Chapter 3
Automatic Software Verification

The complexity of cell phones is continually increasing, with regards to both hardware and software parts. As many complex devices, their components are usually designed and verified separately by specialized teams of engineers and programmers. However, even if each isolated part is working flawlessly, it often happens that bugs in one software application arise due to the interaction with other modules. Those software misbehaviors become particularly critical when they affect the residual battery life, causing power dissipation. An automatic approach to detect power-affecting software defects is proposed. The approach is intended to be part of a qualifying verification plan and complete human expertise. Motorola, always at the forefront of researching innovations in the product development chain, experimented the approach on a mobile phone prototype during a partnership with Politecnico di Torino. Software errors unrevealed by all human-designed tests have been detected by the proposed framework, two out of three critical from the power consumption point of view, thus enabling Motorola to further improve its verification plans. Details of the tests and experimental results are reported.

3.1 Introduction

Verifying all the software running on a given apparatus is a complex problem, especially when the system under test is a mobile device, in which a software misbehavior can affect residual battery life. Traditional software verification techniques are often unable to work on a great number of applications at the same time, and since some software modules could be developed by third parties, verification engineers could not always have access to all data needed for the verification process. Evolutionary computation techniques proved able to tackle difficult problems with relevant degrees of success [43], even if some data of the problem is not completely known. Specialized literature routinely reports techniques that deliver high-return human-competitive machine intelligence simply starting from a high-level statement of what needs to be done and subsequently solving the problem without further need

E. Sanchez et al.: Industrial Applications of Evolutionary Algorithms, ISRL 34, pp. 17–30.
springerlink.com © Springer-Verlag Berlin Heidelberg 2012

of human intervention [86]. In the industrial world, however, the majority of existing processes employ no machine intelligence techniques, even if such approaches have been reported able to provide reliable results when facing complex problems.

The resistance in incorporating evolutionary computation in industrial processes may arise from the lack of experts with deep knowledge in both the machine intelligence and the industrial field. Automatic methodologies are perceived as scarcely controllable, and computational-intelligence techniques are regarded as "black magic", able to deliver impressive results sometimes, but definitely not reliable. In recent years, however, the interest of the industrial world towards automatic techniques has been steadily growing and some computational intelligence techniques have been successfully applied to some niche cases (e. g. credit card fraud detection performed by neural networks [19]).

An automatic approach based on an Evolutionary Algorithm (EA) is proposed, to add content to a human-designed verification plan for a mobile phone software system. The approach makes use of the EA to effectively [85] generate stimuli for a physical prototype of a cell phone, running simulations whose results are fed back to the EA and used to generate new stimuli. Data obtained from the simulations include physical measures and logs of all running applications. To explore effectively the solutions space, measures extracted from the prototype are integrated with data obtained from a model of the phone dynamically derived from simulation results.

Three different software misbehaviors, previously unrevealed by human-designed tests, are detected by the proposed approach. Incorporating this procedure in an existing set of tests allows Motorola [109] to further improve the effectiveness of qualifying verification plans. Preliminary results have been presented in [63] and [64].

3.2 Background

3.2.1 Mobile Phones

Since 1997, the mobile devices market has been steadily growing. Market researches projected that shipments of cell phones exceeded 1 billion units in 2009, so that mobile phones could become the most common consumer electronic device on the planet. Esteems from Gartner, shown in Fig. 3.1, predicted that there will be 2.6 billion mobile phones in use by the end of 2009 [161].

A great share of mobile devices sold nowadays is represented by the so-called smartphones, able to offer PC-like functionalities at the expense of an ever-growing complexity at both hardware and software level. Devices support more and more functions, running a great number of different applications: hardware miniaturization improves constantly, and thus battery life and power consumption related issues become more and more critical [36]. Thus, prediction of battery life [55] and

