estimated time – average time required to perform the test case;
- List of test case steps – steps are represented by the 4-tuple <step-number, technique, procedure, expected-result>, where step-number is a sequential integer value; technique specifies if the step is manual or automatic; procedure is the description of the step; and expected-result is any verification to be performed on the handset display or on the simulator execution log.

3.3 Mapping Test Cases to Simulator Scripts

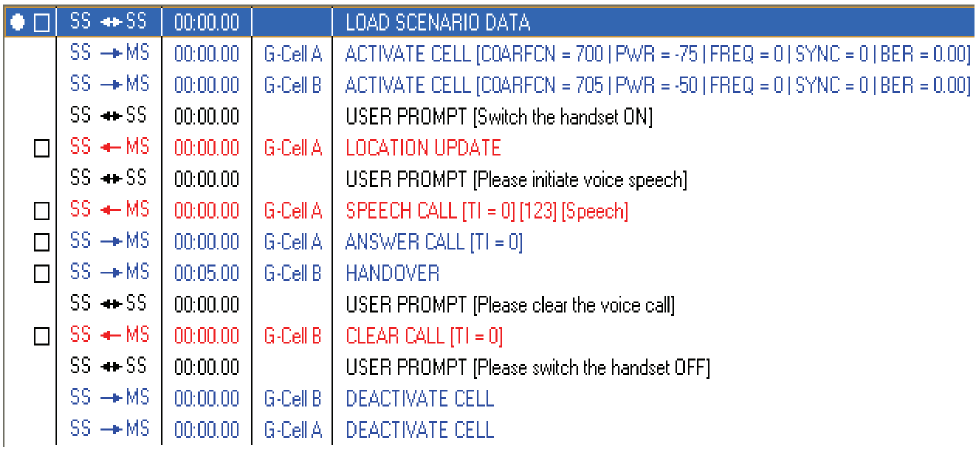
Before running the test in our simulation environment, we need to map each test case to a script, using the simulator language. An example of a script, which represents the mapping of the RF_TC005 test case (Fig 3), is illustrated in follow (Fig. 4).



IDENTIFIER	RF_TC005
DESCRIPTION	Handover between two 1800 MHz cells.
PRECONDITION	Handset with SIMCard and connected to Anite Simulator. Both 1800 MHz cells must be created in the Anite Simulator and activated.
REFERENCES	PI_SCP001.psc
ESTIMATED TIME	00:02:20

STEPS	TECHNIQUE	PROCEDURE	EXPECTED RESULT
1	Manual	Switch handset on.	Verify the following message in the Anite Simulator log: "G-Cell B MS->SS RACH CHANNEL REQUEST"
2	Automatic	Set the LOCATION UPDATE to cell B.	Verify the correct exhibition of the SPN at handset's display and verify the following message in the Anite Simulator log: "G-Cell B SS->MS SDCCH/4 LOCATION UPDATING ACCEPT"
4	Manual	Make a voice call.	Verify the following message in the Anite Simulator log: "G-Cell B SS->MS SDCCH/4 CALL PROCEEDING"
5	Automatic	Answer the voice call.	Verify the following message in the Anite Simulator log: "G-Cell B SS->MS FACCH CALL CONNECT"
6	Automatic	Execute a HANDOVER to the cell A.	Verify the following message in the Anite Simulator log: "G-Cell A SS <- MS FACCH HANDOVER COMPLETE"
8	Manual	Deactivate the voice call.	Verify the following message in the Anite Simulator log: "G-Cell A MS->SS FACCH DISCONNECT"
9	Manual	Deactivate the handset.	Verify the following message in the Anite Simulator log: "G-Cell A MS->SS SDCCH/4 IMSI DETACH INDICATION"
10	Automatic	Deactivate cell A and cell B.	Verify the following message in the playback window in the Anite Simulator: "Verdict : PASS"

Fig. 3. Example of test case specification. This one refers to test case number 5 from the Radio Frequency battery.



● ☐	SS ↔ SS	00:00.00		LOAD SCENARIO DATA
	SS → MS	00:00.00	G-Cell A	ACTIVATE CELL [COARFCN = 700 | PWR = -75 | FREQ = 0 | SYNC = 0 | BER = 0.00]
	SS → MS	00:00.00	G-Cell B	ACTIVATE CELL [COARFCN = 705 | PWR = -50 | FREQ = 0 | SYNC = 0 | BER = 0.00]
	SS ↔ SS	00:00.00		USER PROMPT [Switch the handset ON]
☐	SS ← MS	00:00.00	G-Cell A	LOCATION UPDATE
	SS ↔ SS	00:00.00		USER PROMPT [Please initiate voice speech]
☐	SS ← MS	00:00.00	G-Cell A	SPEECH CALL [TI = 0] [123] [Speech]
☐	SS → MS	00:00.00	G-Cell A	ANSWER CALL [TI = 0]
☐	SS → MS	00:05.00	G-Cell B	HANDOVER
	SS ↔ SS	00:00.00		USER PROMPT [Please clear the voice call]
☐	SS ← MS	00:00.00	G-Cell B	CLEAR CALL [TI = 0]
	SS ↔ SS	00:00.00		USER PROMPT [Please switch the handset OFF]
	SS → MS	00:00.00	G-Cell B	DEACTIVATE CELL
	SS → MS	00:00.00	G-Cell A	DEACTIVATE CELL

Fig. 4. Simulator playback of the RF_TC005.

Let's comment some details about this script and its relation to the test case (Fig. 3). First, the initial command in the script (LOAD SCENARIO DATA) loads a network scenario, which must be previously defined in the simulator. Specific parameters of this scenario can be changed, according to preconditions of the test case (Fig. 3). For example, this script sets the power value of Cell A to -75dBm and Cell B to -50dBm. Second, automatic steps of the test case can be directly mapped to simulator commands. For example, step 2 (Set the LOCATION UPDATE to cell A) is mapped to the LOCATION UPDATE command. Commands like that are pre-defined in the simulator language. Finally, manual steps are mapped to USER PROMPT commands. These commands are, in fact, requests to testers so that they execute some specific command in the handset, such as *Switch the handset ON*.

