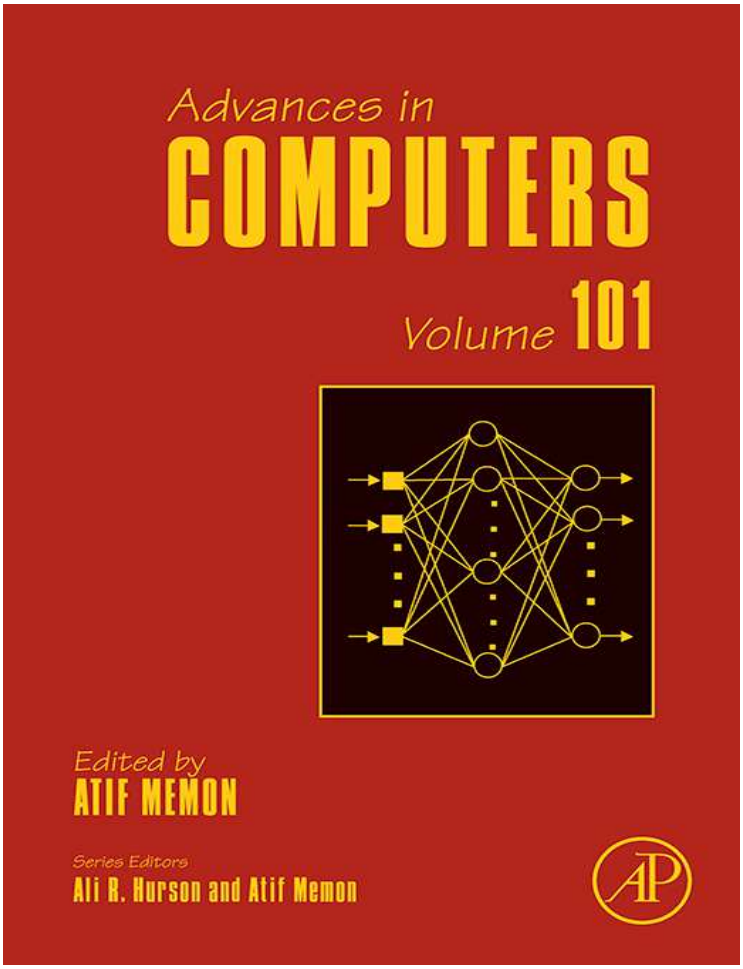

From Abhijeet Banerjee, Sudipta Chattopadhyay and Abhik Roychoudhury, On Testing Embedded Software. In: Atif Memon, editor, *Advances in Computers, Vol. 101*, Burlington: Academic Press, 2016, pp. 121-153.
ISBN: 978-0-12-805158-0

Academic Press

CHAPTER THREE

On Testing Embedded Software

Abhijeet Banerjee*, Sudipta Chattopadhyay†, Abhik Roychoudhury*
*National University of Singapore, Singapore
†Saarland University, Saarbrücken, Germany

Contents

Abstract

For the last few decades, embedded systems have expanded their reach into major aspects of human lives. Starting from small handheld devices (such as smartphones) to advanced automotive systems (such as anti-lock braking systems), usage of embedded systems has increased at a dramatic pace. Embedded software are specialized software that are intended to operate on embedded devices. In this chapter, we shall describe the unique challenges associated with testing embedded software. In particular, embedded software are *required* to satisfy several non-functional constraints, in addition to functionality-related constraints. Such non-functional constraints may include (but not limited to), timing/energy-consumption related constrains or reliability requirements, etc. Additionally, embedded systems are often required to operate in

Advances in Computers, Volume 101
ISSN 0065-2458
http://dx.doi.org/10.1016/bs.adcom.2015.11.005

interaction with the physical environment, obtaining their inputs from environmental factors (such as temperature or air pressure). The need to interact with a dynamic, often non-deterministic physical environment, further increases the challenges associated with testing, and validation of embedded software. In the past, testing and validation methodologies have been studied extensively. This chapter, however, explores the advances in software testing methodologies, specifically in the context of embedded software. This chapter introduces the reader to key challenges in testing non-functional properties of software by means of realistic examples. It also presents an easy-to-follow, classification of existing research work on this topic. Finally, the chapter is concluded with a review of promising future directions in the area of embedded software testing.