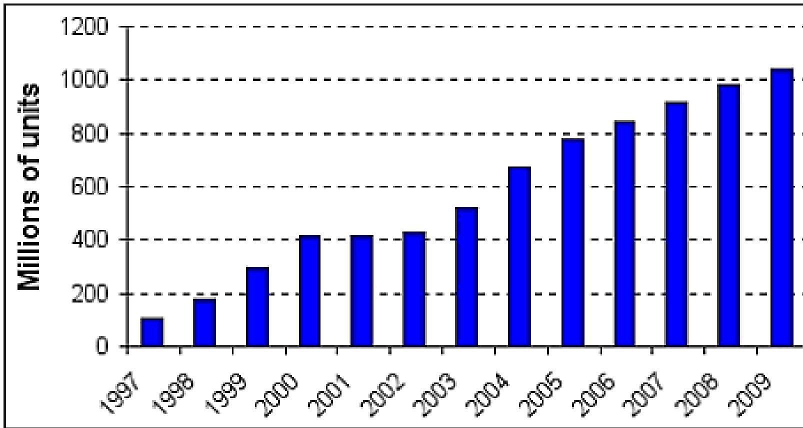

Fig. 3.1 Projection of cell phone sales by Gartner

improvement of energy supplies for mobile devices [90] [129] are research topics of great interest with significant contributions in literature.

Since the introduction of smartphones, the increasing number of applications run by mobile systems led to a great number of possible misbehaviors caused by software bugs. The most displeasing errors for the user are obviously those related to battery life, and in particular incorrect behaviors happening during the state where the cell phone consumes a minimal quantity of energy, called deep sleep. A mobile device enters deep sleep mode when it is left idle for a given amount of time or when a certain signal is given by the user (e. g. when the cap of a mobile phone is closed). Errors that arise in deep sleep can completely exhaust the battery of a cell phone while the user is oblivious to what is happening: a customer could find out that her mobile phone is discharged even if it was fully charged a few hours before.

3.2.2 Verification Techniques

Verification is the process that aims at guaranteeing the correctness of the design. Verification techniques exploit different paradigms, but, roughly speaking, it is possible to state that almost all can be classified either as formal or simulation-based. The former exploits mathematical methodologies to prove the correctness of the design with respect to a formal specification or property, while the latter is based on a simulation that aims at uncovering incorrect behaviors. Exploiting formal methods allows to verify the module with all possible inputs passing through all possible states. Therefore, these techniques in theory guarantee the highest levels of confidence in the correctness of the results, but when a formal method fails to prove a property, nothing can be determined about it, not even with a low amount of confidence. The human and computational effort required to apply formal verification

techniques, severely limit their applicability. Such methods, as a result, are applied in the industrial field only when facing few software or hardware modules, when validation task can be significantly constrained by boundary conditions or when oversimplified models are employed, thus significantly impairing the confidence of the results [50]. Systems composed of a great number of modules usually cannot be tackled by formal verification, due to the growth of complexity of these techniques. To maximize their efficiency, formal verification techniques are usually applied to the source code of the model description. However, in the mobile phone prototyping arena, the very first time a mobile phone prototype is implemented, some applications running on the phone are developed by third parties and their original code is often non accessible [3]. Therefore, it is not always feasible to exploit formal verification techniques during the verification plan of a mobile phone.

Simulation-based techniques rely on the generation of a set of stimuli able to thoroughly excite the device under verification: the stimuli set is simulated exploiting the considered module. Subsequently, all data obtained from the simulation is gathered and analyzed, aiming to unearth misbehaviors by comparison with the expected results. A simulation-based approach may be able to demonstrate the presence of a bug even in frameworks with a great number of applications or hardware modules running simultaneously, but will never be able to prove its absence. Indeed, verification engineers may assume that no bugs exist depending on the level of confidence related to the quality of the simulated test set. Stimuli sets can be applied to either a physical prototype or a simulable model of the device. Both approaches have advantages and disadvantages: while models often describe only some aspects of the system, they may allow verification engineers to control all details of the simulation and gather a large amount of information. On the other hand, a physical prototype may be more difficult to control, but results of the physical emulation are completely unbiased, and the computational time required to apply the stimuli set is lower compared to model simulation. Either using a model or a prototype, the generation of a qualifying set of stimuli is the key problem with simulation-based techniques.

As mentioned by Piziali in [122], the real success of a simulation-based verification process relies on the adequacy of the initial verification route-map, called functional verification plan. A verification plan must define important test cases targeting specific functions of the design, and it must also describe a specific set of stimuli to apply to the design model. The verification plan can also allocate specific tasks to specialized engineers.

One of the most important tasks of the verification plan is the generation of the stimuli that thoroughly exercise the device, obeying the directives defined in the route-map.