3.4 Using a Formal Description

Whereas test cases documentation (Fig. 3) is understandable and simple to human readers, since they use natural language, scripts (Fig. 4) are defined in a proprietary language, whose syntax is defined to increase the performance of the simulator. We need a new description that considers such features together. This means that the description needs to have a good semantic level and be easily manipulated by computational components. Consider, for example, the concept of *ontology* (Gruber, 1995). An ontology defines a set of representational primitives with which to model a domain of knowledge. The representational primitives are typically classes (or sets), attributes (or properties), and relationships (or relations among class members). The definitions of the representational primitives include information about their meaning and constraints on their logically consistent application. In this way, an ontology can also be viewed as an abstraction of data models, analogous to hierarchical and relational models, but intended for modelling knowledge about individuals, their attributes, and their relationships to other individuals. For this reason, an ontology is said to be at the semantic level of a system description.
We do not intend to develop a complete ontology for handsets' network test cases. However we can use some fundaments to design a test case specification, which support the execution of a decision process by the test system. One of the main ideas of an ontology is to define the classes, attributes and relationships of a domain (Guarino, 1995). For that end, consider the BNF metasyntax, used to express context-free grammars and, thus, a formal way to describe formal languages. Using BNF we can create the following specification:

```
<test-case> ::=  <tc-head> <list-of-steps>
<tc-head> ::= <identifier> <description> <list-of-preconditions>
              <list-of-references> <estimated-time>
<list-of-steps> ::=  <list-of-steps> | <step>
<step> ::= <step-number> <technique> <procedure> <expected-result>
```

This is just the first level of a specification. The complete specification must decompose the classes to primitive concepts. Note that descriptions like these allow us to abstract away from data structures and implementation strategies. However we can approximate such a description to computational structures if we apply a direct mapping from BNF to a marked language, such as XML. For example, consider the XML version of the above BNF description:

```
TEST-CASE :: = <test-case>
                   <tc-head > TC-HEAD </tc-head>
                   <list> <step> STEP </step> </list>
               </test-case>
TC-HEAD ::=    <tc-head id="SYMBOL" description="NAME" estimated-time = "TIME">
                   <list> <precondition> PRECONDITION </precondition> </list>
                   <list> <reference> REFERENCE </reference> </list>
               </tc-head>
STEP ::=       <step step-number="NUMBER" technique="SYMBOL">
                   <procedure> PROCEDURE </procedure> </list>
                   <expected-result> EXPECTED-RESULT </ expected-result >
               </step>
```

Several ontological concepts are represented in such a description. For example, TEXT-CASE, TC-HEAD and STEP are examples of classes. TC-HEAD has three attributes: *id*, *description* and *estimated-time*. Finally, the definition structure itself defines the relations among the classes. For example, TEST-CASEs contain one TC-HEAD and a non empty list of STEPs. Modern programming languages, like Java, provide a strong support for the manipulation of XML-based descriptions via, for example, packages to create parsers and objects that represent the classes and attributes defined in the description. The next section shows how this kind of description can be used during the test process.

4. Planning the Test Suite

As discussed before, test automation is an appropriate technique to deal with the current handset test scenario. We have observed that the common approach used in test automation is to employ pre-defined recorded test cases, which can be created and edited in tools provided by simulation environments. Such tools also enable the building of different and more complex tests from previous ones. However, this entire process is carried out manually. Thus, we have an automatic execution of tests, but using fixed sequences of test cases, so that the sequence is not automatically adapted to specific scenarios.
The idea explored in this section is to develop a mechanism that autonomously creates test cases based on devices and environment features. This mechanism must specify the most appropriate sequence of test cases for a specific handset, also minimizing the total test time. For that end we have investigated the *Artificial Intelligence* (AI) *Planning* technique (Ghallab et al., 2004), which is used to create optimal sequences of actions to be performed in some specific situation. One of the advantages of AI planning is its modelling language, which has a similar syntax to the language used for test case description. Furthermore, the language is open, providing a direct way to represent states (situations of tests), goals (desired results of tests) and actions (steps of each test case). This section details these issues, starting with a brief introduction to AI planning and its fundamentals.

4.1 Fundamentals of AI Planning

AI Planning can be viewed as a type of problem solving in which a system (Planner) uses beliefs about actions and their consequences to search for a solution over a space of plans or states. The key idea behind planning is its open representation of states (e.g., test case scenarios), goals (e.g., expected results of test cases) and actions (e.g., steps of a test case). States and goals are represented by sets of sentences, and actions are represented by logical descriptions of preconditions and effects. This enables the planner to make direct connections between states and actions. The classical approach to describe plans is via the STRIPS language (Fikes & Nilsson, 1971). This language represents states by conjunctions of predicates applied to constant symbols. For example, we can have the following predicates to indicate the initial state of a handset network: FreqRange($cell_a$,900) $\wedge$ FreqRange($cell_b$,1800). Goals are also described by conjunctions of predicates, however they can contain variables rather than only constants. Actions, also called operators, consist of three components: action descriptions, conditions and effects. Conditions are the (partial) mandatory states to the application of the operator and effects are the (partial) final state after its application. Using these basic definitions, the representation of plans can be specified as a data structure consisting of the following four elements (Wilkins, 1994):

- A set of plan steps, each of them representing one of the operators;
- A set of tasks ordering constraints. Each ordering constraint is of the form S_i » S_j, which is read as "S_i before S_j" and means that step S_i must occur sometime before step S_j;
- A set of variable binding constraints. Each variable constraint is of the form v = x, where v is a variable in some task and x is either a constraint or another variable; and
- A set of *causal links*. A causal link is written as $\{T_i \rightarrow T_j\}^c$ and read as "Ti achieves c for T_j". Causal links serve to record the purpose(s) of steps in the plan. In this description, a purpose of T_i is to achieve the precondition c of T_j.

Using these elements, we can apply a *Partial-Order Planning* (POP) algorithm, which is able to search through the space of plans to find one that is guaranteed to succeed. More details about the algorithm are given later.

4.2 Test Case as Planning Operators

Let us consider now the process of mapping a test case to a planning method. One of the parts of this research is to codify each of the user cases in a plan operator. As discussed before, a plan operator has three components: the action description, the conditions and the effects. Therefore we need to find information inside user tests to create each of these components. Starting by the action description, this component is just a simple and short identifier that does not play any active role during the decision process of the planner. For its representation we are issuing identifiers that relate each action description with a unique test case TC. Second, we need to specify the operator conditions. The test case definition has an attribute called precondition that brings exactly the semantic that we intend to use for operator conditions. Note, however, that this description is a natural language sentence that must be translated to a logic predicate before being used by the planner. Finally, the effects do not have a direct mapping from some component of the test case descriptor. However, each step of the test case (e.g., Table 1) is one action that can change the current state of the domain. Consequently, the effects can be defined by the conjunction of the expected results of each step, if this step has the feature of changing the domain status. Another observation is that sequential steps can change a given status. Thus, only the last change of a status must be considered as an effect. As discussed for preconditions, this conjunction of plan steps results also must be codified in logic predicates.