1. INTRODUCTION

Over the last few decades, research in software testing has made significant progress. The complexity of software has also increased at a dramatic pace. As a result, we have new challenges involved in validating complex, real-world software. In particular, we are specifically interested in testing and validation of embedded software. In this modern world, embedded systems play a major role in human lives. Such software can be found ubiquitously, in electronic systems such as consumer electronics (eg, smartphones, mp3 players, and digital cameras) and household appliances (eg, washing machines and microwave ovens) to automotive (eg, electric cars and antilock braking systems) and avionic applications. Software designed for embedded systems have unique features and constraints that make its validation a challenging process. For instance, unlike Desktop applications, the behavior of an embedded systems often depends on the physical environment it operates in. As a matter of fact, many embedded systems often take their inputs from the surrounding physical environment. This, however, poses unique challenges to testing of such systems because the physical environment may be non-deterministic and difficult to recreate during the testing process. Additionally, most embedded systems are required to satisfy several non-functional constraint such as timing, energy consumption, reliability, to name a few. Failure to meet such constraints can result in varying consequences depending upon the application domain. For instance, if the nature of constraints on the software are hard real time, violation may lead to serious consequences, such as damage to human life and property. Therefore, it is of utmost importance that such systems be tested thoroughly before being put to use. In the proceeding sections, we shall discuss some of the techniques proposed by the software engineering community that are

targeted at testing and validation of real life, embedded systems from various application domains and complexities. However, first we shall present an example, inspired from a real life embedded system, that will give the reader an idea on the nature of constraints commonly associated with embedded systems.

Fig. 1 provides the schematic representation of a *wearable fall detection application* [1]. Such an application is used largely in the health care domain to assist the frail or elderly patients. The purpose of the system, as shown in Fig. 1, is to detect a *potential fall* of its wearer and to invoke appropriate safety measures. In order to detect a fall, the system needs to monitor the user's movement. This task is accomplished via a number of sensors, that are positioned at different parts of the patient's body. These sensors detect physical motions and communicate the information via wireless sensor networks. In the scenario when the system detects a potential fall it activates appropriate safety measures, such as informing the health care providers over mobile networks. Testing the fall-detection system is essential to ensure its functional correctness, such as *a potential fall must not go undetected*. However, such a testing requires the inputs from the sensors. To properly test the system, its designers should be able to systematically model the inputs from sensors and the surrounding environment.

Apart from the functional correctness, the fall-detection system also needs to satisfy several non-functional constraints. For instance, the detection of a fall should meet hard timing constraints. In the absence of such constraints, the respective patient might get seriously injured, making the system impractical to use. Moreover, if the application is deployed into a battery operated device, its energy consumption should be acceptable to ensure a graceful degradation of battery life. Finally, due to the presence of unreliable hardware components (eg, *sensors*) and networks (eg, sensor and mobile

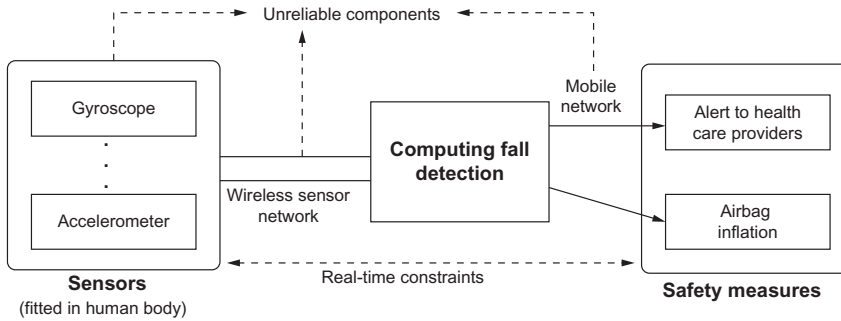

Figure 1 A wearable fall-detection application.

networks), the application should also guarantee that a potential fall of the patient is detected with acceptable reliability.

Non-functional properties of embedded software, such as timing and energy, are extremely sensitive to the underlying execution platform. This makes the testing process complicated, as the underlying execution platform may not be available during the time of testing. Besides, if the embedded software is targeted at multiple execution platforms, its non-functional properties need to be validated for each such platform. To alleviate these issues, a configurable model for the execution platform might be used during the testing process. For instance, such a configurable model can capture the timing or energy behavior of different hardware components. Building such configurable models, however, may turn out challenging due to the complexity of hardware and its (vendor-specific) intellectual properties.

Over the last two decades, numerous methods in software testing have been proposed. These include random testing, search-based testing, and directed testing (eg, based on symbolic execution), among several others. These testing methodologies have focused primarily on the validation of functional properties. Validation of non-functional software properties, have gained attention only recently. In this Chapter, we explore the potential of different testing methodologies in the context of embedded software. For an embedded software, its non-functional aspects play a crucial role in the validation process. We introduce some salient properties of validating typical embedded systems in Section 2. Subsequently, we shall explore the recent advances in testing embedded systems in Section 3. We first categorize all testing methodologies into three broader categories. Such categories reflect the level of abstraction, in which embedded systems are validated. In particular, our first category captures black-box testing, where the system is abstracted away and test inputs are generated via sampling of the input space. The remaining categories either use an abstract model of the system or the actual implementation. We shall discuss that different testing machineries (eg, evolutionary testing and symbolic execution) can be employed for such categories. Based on our categorization of testing embedded systems, we shall argue that no single category can be decided to be superior than others. In general, the choice of abstraction, for testing embedded system, largely depends on the intention of the designer. For instance, if the designer is interested in detecting fine-grained events (eg, memory requests and interrupts), it is recommended to carry out the testing process on the actual implementation (eg, binary code). On the contrary, testing binary code may reveal non-functional bugs too late in the design process, leading to a complete redesign of the software.