According to the defined plan, different methodologies may be used to properly generate verification stimuli sets, for example deterministic, pseudo-random, or constrained-random. The generation of stimuli can be driven by past experience of the verification engineers or by exploiting the extracted information of a given model of the system. The latter technique is called model-based testing, and for complex software systems it is still an actively evolving field [52].

A typical verification plan usually starts by tackling corner cases with hand-written tests. The verification stimuli set is then improved by adding information automatically generated exploiting simulation-based approaches. At last, the automatically generated test set requires an additional analysis by the verification engineers. Tests developed in such a way require a considerable amount of expertise related to the device under test, they are not always portable, and their preparation is time-consuming and expensive.

Completely automated approaches for stimuli generation can follow several methodologies: constrained-random generation, sometimes simply referred to as random or pseudo-random test generation, and feedback-based generation are the most widely adopted.

In a constrained-random test generation [69], random stimuli set are created by following a constrained generation process. Templates and constraints previously specified are exploited to define the structure of each stimuli fragment which is then randomized. When targeting real designs, such techniques have been proved to be really challenging, and are outperformed by feedback-based approaches [114].

Feedback-based approaches initially apply stimuli to the system, check the output produced and obtain information that is eventually exploited to produce new, and probably better, stimuli. This process is repeated, generating a set of stimuli able to stress the system very effectively: considerable proofs support the predominance of feedback-based techniques over other simulation-based ones [145]. Another important advantage of feedback-based approaches is that at the end of the process, a very compact set of data is produced: even though a large number of stimuli is simulated, most of the results are fed back to the system and exploited internally. Thus, verification engineers are required to analyze smaller quantities of information.

In a typical hand-written test for a new mobile phone, the phone is woken up from deep sleep mode, a sequence of key pressures is given in input to it, it is turned back to deep sleep and power consumption is eventually determined. Frequently, these sequences of keys mimic actions that will be likely performed on the phone, e. g. starting a video call, inserting a new field in the address book, etc. Once a number of similar devised tests are completed, a test set is created thanks to an automated approach that generates stimuli similarly structured to the hand-written ones.

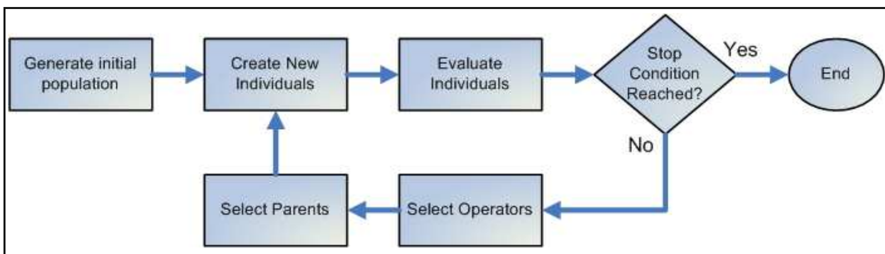

Fig. 3.2 Flowchart of a generic EA. During the evaluation step, individuals with lowest values of goodness are removed from the population.

Verification plans focused on simulation-based techniques are developed by industries to provide a set of stimuli able to excite completely the functionalities of the device under verification, consequently locating possible software bugs. When tackling the software of mobile phones, first of all verification engineers perform module-oriented verification procedures on single software application: this process is often developed separately for each component. In a second step, different applications are run at the same time, studying reciprocal influences among the modules and performing new verification tests on the whole system. In a third step, technical experts use the device, trying to locate weaknesses of the complete framework. During each step, verification engineers may rely on techniques available in literature on a single phase.

Among feedback-based techniques, Evolutionary Algorithms (EAs) are stochastic search techniques that mimic the metaphor of natural biological evolution to solve optimization problems [107]. Initially conceived at the end of 1960s, the term EAs now embraces genetic algorithms, evolutionary strategies, evolutionary programming, and genetic programming. Scientific literature reports several success stories in different domains, for instance [131].