Step	Procedure	Expected result
01	Switch handset on.	"G-Cell B MS->SS RACH CHANNEL REQUEST"
02	Set the LOCATION UPDATE to cell B.	"G-Cell B SS->MS SDCCH/4 LOCATION UPDATING ACCEPT"
03	Make a voice call.	"G-Cell B SS->MS SDCCH/4 CALL PROCEEDING"
04	Execute HANDOVER to the cell A.	"G-Cell B SS->MS FACCH CALL CONNECT"
05	Execute HANDOVER to the cell A.	"G-Cell A SS <- MS FACCH HANDOVER COMPLETE"
06	Deactivate voice call.	"G-Cell A MS->SS FACCH DISCONNECT"
07	Deactivate the handset.	"G-Cell A MS->SS SDCCH/4 IMSI DETACH INDICATION"
08	Deactivate cells and B.	"Verdict : PASS"

Table 1. Partial specification of a test case.

A close investigation into the test cases shows that they are not dealing with operations related to changes in some of the network domain parameters, such as frequency band or number of transceivers in each cell. To have a complete automation of the handsets' network tests, the planner needs special operators that are able to change such parameters. Considering this fact, the set of automation operators can be classified into two groups: the *Test Case Operators* (TCO) and the *Domain Modify Operators* (DMO). The use of both operators is exemplified in the figure below (Fig. 5).

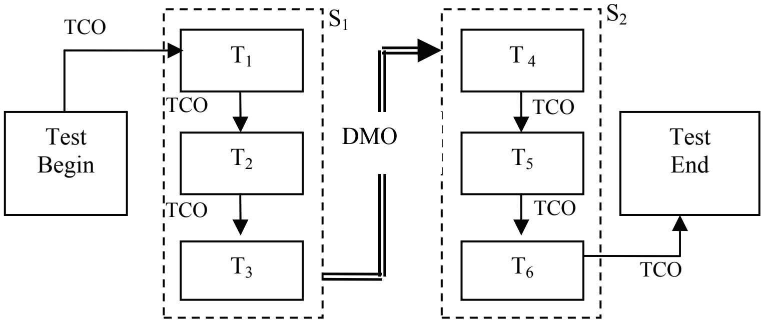


Fig. 5. Test case for a handover between two cells of 1800 MHz.

During the planning process, a planner should normally consider all the test case operators (TCO) of a scenario S_1 before applying a DMO and change to S_2. Actually the number of changes between scenarios (or use of DMO's) can be a measure of the planner performance.

A complete test case plan is a sequence of these operators in which every precondition, of every operator, is achieved by some other operator.

4.3 Planning Algorithm

The reasoning method investigated during this project is described via a *Partial-Order Planning* (POP) algorithm (Nguyen & Kambhampati, 2001). A POP planner has the ability of representing plans in which some steps are ordered with respect to each other while other steps are unordered. This is possible because they implement the principle of least commitment (Weld, 1994), which says that one should only make choices about things that it currently cares about, leaving the other choices to be worked out later.
The pseudocode below (1 to 4) describes the concept of a POP algorithm, showing how it can be applied to the handsets' network test domain. This pseudocode was adapted from the original POP algorithm (Russel & P. Norvig, 2002).

```
function POP(testBegint,testEnd,operators) return plan
  plan ← Make-Basic-Plan(testStart, testFinish)
  loop do
    if Solution?(plan) then return plan                                  (1)
    $T_i$, c ← SELECT-SUBGOAL(plan)
    CHOOSE-OPERATOR(plan,operators, $T_{i,}c$)
    RESOLVE-THREATS(plan)
  end
```

Code (1) shows that the POP is a loop that must create a plan from a *testBegin* state (no tests performed) to a *testEnd* state (all tests performed). For that end, the loop extends the plan by achieving a precondition *c* of a test case T_i, which was selected as a subgoal of *plan*.
Code (2) accounts for the selections of this subgoal:

```
function SELECT-SUBGOAL(plan) returns $T_{i,}c$
  pick a test state$T_i$ from TEST_STEPS(plan)                           (2)
    with a precondition c that has not been achieved
 return step $T_{i,}c$
```

Code (3) details the choice of an operator T_{add} (TCO or DMO), which achieves *c*, either from the existing steps of the plan or from the pool of operators. Note that the causal link for *c* is recorded together with an ordering constraint. If T_{add} is not in TEST_STEPS (T_i), it needs to be added to this collection.
We can improve the SELECT-SUBGOAL function by adding a heuristic to lead the choice of a test goal. During the test process, if one of the tests fails, the problem must be fixed and all the test collection carried out again. Imagine an extreme scenario where a problem is detected in the last test. In this case, the test process will take about twice the normal time to be performed if any other problem is found. Considering this fact, the SELECT-SUBGOAL function could identify and keep track of the tests where errors are more commons. This could be implemented via a module of learning (Langley & Allen, 1993), for example. Based

on this knowledge, the function should give preference to tests with a bigger probability to fail because then the test process will be interrupted earlier.

```
**procedure** CHOOSE-OPERATOR(*plan,operators*, $T_{i,}c$)
  **choose** a step $T_{add}$ from *operators* or TEST_STEPS (*plan*) that has  *c* as an effect
  **if** there is no such step **then fail**
  add the causal link $\{T_{add} \rightarrow T_i\}^c$ to LINKS(*plan*)
  add ordering constraint $T_{add}$ » $T_i$ to ORDERINGS(*plan)*                                    (3)
  **if** $T_{add}$ is a newly added step from *operators* **then**
    add $T_{add}$ to TEST_STEPS ($T_i$)
    add Start » $T_{add}$ » Finnish to ORDERINGS(*plan*)
  **end**
```

The last procedure, Code (4), accounts for resolving any threats to causal links. The new step T_{add} may threaten an existing causal link or an existing step may threaten the new causal link. If at any point the algorithm fails to find a relevant operator or fails to resolve a threat, it backtracks to a previous choice point.

```
**procedure** RESOLVE-THREATS(*plan*)
  **for each** $T_{threat}$ that threatens a link  $\{T_i \rightarrow T_j\}^c$ in LINKS(*plan*) **do**
    **choose** either
      *Promotion*: Add $S_{threat}$ » $S_i$ to ORDERINGS(*plan*)                                (4)
      *Demotion*: Add $S_j$ » $S_{threat}$ to ORDERINGS(*plan*)
    **if not** CONSISTENT(plan) **then fail**
  **end**
```

POP implements a regression approach. This means that it starts with all the handset network tests that need to be achieved and works backwards to find a sequence of operators that will achieve them. In our domain, the final state “Test_End” will have a condition in the form: Done(TC_1) $\wedge$ Done(TC_2) $\wedge \ldots \wedge$ Done(TC_n), where n is the total number of test cases (TC). Thus, the “Test_Begin” has a effect in the form: $\neg$Done(TC_1) $\wedge$ $\neg$Done(TC_2) $\wedge \ldots \wedge$ $\neg$Done(TC_n). In this way, all TCOs must have an effect in the form Done(TC_i), where i is an integer between 1 and n. The other effects of each TCO change the current plan state, restricting the operations that can be used. If there is a fail, this could indicate that a DMO must be applied to change the scenario (network parameters).
According to [VanBrunt, 1993], the test methodology to handset network, which is based on GSM, are focused on conformance evaluations used to validate the underlying components of the air interface technology. However, the launch of new technologies, such as the WCDMA (*Wideband Code Division Multiple Access*), could change the current way that tests are performed, requiring, for example, a more progressive and integrated approach to evaluation of user equipments. Considering this fact, features like maintenance and extensibility must be considered during the development of test automation. In our approach, any new requirement can be easily contemplated via the creation of new operators. New network configurations could also be defined via the definition of new scenarios and DMOs that change the conditions of test applications.