Despite great differences, all EAs have many properties in common. EAs operate on a population of individuals; underlying each individual encodes a possible solution for the given problem. The goodness of every solution is expressed by a numeric value called fitness, usually obtained through an evaluator able to estimate how well the solution performs when applied to the problem. An evolutionary step, called generation, always consists of two phases: a stochastic one where some of the best individuals are chosen at random to generate new solutions; and a deterministic one, where solutions are ranked by their fitness and the worst ones are removed from the population. The process is then repeated until a user-defined stop condition is met. Fig. 3.2 shows a classical flow for an EA. When facing verification problems, stimuli created by an EA explore the solution space very efficiently. Moreover, the solutions found by EAs are somewhat very different from, and thus complementary to, human-made solutions [97].

3.3 Proposed Approach

The objective of the proposed approach is to find a set of stimuli able to detect errors triggered by the interaction of software applications on a mobile phone by stressing the functionalities of all the modules as much as possible. The approach is feedback-based, driven by an EA that evolves a population of candidate stimuli, coded as sequences of key pressures and pauses, similar to hand-written tests devised by expert engineers. The approach is also model-based: a finite-state machine (FSM) representing the system under verification is exploited to extract measures for the goodness of each solution. The FSM is automatically generated from scratch thanks to the data obtained by running simulations with the stimuli as an input to a physical prototype of the phone itself. The model supplies information on the

number of different applications' features activated by each stimulus; this data is later used to assign a value to the stimulus, expressing its goodness. Fig. 3.3 shows a schema of the proposed framework: the EA manages a population of individuals that map stimuli. Such stimuli are evaluated by the model dynamically extracted from the physical device.

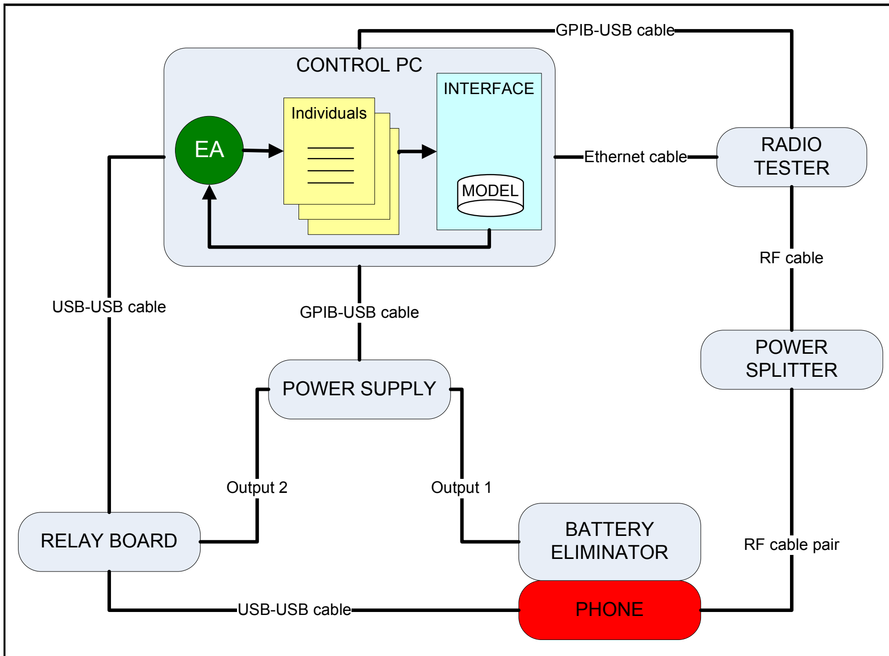

Fig. 3.3 Schema of the proposed framework

3.3.1 Model

The device under verification is modeled with a FSM, where each state defines a situation in which all active software modules are waiting for new inputs. A transition is a series of inputs that connect a state to another, turning on/arresting different applications or exciting some functionalities of the active ones. The FSM is exploited to evaluate the number of distinct states traversed and the transitions activated during the simulation of a stimulus [127].

Creating a complete model of all the software running on the mobile phone with the classical methodologies of software engineering would be impractical, requiring an excessive amount of time: the source code of each software module on the device should be provided and analyzed. Since some applications are developed by

third parties, not all the software modules' source code is obtainable, thus critical data to build a complete model is missing. On the contrary, the FSM in the proposed framework is created as the simulations go on, and each time a new state is discovered the model is dynamically updated. Since this approach does not rely on a-priori knowledge, errors that could occur in the model-building phase are avoided.

The Operating System (OS) and most applications on mobile phones can run in a test mode where they write a log of their execution to ease the debugging process. By reading system messages recording applications starting and closing, called events in the following, it is possible to create a list of states. Each state is identified by a status word, obtained by parsing the debug logs. Every time an event is raised or a new feature of an active application is activated, the debug log register the changes. When all applications active on the phone are waiting for new input, the status word is collected by parsing the logs.

Starting with an empty FSM, new states and transitions are added each time a new status word is discovered. Old status words are stored, thus the framework can add transitions returning to states already known. Since the proposed framework makes mainly use of the number of different transitions fired, it does not require the supporting model to be complete or perfect.

μGP [149] [134], a general-purpose tool developed by the CAD Group of Politecnico di Torino, is the EA chosen to be included in the framework. μGP is available as a GPL tool [146]. Candidate solutions of a problem in μGP are represented as graphs, while the problem itself is encoded as an external program or script that evaluates each candidate solution and supplies the tool with a measure of its goodness. Since the evolutionary core is loosely coupled with the evaluation, μGP can be used in a wide range of different problems with no modifications needed.

While the tool was originally exploited to generate Turing-complete programs in assembly language, over the years μGP handled different problems whose solutions had complex structures.

Genetic operators, such as classical mutation and cross-over, modify the graph that encode the individuals. The tool architecture is designed to handle a large number of genetic operators, to ease the addition of new ones and to let the user choose the operators to apply to the problem. Each operator is associated with an activation probability, that is managed internally by μGP, and an endogenous parameter called *strength* that defines the differences between the parents chosen and the offspring generated.

In μGP version 3, individuals are represented as constrained tagged graphs, i. e. graphs with added information to nodes and edges, while the possible structures are limited by the user. Thanks to the constrained graphs, the tool can handle problems where the solution has structures simpler than Turing-complete assembly problems, like linear graphs, linear genomes or fixed-length bit strings.

The fitness of each candidate solution is computed by a script or program that runs a simulation using the individual as input and feeds back the results to μGP. The fitness in the tool is described by a vector of floating point numbers followed optionally by a comment. Each position of the vector is considered more important that the following: fitness A is greater than fitness B if the number in the nth position

of vector A is greater than number in the nth position of vector B, and all the number in previous positions (if any) are equal; if all components are equal then the two fitness are considered equal.

The proposed framework makes use of μGP in its basic version, with no changes or additions to the original code. Configuration files in eXtensible Markup Language (XML) describe individuals' structure and all necessary parameters such as population size, stop conditions, number of genetic operators activated at each step. Since in the specific problem individuals map sequences of keys, the related graphs are linear genomes.

In the architecture of μGP, the evaluator is completely separated from the evolutionary core, so the evaluation program is designed from scratch and it is specific for each problem.

3.3.2 Candidate Solutions

Stimuli candidates to solve the problem are handled as a population of individuals by the EA. Each individual is a small program in Java that encodes sequences of keys and pauses: the programming language is chosen for the ease of compatibility with the OS running on the cell phone. The first part of each individual inscribes procedures of device initialization, while the last part makes the phone revert to an idle state to subsequently trigger deep sleep mode.

The initial population provided to the EA contains both individuals encoding random sequences of keys and pauses, and individuals encoding the most common actions performed by a user on the mobile device, e. g. selecting a number in the address book, making a video call, etc. Making the EA discover autonomously those sequences is possible, but it would take a great amount of time. Since human-devised tests have been already run on the prototype when the proposed methodology is used, starting the evolution from scratch is redundant. Individuals encoding common actions are derived from human-written tests cases used in other verification steps with an ad-hoc tool.

The EA manipulates and reassembles user-defined sequences and random individuals, mixing and modifying them to create new individuals with the aim to maximize the goodness of an individual. For example, a sequence derived from human-designed tests in the initial population may be later mixed with a different sequence and mutated by adding, removing or changing random lines of code.

3.3.3 Evaluator

Detecting a software bug that affects negatively battery life is the final goal of the evolution. Unlike other problems, where the goal leads straightforwardly to the definition of a continuous evaluation function, the presence of a bug cannot be expressed

with such a function. A software error can be either detected or silent, with no other values. As natural evolution, EA "can act only by the preservation and accumulation of infinitesimally small inherited modifications, each profitable to the preserved being" [47]. The evaluation function of the specific problem needs consequently to be refined using heuristic methods.