5. Automation and Autonomic Architectures and Autonomic Computing

Automation test architectures intend to support the execution of tests by computational processes, independently from human interference. This automation considers a pre-defined and correct execution, so that concepts such as adaptation and self-correction are not generally contemplated. The second and more complex level of automation is characterised by systems that present some level of autonomy to take decisions by themselves. This kind of autonomic computing brings several advantages when compared with traditional automation and several approaches can be employed to implement its fundaments. All these issues are detailed in this section.

5.1 The CInMobile Automation Tool

We have implemented an automation environment by means of CInMobile (*Conformance Instrument for Mobiles*). This tool aims to improve the efficiency of the test process that is being carried out over the network simulation environment. The CInMobile architecture is illustrated in follow (Fig. 6), where its main components and their communication are presented. As discussed before (see Fig. 4), simulator scripts can have prompt commands that request some intervention from human testers. The first step of our approach was to lead such commands to the serial port so that they could be captured by an external process called *Handset Automator* (HA), running in a second computer. In this way, the HA receives prompt commands, as operations requests, and analyzes the string content to generate an appropriate operation, which is sent to the handset in evaluation. The HA can use pre-defined scripts from the *Handset Script Base* during this operation. The HA also accounts for sending synchronization messages to the SAS software, indicating that the requested handset operation was already carried out.

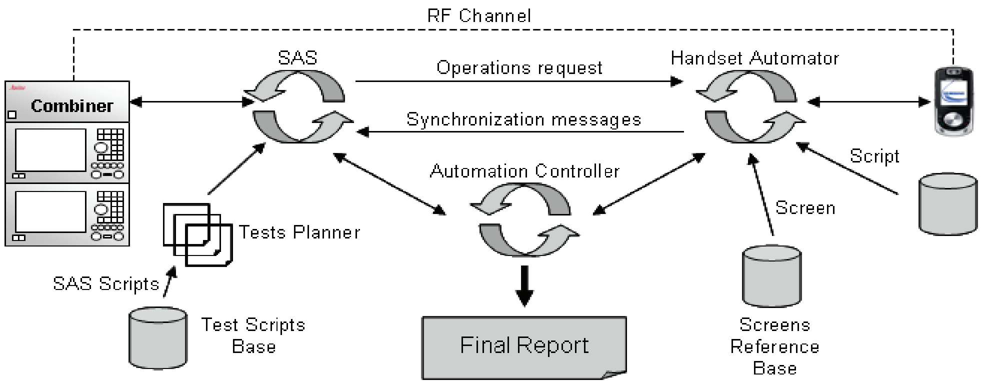


Fig. 6. CinMobile Architecture.

Both SAS and HA generate logs of the test process. However, results of several handset operations, over the simulated wireless network, can be identified via comparisons of resulting handset screens to reference images. Our architecture also supports this kind of evaluation and its results are considered together with the SAS and HA logs. In fact, each of these results (images comparison, SAS and HA logs) brings a different kind of information that must be analyzed to provide useful information about a particular test case.

One of the functions of the *Automation Controller* (AC) process is to consolidate all the test process resulting information, performing a unified analysis so that a unique final report can be generated (Fig. 7). The advantage of this approach is that we are passing the responsibility of reporting the test results from the human testers to an automatic process. Furthermore, the *Final Report Generator* (see Fig. 7) can be used to customize the final report appearance (e.g., using templates) according to user needs.

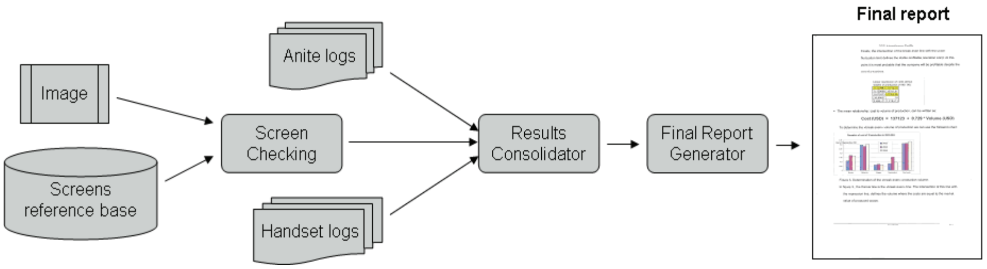


Fig. 7. Test report generation process.

A second important AC function is associated with the control of script loading in simulator. This is useful, for example, because some of the tests must be repeated several times and there is an associated approval percentage. For instance, consider that each test is represented by the 3-tuple $<t,\eta,\varphi>$, where t is the test identifier, η is the number of test repetitions for each device, and φ is the approval percentage. Then a 3-tuple specified as $<t_1,12,75\%>$ means that the t_1 must be performed 12 times and the device will only be approved if the result is correct at least in 9 of them. However, if the 9 first tests are correct, then the other 3 do not need to be executed, avoiding waste of time.

A last AC core function is to manage the performance of several handsets automators. Some tests cases, mainly found in the Bluetooth battery, require the use of two of more handsets to evaluate, for example, operations of voice conference (several handsets sharing the same *Voice Traffic Channel*) over the network. In this scenario, the AC accounts for the tasks of synchronization, among simulator and handset automators, and consolidation of multiple handset logs during the generation of test reports.

5.2 Autonomic Computing

The test process, carried out by our Test Center team, currently provides a percentage of about 53% of automation. However the level of automation provided to such tests is not enough to support a total *autonomic execution*, that is, without the presence of human testers. Considering this fact, our aim is to implement new methods that enable a total autonomic execution, so that tests can be executed at night, for example, increasing the use time of the simulation environment per day and, consequently, decreasing the total test time and operational cost. Our fist experiment was carried out with the E-mail battery. This battery was chosen because it is currently 100% automatic and represents about nine hours of test time. Thus, we could decrease in about one day the test process if such a battery could be performed at night. These new sets of tests, that can run without the human presence, we call *autonomic tests*.

The study of autonomic computing was mainly leaded by IBM Research and a clear definition is (Ganek & Corbi2003): "Autonomic computing is the ability of systems to be more self-managing. The term autonomic comes from the autonomic nervous system, which controls many organs and muscles in the human body. Usually, we are unaware of its workings because it functions in an involuntary, reflexive manner -- for example, we do not notice when our heart beats faster or our blood vessels change size in response to temperature, posture, food intake, stressful experiences and other changes to which we're exposed. And, by the way, our autonomic nervous system is always working".
An autonomic computing paradigm must have a mechanism whereby changes in its essential variables can trigger changes in the behavior of the computing system such that the system is brought back into equilibrium with respect to the environment. This state of stable equilibrium is a necessary condition for the survivability of a system. We can think of survivability as the system's ability to protect itself, recover from faults, reconfigure as required by changes in the environment, and always to maintain its operations at a near optimal performance. Its equilibrium is impacted by both the internal and external environment.
An autonomic computing system (Fig. 8) requires: (a) sensor channels to sense the changes in the internal and external environment, and (b) return channels to react to and counter the effects of the changes in the environment by changing the system and maintaining equilibrium. The changes sensed by the sensor channels have to be analyzed to determine if any of the essential variables has gone out of their viability limits. If so, it has to trigger some kind of planning to determine what changes to inject into the current behavior of the system such that it returns to the equilibrium state within the new environment. This planning would require knowledge to select the right behavior from a large set of possible behaviors to counter the change. Finally, the manager, via return channels, executes the selected change. Thus, we can undersdand the operation of an aunomic system as a continuous cycle of sensing, analyzing, planning, and executing; all of these processes supported by knowledge (Kephart & Chess, 2003).

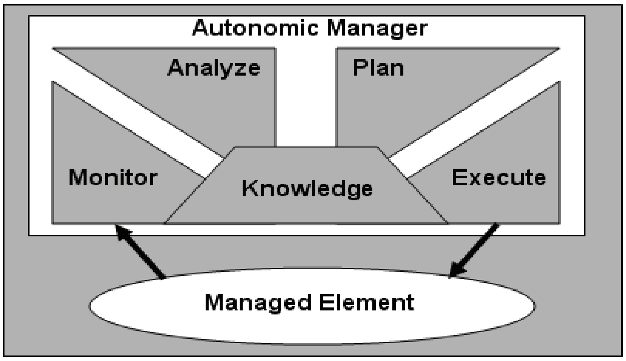


Fig. 8 – The classical autonomic computing architecture.

This classical autonomic architecture (Fig. 8) acts in accordance with high-level policies and it is aimed at supporting the principles that govern all such systems. Such principles have been summarized as eight defining characteristics (Hariri et al, 2006):

- Self-Awareness: an autonomic system knows itself and is aware of its state and its behaviour;
- Self-Protecting: an autonomic system is equally prone to attacks and hence it should be capable of detecting and protecting its resources from both internal and external attack and maintaining overall system security and integrity;
- Self-Optimizing: an autonomic system should be able to detect performance degradation in system behaviour and intelligently perform self-optimization functions;
- Self-Healing: an autonomic system must be aware of potential problems and should have the ability to reconfigure itself to continue to function smoothly;
- Self-Configuring: an autonomic system must have the ability to dynamically adjust its resources based on its state and the state of its execution environment;
- Contextually Aware: an autonomic system must be aware of its execution environment and be able to react to changes in the environment;
- Open: an autonomic system must be portable across multiple hardware and software architectures, and consequently it must be built on standard and open protocols and interfaces;
- Anticipatory: an autonomic system must be able to anticipate, to the extent that it can, its needs and behaviours and those of its context, and to be able to manage itself proactively.

An autonomic manager component does not need to implement all these principles and the choice for one or more depends on the kind of managed element that we are working with. For example, the implementation of the Self-Protecting principle only makes sense if we are working with systems that require a high level of security, such as Internet or network systems. In our case, in particular, we are initially focusing our investigation on three of these principles: Self-Awareness, Self-Healing and Contextual Awareness.

We can find several similarities if we compare the classical autonomic computing architecture to the structure (Fig. 9) of an intelligent utility-based agent (Russel & Norvig, 2002). First, both systems present specific components to sense (sensors channels) the environment (or managed element) and to execute operations (return channels) on such an environment. Second, the knowledge of autonomic computing architectures can be compared to the knowledge of agents (knowledge about its state, how the world evolves, what its actions do and utility of decisions). Finally, in both cases, such knowledge supports the process of analysis and planning of actions, which are going to be executed on the environment.

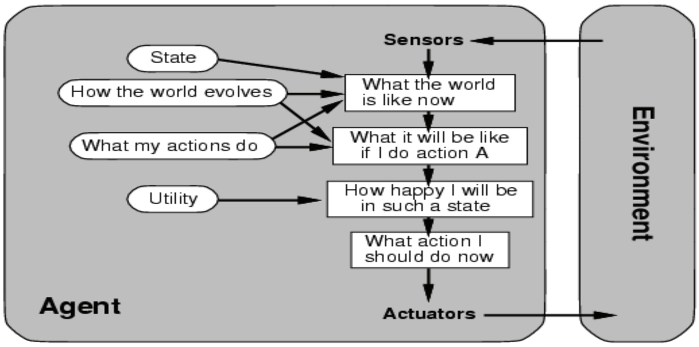

Sensors
State
How the world evolves
What the world is like now
What my actions do
What it will be like if I do action A
Utility
How happy I will be in such a state
What action I should do now
Agent
Actuators
Environment

Fig. 9. Utility-based agent structure.

The autonomic manager module, which is in development by our team, is based on this agent approach. For that end, we are specifying the five components of the autonomic architecture (monitor, knowledge base, analyzer, planner and executor) in the following way:

- Monitor – this component is a listener of exceptional events (e.g., Java exceptions), variable content updating (e.g., serial port identifiers) and temporal references (when specific no-exceptional start events are flagged, a timer is started to check if such events finish within pre-defined intervals) ;
- Knowledge base – set of objects that represent both a collection of facts and rules. Facts abstract the current status of the managed element, whereas rules mainly indicate what the systems must do if some event is flagged;
- Analyzer – events received by the monitor need to be analysed in a holistic way. This means, rather than analysing some event individually, we must perform such analysis considering the current status of the knowledge base. For this analysis we use a set of analysis rules, which are part of the knowledge base;
- Planner – analysis rules insert new facts into the knowledge basis, which are used by the planner to decide the actions that are going to be performed. The planner is in fact a set of rules, which we call decision rules;
- Executor – this component has a set of methods that are able to modify the state of the managed element. This set of methods is limited and must be predefined, according to a previous study of the system.