The measure of the power consumption in deep sleep mode is surely included in the goodness value of each individual, because of the goal of the experience, but since most individuals use the same amount of energy, it is not enough to smoothen the landscape of the evaluation function. Parameters that lead to a quicker location of bugs must be taken into account as well.

The more an individual activates different software applications or different functionalities of the same application, the greater the probability that it will trigger a bug: consequently, individuals which excite more phone applications should be rewarded with a higher value when evaluating their goodness.

Three contributions (Pi, Ti, Ei) are taken into account for the global goodness value of individual i:

1. The mean value of power consumption while the cell phone prototype is in deep sleep mode, measured over 30 s and defined as
$$Pi = \frac{\sum_{t=0s}^{30s} P(t)}{30}$$
where $P(t)$ is the power consumption at time t;
2. The number of *transitions* covered in the FSM that models all the software applications running on the phone, as described in 3.3.1, defined as
$$Ti = \sum_{tr=0}^{TR} 1$$
where TR is the total number of transitions fired. A transition is defined as a passage from one state to another;
3. The number of different events activated, defined as
$$Ei = \sum_{e=0}^{E} 1$$
where E is the number of events raised from different applications.

The structure composed of these three contributes aims at discovering as many states as possible in the FSM built dynamically and at activating the maximum possible number of transitions. As in the initial idea, a high value is associated to solutions that excite a great number of different software modules on the phone, and have an extreme power drain in deep sleep mode.

As the measure of the goodness of the solutions is conceived, rewarding candidate solutions that activate more software applications could lead to the exclusion of an individual that targets only one module: that module, however, could be ignored

by the rest of the population. In a similar way, the population in the long run could be filled with individuals very much alike. μGP, the EA chosen for the experience, has features enforcing diversity preservation in a population which help to avoid both those risks [43].

3.4 Experimental Results

The proposed framework was tested on a cell phone prototype running a Motorola P2K OS. The phone had been analyzed by verification engineers and passed all human-designed test. Thus, there were no known misbehaviors in the phone software modules when the proposed approach was applied to the device. The features of each component involved in the experience are summarized in Table 3.1. The experiments made use of the phone prototype, a radio tester, a power supply and a computer to control the instruments.

To measure the power consumption in deep sleep mode and to keep the phone powered during the experiments, a battery eliminator was connected to the phone. The battery eliminator is made of a battery whose contacts are disconnected from the inner cells and attached to a power supply. A relay board managed the connection between the PC and the device under verification: the phone does not enter deep sleep mode as long as it is connected to a PC. The relay board switched the phone from a state where it is in use to a state where it is no longer in use and can thus enter deep sleep.

To simulate a mobile network providing voice, video and data packet services, a radio analyzer was linked to the phone, thus producing an environment completely under the user's control.

By means of an ad-hoc tool, human-devised tests written in Java were converted to an XML representation later used as part of the initial population used by μGP. Such Java programs described every possible command that a user could issue to the mobile device, and each test was converted by μGP into a sequence of macro instances. The tool is problem-specific and it was developed in Motorola Research Laboratories located in Torino, Italy.

Since not all the applications stopped their execution when the mobile phone's flip was closed, the cell phone was sometimes blocked from entering deep sleep. For example, the software manager for the photo camera shows what the camera is shooting on the external monitor even if the flip is closed. Constraints were consequently modified to solve the issue. During the experience, every test was performed with a population of 50 individuals and an average of 40 new individuals generated at each generation. Each test took up to 100 generations of evolution.

With these parameters, it is clear that a great number of evaluations are performed during each experiment, so one of the first goals of the experience was to shorten the temporal length of a single evaluation as much as possible. Instead of setting the parameters of the phone by browsing through the menus, a time-expensive activity, by using seem elements, similar to configuration bits for the P2K OS, the duration of a

test was reduced by 30 seconds. A time-out coded in the Java class that manages the logs of the phone was removed to further improved the performance by 20 seconds more. Nevertheless, some of the most dilatory steps could not be shortened: it took the phone roughly 60 seconds to enter deep sleep mode after the flip was closed, and the master clear needed to return the phone to its initial state after each test took 35 seconds.