The knowledge base, analyzer and planner are being specified as a production system and for that we are using JEOPS - *Java Embedded Object Production System* (Filho & Ramalho, 2000). The main reason for its use is its first-order forward-chaining inference approach, which starts with the available facts and uses inference rules to extract more facts until an appropriate action is reached. Another reason is its complete integration with Java, which is used in the development of our software. Such components together provide the mechanisms to support the principles of Self-Awareness (knowledge base represents an internal state and updates it), Self-Healing (rules identify problems and trigger recovering methods) and Context Awareness (rules consider the current context once events are analysed in a holistic way).

6. Automation Monitoring and Control via DMAIC Concepts

This section discusses the use of DMAIC (Define, Measure, Analyze, Improve, and Control) (Simon, 2007), a Six Sigma (Harry, 1998) framework based on measures and statistical analysis, which has commonly been applied during several stages of software development (Biehl, 2004). We show that, using DMAIC, we could be able to both find out automation process failures and identify potential points in a test process that require a review. Furthermore, we could monitor the quality of the test process, as discussed below.

6.1 Six Sigma and DMAIC Concepts

Six Sigma is a set of practices to systematically improve processes by eliminating its defects. For that, Six Sigma stresses two main points: processes can always be measured, analyzed, improved and controlled; and continuous efforts to reduce variation in process outputs are essential to business success.

In this discussion we are interested in the statistical fundaments of Six Sigma. In fact, one of the features of its frameworks is the use of measurements and statistical analysis. However, it is a mistake to view the core of Six Sigma frameworks as statistics; an acceptable Six Sigma project can be started with only rudimentary statistical tools. The Six Sigma idea is very similar to SPC (Statistical Process Control) analysis (Florac & Carleton, 1999), which identifies both the location of problems (producing defectives) and whether or not you can cost-effectively fix them.

Six Sigma provides two main frameworks, which can be better applied according to the scenario that we have. Such scenarios are: (1) there is no process at all (note that a bad process is as good as no process); and (2) there is already existing process(es) that is working reasonably well. In the first scenario, whose focus is on process design, Six Sigma suggests the use of the DMADV framework. DMADV is summarized by the following ideas:

- **Define** the project goals and customer (internal and external) deliverables;
- **Measure** and determine customer needs and specifications;
- **Analyze** the process options to meet the customer needs;
- **Design** (detailed) the process to meet the customer needs;
- **Verify** the design performance and ability to meet customer needs.

In the second scenario, whose focus is on significant process improvements, Six Sigma suggests the use of the DMAIC framework. DMAIC stands for:

- **Define** process goals in terms of key critical parameters (i.e. critical to quality or critical to production) on the basis of customer requirements;
- **Measure** the current process performance in context of goals;
- **Analyze** the current scenario in terms of causes of variations and defects;
- **Improve** the process by systematically reducing variation and eliminating defects;
- **Control** future performance of the process.

Sometimes a DMAIC application may turn into a DMADV application because the process in question requires complete re-design to bring about the desired degree of improvement. Such a discovery usually occurs during the improvement phase of DMAIC. In our case we have decided for DMAIC because we already have a process and our intention is the improvement of this process and detection of its problems.

6.2 DMAIC Phases Application

As discussed before, Six Sigma specifies more than one problem-solving framework, as processes and situations can vary on their nature. We have decided for DMAIC because it is more appropriate for existing processes. In this way, this section summarizes the role of each DMAIC phase and details how we have specified each of these phases to work during the measure and analysis of our test process.

Define Phase

The main role of the Define phase is to formally specify the DMAIC project, its elements, context, importance and purpose. For our test process, in particular, the most relevant output from this phase is the definition of what is the issue that we intend to improve. This issue is the prediction of total test time or test effort for a handset.

To better understand this problem, consider a partial list of handset features (MMS, Bluetooth, EDGE, etc.). Each handset that is going to be evaluated supports a subset of such

features, which are used to compose its test suite. For example, if a handset does not support Streaming, all the tests related to this features are removed from the evaluation set. Thus, we can conclude that the total test time is not the same for all handset models.
When the development unit sends a handset to our test team, they need to know the total test time so that they can plan their next actions. To deal with unpredictable problems, we can add an error limit to our estimations. For example, if we estimate that a battery is performed in 120 minutes, we can add 20% of error and say that it performs in 144 minutes. This approach can generate delays in the development unit process and, consequently, in the process as a whole. To exemplify the problem of simple estimations, i.e. estimations without a real statistical investigation, observe the graph below (Fig. 10) where the Y-axis represents time and X-axis represent the test batteries. This graph represents the results of our first evaluation running and it brings information about the estimated time for each test battery and the real time of its execution[2].

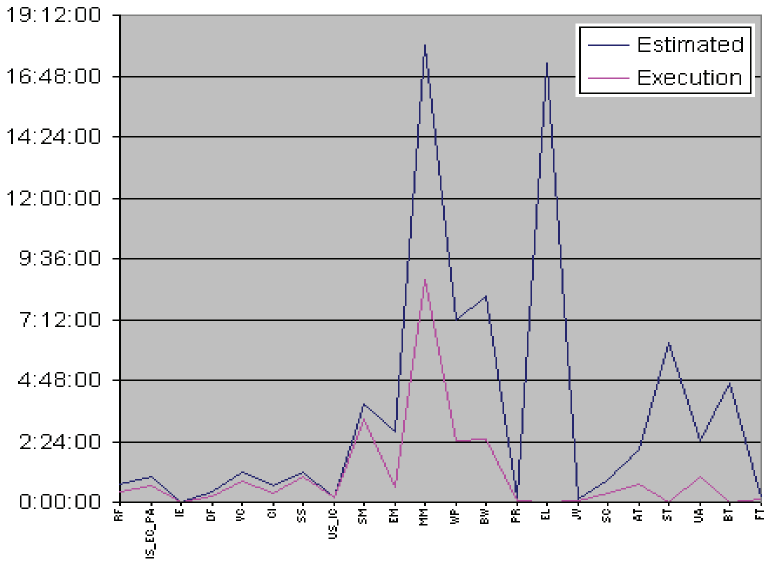


Fig 10. Relation between estimated and execution time.

All our estimates were too high and in some cases (e.g., MM - *Multimedia Messaging Service* - and WP – *Wireless Application Protocol*) the estimates were very deficient. All these estimation problems raise collateral effects to the development unit, which could have defined a better operational plan if they had more effective test time estimates.