Even with the improvements achieved, the average evaluation time of a candidate solution during the experiments was between 6 and 7 minutes, and thus it took about 5 hours to complete a single generation step. A significant saving of time was obtained thanks to the features of μGP, that keep the number of evaluations to a minimum.

To avoid the generation of multiple individuals triggering the same software errors, μGP constraints were altered after each discovery of a bug.

All the experiments had been completed in the Motorola Research Labs in Turin.

3.4.1 Video Recording Bug

During the first evolution, the majority of individuals showed a deep sleep power consumption between 2,5 and 3,2 mA. It took 16 hours of computation and 150 candidate solutions evaluated to find three individuals with a deep sleep consumption of about 7,0 mA which is 2,5 times the normal value.

The shortest of the three was composed by 100 lines of Java code, defining pressures of keys and pauses. It was analyzed through a series of runs on the framework already developed for the experiments, and the cause of the bug was uncovered: the pressure of specific buttons while the phone was in video recording mode caused a warning dialog to pop up on the display and froze the OS completely, thus keeping the phone from entering deep sleep. The error was caused by the interaction of the software controlling the video recording and the software managing the address book.

Further analyses on the two other candidate solutions uncovered the same subsequence of keys that caused the error in the first one.

3.4.2 Voice Call Bug

It took additional 120 hours of computation and the evaluation of about 1120 individuals to find out a second power-related bug. The best candidate solution found during this experience let the phone enter deep sleep, but the power measurements revealed a consumption of 50 mA, more than 16 times the expected use of power in deep sleep mode.

The mobile phone did not show messages or exhibit unexpected behaviors: it was necessary to analyze the individual's code line by line, but the cause of the error was

eventually located. If the video call button was pressed along with a special series of keys during a voice call, the phone camera was powered up and it kept being active even when the device returned to deep sleep mode, thus consuming an extreme quantity of power. This error was triggered by an interaction between the software controlling the camera and the software controlling the voice call.

3.4.3 Incorrect Menu Behavior

The experiments uncovered a third misbehavior, not affecting battery life but not previously located by human-devised tests. By entering a specific settings menu and exiting hereupon without making modifications, the device reset some of its settings to their initial values. A candidate solution with this pattern made the following tests fail. The end user would probably not affected greatly by this misbehavior, but the problem had to be taken into account during the experience.

Table 3.1 Hardware involved in the experience

Device	Details	Description
Motorola mobile phone prototype	P2K Platform	Runs the tests
Personal computer	OS: Microsoft Windows XP, Primary Memory: 512 MB, Ports: USB 2.0	Controls the power supply, the radio tester and the phone; provides packet data services to the radio tester
Power supply	Double-output with measurement capabilities; VISA interface and SCPI protocol support	Controls the relay board; performs current drain measurement of the phone during tests
Anritsu MT8802A Radio Communication Analyzer	Model with the proper installed options (see below)	Simulates the cellular network providing voice, video and data packet services
Two VISA bus cables	Any supported VISA interface	Connect the control PC to the power supply and the radio tester
Power splitter	500 - 5000 MHz	Joins the signals of the two antennas of the phone (required only to test simultaneously GSM and 3G)
RF cable	N type connector (to the radio tester)	Connects the radio analyzer to the power splitter
Two short RF cables	QMA type connector (to the phone)	Connect the two antennas of the phone to the power splitter (actually only one cable is required if testing only WCDMA or GSM standard)
Battery eliminator	Built in house	Bypasses the phone battery, feeding the phone with the power provided by the power supply
USB relay board	Allows computer controlled switching	Switches on/off the USB connection between the phone and the PC

3.5 Conclusions

A framework for automated verification is proposed to attest the correct behavior of a cell phone, uncovering software defects not detected by human-devised tests, searching specifically for bugs affecting power consumption. The approach makes use of feedback from the device under verification to produce new stimuli, with an EA providing the necessary intelligence. The quantity of final data is limited to a small significant amount.

The effectiveness of the system is demonstrated on a mobile phone prototype implementing the P2K OS. The framework successfully locates software misbehaviors previously undetected by standard human-supervised verification. Two of the bugs uncovered are critical with regards to power consumption in deep sleep mode, thus making them high-priority from the user's point of view.

The research team at Motorola Research Laboratories in Torino finds a way to further improve the qualifying verification plan for mobile devices.