Measure Phase

The most important output of the *Measure* phase is the *Baseline*, a historical measurement of indicators chosen to determine the performance before changes made by the DMAIC

[2] Zero Execution time means that the related battery was not applied to the handset in test (e.g., ST – Streaming).

application. These indicators are also measured to assess the progress during the *Improve* phase and to ensure that such an improvement is kept after the *Control* phase. The lead indicator, in our case the prediction of the total test time, is used to track variables that affect its value. Our investigation in this phase is separately performed on each test battery, so that we can perform a more granular investigation on the results. This approach is justified because the batteries have very particular features, which can be better identified and understood if they are analysed in this way. For simplifications, let us consider that there exists only one battery and the total test time is the time to perform this battery. Considering this premise, The *Shewhart Graph* (Florac & Carleton, 1999) (Fig. 12) shows the results related to initial test executions, which were manually carried out in the simulator environment.

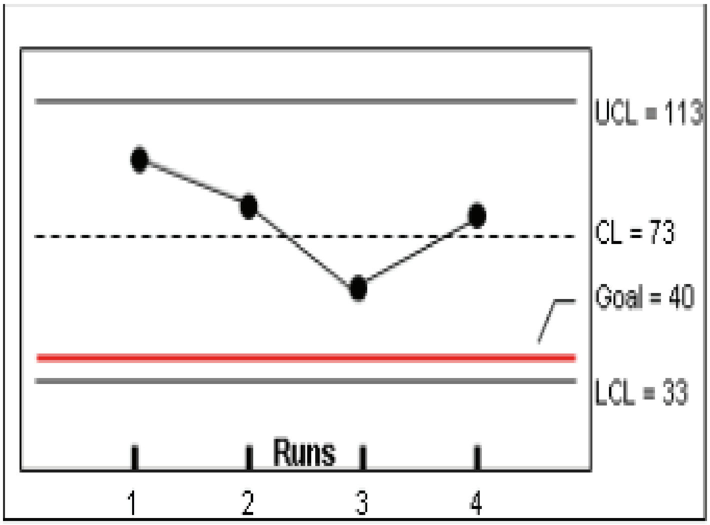


Fig. 12. Shewhart Graph for total time of test runs.

The Shewhart graph contains a Central Line (CL), a Lower Control Line (LCL), an Upper Control Line (UCL) and values related to the issue that we intend to control or improve. In our case, these values are related to the total test time of initial test runs. These data is sequentially plotted along the time, so that we have a historical registry of this information. This is a continuous process and the more data we have, the better will be our Baseline. The CL represents a central value or average of the measures performed on the issue. Both control limits, which are estimates of the process bounds based on measures of the issue values, indicate the limits to separate and identify exceptional points (Humphrey, 1988). The control limits are placed at a distance of 3-sigma (or 3σ) from the central line (sigma σ is the standard deviation).

This graph is an appropriate resource to clarify our objectives in using DMAIC. These objectives and their relation with the graph are:

- More accurate prediction of total test time: this is indicated by the distance between LCL and UCL. The shorter this distance is, the more accurate our prediction will be;

- Test process improvement: the central line (CL) represents our current prevision for the performance of a test run, considering the complete set of test batteries. Test process improvements mean to reduce such a prevision. Our goal is indicated by the red bound line in the graph (goal line);
- Better control: if all total test times are between the control limits, then the test presents only common causes of variation and we can say that the test is in a statistically controlled state (stable test). Differently, if a total test time is out of the control limits, the test presents special causes of variation and we can say that the test is out of statistical control (unstable test).

Before continuing to the next phase, it is important to understand the meaning of common and special causes of variation. Common causes of variation are problems inherent in the system itself. They are always present and affect the output of the process. Examples are poor training and inappropriate production methods. Special causes of variation are problems that arise in a periodic fashion and they are somewhat unpredictable. Examples of special causes are operator error and broken tools. This type of variation is not critical and only represents a small fraction of the variation found in a process (Deming, 1975).

Analysis Phase

The Analysis phase accounts for raising and validating main causes of problems during the handsets network test process. Table 2 summarizes examples of such problems for our domain.

#	**Cause**	**(I)**	**(C)**	**(P)**
1	New testers/operators	L	M	L
2	Operational errors	H	L	H
3	Later found incompatibilities	M	L	M
4	Infra-structure support	M	H	M
5	Bugs in network simulator	H	H	L
6	Test version no longer compatible	M	L	H

Table 2. Summary of problems for the handset test domain

Three parameters are associated with each cause of problems: influence on lead indicator (I), theoretical cost to fix this cause (C), and priority to apply some solution (P). These parameters can assume three qualitative values: low (L), medium (M) and high (H). Such values were set based on our first test experiments and discussions with the technical team. For example, for the first cause we have concluded that the insertion of new testers (externals and trainers) has a low impact on the test process, despite the fact that they only have an initial experience in this process. The cost to carry out a special training is medium, once that we need to allocate a tester engineer to this task. Thus this task is not a priority.

Improve Phase

The Improve phase accounts for selecting and implementing solutions to reduce or eliminate the causes of problems discovered during the Analysis phase. During the Analysis phase, we have already started the discussion about potential solutions for these causes. The following actions are examples of solution that could be implemented in our process: specification of a more granular and formal test process, which focuses mainly on avoiding loss of data (e.g., extensive use of backups) and finding the points where we can carry out tasks in parallel to eliminate dependences and producer-consumer like errors. Solutions are not fixed and they must be adapted to new causes that may appear. This is also one of the reasons to monitor the process even after the application of solutions.

Control Phase

The Control Phase accounts for maintaining the improvement after each new cycle. In our case, this cycle represents the execution of a complete test run. This phase is also related to the process of monitoring the execution of each test, so that new problems can be detected. According to DMAIC, a new problem is raised when the lead indicator, in our case the total test time, is out of the control limits. The graph below (Fig. 13) shows an example of new limit controls that could appear after the application of some solutions. We can observe that the central line is not reached. However we can see some improvement in terms of a new central line (shorter total test time) and narrower control limits. From now on, all the measures must respect such limits.

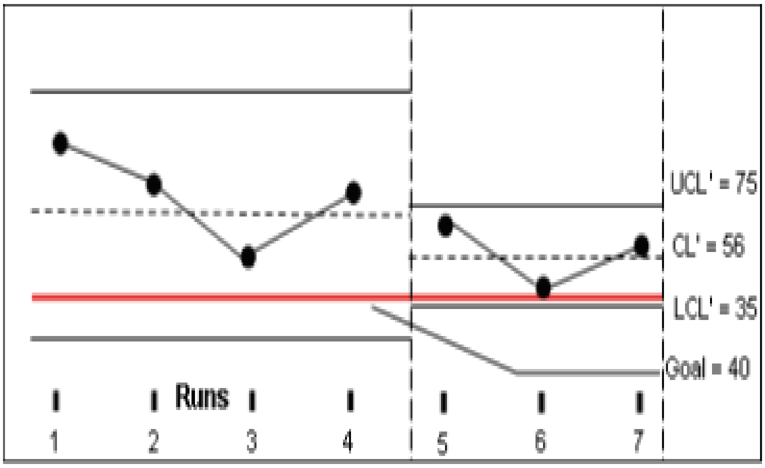


Fig. 13. New values after the application of solutions

Our test suit is not complete. Currently we are executing 60% of the total number of tests, already specified by our telecom test team. Furthermore, the evolution of new technologies, such as 3G, will certainly bring the need for new test cases. All these facts also contribute to a continuous and dynamic change of the indicator values, which can be monitored via the use of statistic methods such as the ones used in this work.

6. Test Management Tools

A last point to be discussed in this chapter is the use of test management tools as an alternative to complete some functions required by an automation test architecture. Current tools[3] can provide one or more from the following features:

- Specification for plans of tests - considers definition of test cases to be applied, test priority, test strategies, test schedule and resources to be allocated;
- Test evaluation support - considers specification and customization of test evaluation documents and summary of tests (test results, test cases coverage, process test environment);
- Extraction of statistical indicators - considers the capture of statistical indicators, specification and customization of statistical reports;
- Test cases creation - considers the edition and maintenance of test cases;
- Integration support - considers the existence of an API to enable the integration of a tool platform with external applications;

We can observe that several of these features are already considered by our proprietary solution. For example, the specification for test plans is covered by the test suit planner (Section 4). The main advantage in using external test management tools is the quality provided by several existing commercial systems. However, we must consider if such systems are flexible enough so that we can make adaptations and customizations in some of their functions.

7. Conclusion

The purpose of this chapter was to discuss several aspects related to the process of test automation. With this objective, we have introduced our domain and initial simulation test environment. Using this environment we have discussed several solutions and concepts that are currently under investigation by our research team. Furthermore, we have also discussed a statistical technique for controlling the process during its continuous evolution.

8. Acknowledgements

The authors would like to thank all the test engineers (Amanda Araujo, Karine Santos, Rivaldo Oliveira and Paulo Costa), software engineers (Angela Freitas and Kleber Carneiro), telecom internals (Anniele Costa and Ronaldo Bitu) and product manager (Fernando Buononato) of the CIn/SIDI-Samsung Test Center, which provided the technical details about the handsets' network test domain, simulation environment and GSM principles. The team is also very grateful for the support received from Samsung/SIDI team, in particular from Ariston Carvalho, Miguel Lizarraga, Ildeu Fantini and Vera Bier. The National Council for Scientific and Technological Development (CNPq) has provided valuable support to the project through the Brazilian Federal Law no. 8010/90.
UFPE and Samsung are authorized to reproduce and distribute reprints and on-line copies for their purposes notwithstanding any copyright annotation hereon. The views and

[3] TestLink (http://testlink.org), QATraq (http://www.testmanagement.com), RHT (http://sourceforge.net/projects/rth/), Salome-TMF (https://wiki.objectweb.org/salome-tmf) and so on.

conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of other parties.

8. References

Anite (1999). SAS 12447D/UMOOI GSM-9OO/DCSI800 /PCS-1900. *Stand Alone Simulator User Manual*, Rel. 3.0, Anite Telecoms Ltd, Fleet, Hampshire, UK.

Biehl, R. (2004). Six Sigma for Software, *IEEE Software*, 21, 2, 68-70, USA.

Deming, W. (1975). On probability as a basis for action, *The American Statistician*, 29, 4, 146-152.

De Vriendt, J. ; Laine, P. ; Lerouge, C. & Xiaofeng, X. (2002). Mobile network evolution: a revolution on the move, *IEEE Communications Magazine*, 40(4):104-111.

Fikes, R. & Nilsson, N. (1971). STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving, *Proceedings of Second International Joint Conference in Artificial Intelligence*, London, UK.

Filho, C & Ramalho, G. (2000). JEOPS – The Java Embedded Object Production System, *Springer Verlag's Lecture Notes in Artificial Intelligence*, 1952, 52-61, Heidelberg Germany.

Florac, W. & Carleton, A. (1999). Measuring the software process: statistical process control for software process improvement. *The SEI Series in Software Engineering*, Addison-Wesley.

Ganek, A. & Corbi, C. (2003). The Dawning of the autonomic computing era, *IBM Systems Journal*, 42, 1, 5-18.

Garg, V. (2001). *Wireless Network Evolution 2G to 3G*. Prentice Hall, 0-13028-077-1, USA.

Ghallab, M.; Nau, D. & Traverso, P. (2004). *Automated Planning: theory and practice*, Morgan Kaufmann Publishers, 1-55860-856-7, USA.

Gruber, R. (1995). Toward Principles for the Design of Ontologies Used for Knowledge Sharing. *International Journal Human-Computer Studies*, 43, 5/6, 907-928.

Guarino, N. (1995). Formal Ontology, Conceptual Analysis and Knowledge Representation, *International Journal of Human-Computer Studies*, 43, 5/6, 625–640.

Hariri, S. et al. (2006). The Autonomic Computing Paradigm, *Cluster Computing: The Journal of Networks, Software Tools, and Applications*, 9, 1, 5-17, Kluwer Academic Publishers.

Harry, M. (1998). Six Sigma: A Breakthrough Strategy for Profitability, *Quality Progress Publications*, 31, 5, 60-64.

Humphrey, W. (1988). Characterizing the Software Process: A Maturity Framework. *IEEE Software*, 5, 2, 73-79.

Kephart, J. & Chess, D. (2003). The Vision of Autonomic Computing, *IEEE Computer*, 36, 1, 41-50.

Langley, P. & Allen, J. (1993). A unified framework for planning and learning. In S. Minton (Ed.), *Machine learning methods for planning*. San Mateo, CA: Morgan Kaufmann.

Nguyen, X. & Kambhampati, S. (2001). Reviving partial order planning, *Proceedings of Seventeenth International Joint Conference in Artificial Intelligence*, 459-466, Seattle, WA, USA.

Russel, S. & Norvig, P. (2002). Artificial Intelligence: A Modern Approach, 2nd Edition, Prentice Hall, 0-13790-395-2, USA.

Rahnema, M. (1993). Overview of the GSM system and protocol architecture, *IEEE Communications Magazine*, 42, 4, 493-502.

Simon, K. (2007). DMAIC versus DMADV, *Six Sigma WebPage*. Available in: http://www.isixsigma.com/library/con tent/c001211a.asp

VanBrunt, R. (2003). WCDMA versus GSM: handset performance testing, *RF Design*, 26, 9, 14-23, Cardiff Publishing Company Inc, USA.

Weld, D. (1994). An introduction to least-commitment planning, *AI Magazine*, 15, 4, 27-61.

Wilkins, D. (1984). Domain-independent planning: representation and plan generation, *Artificial Intelligence*, 11, 3, 269- 301.