Through this chapter, we aim to bring the attention of software engineering community towards the unique challenges involved in embedded software testing. Specifically, testing of non-functional properties is an integral part of validating embedded software. In order to validate non-functional properties, software testing methodologies should explicitly target to discover non-functional bugs, such as the loss of performance and energy. Moreover, in order to test functional properties of embedded software, the designer should be able to simulate the interaction of software with the physical environment. We shall discuss several efforts in recent years to discover functional as well as non-functional bugs in embedded software. In spite of these efforts, numerous challenges still exist in validating embedded software. For instance, non-functional behaviors of embedded software (eg, time and power) can be exploited to discover secret inputs (eg, secret keys in cryptographic algorithms). Testing of timing and energy-related properties is far from being solved, not to mention the immaturity of the research field to validate security constraints in embedded software. We hope this chapter will provide the necessary background to solve these existing challenges in software testing.

2. TESTING EMBEDDED SOFTWARE

Analogous to most software systems, testing embedded software is an integral part of the software development life cycle. To ensure the robustness of embedded software, both its functional and non-functional properties need to be examined. In the following discussion, we outline some salient features that make the testing of embedded systems unique and challenging, compared to traditional software systems.

2.1 Testing Functional Properties

The functionality of software systems capture the way such systems should behave. Therefore, testing functional properties is a critical phase for all applications. Typically, the functionality testing of software aims to discover "buggy" scenarios. For instance, such buggy scenarios may capture the violation of software behavior with respect to the specification or an implementation bug (eg, null pointer dereference and assertion failure). To discover and investigate a buggy scenario, the designer must be provided with appropriate test inputs that trigger the respective bug. Therefore, software testing tools should have a clear domain knowledge of the relevant inputs to the system. For embedded software, the functionality is often (partially)

controlled by the physical environment. Such physical environment might include air pressure, temperature, physical movement, among others. Unfortunately, the physical environment, where an embedded software is eventually deployed, is often not present during the testing time. For instance, consider the fall-detection application, which was introduced in the preceding section. It is crucial that the designed software invokes appropriate actions according to the movement of the patient. In the actual working environment, such movements are sampled from sensor inputs. Consider the code fragment in Fig. 2, which reads an accelerometer and takes action accordingly. The function `f(buffer)` captures a predicate on the values read into the buffer. The `else` branch of the code fragment exhibits a *division-by-zero* error when `buffer[0] = 0`. In order to execute the `else` branch, the test input must, additionally, satisfy the condition `f(buffer) = 0`. As the value of `buffer` depends on the physical environment, the inputs from the accelerometer might often need to be simulated via suitable abstractions. Similarly, for embedded software, whose functionality might depend on air pressure or temperature, the testing process should ensure that the respective software acts appropriately in different environmental conditions. In general, to simulate the physical environment, the designer may potentially take the following approaches:

- The physical environment (eg, inputs read from sensors) might be made completely *unconstrained* during the time of testing. This enables the testing of software under all operating conditions of the physical environment. However, such an approach might turn infeasible for complex embedded software. Besides, unconstraining the physical environment might lead to unnecessary testing for irrelevant inputs. Such inputs may include sensor readings (such as −300 K for air temperature readings) that may never appear in the environment where the software is deployed.

```
int x, y, buffer[128];

buffer = read_accelerometer(); //read accelerometer

if (f(buffer))
  Code A; //non-buggy code fragment
else
  y = x/buffer[0]; //buggy code fragment
```

Figure 2 The dependency of functionality on the physical environment.

- The physical environment might be simulated by randomly generating synthetic inputs (eg, generating random temperatures readings). However, such an approach may fail to generate relevant inputs. However, like traditional software testing, search-based techniques might improve the simulation of physical environment via evolutionary methods and metaheuristics.
- With a clear knowledge of the embedded software, the testing process can be improved. For instance, in the fall-detection system, it is probably not crucial to simulate the movement for all possible movement angles. It is, however, important to test the application for some inputs that indicate a fall of the patient (hence, indicating safety) and also for some inputs that does not capture a fall (hence, indicating the absence of false positives). In general, building such abstractions on the input space is challenging and it also requires a substantial domain knowledge of the input space.

We shall now discuss some non-functional properties that most embedded software are required to satisfy.

2.2 Testing Non-functional Properties

In general, most embedded software are constrained via several non-functional requirements. In the following and for the rest of the chapter, we shall primarily concentrate on three crucial properties of embedded software—timing, energy, and reliability.

2.2.1 Timing Constraints

Timing constraints capture the criteria to complete tasks within some time budgets. The violation of such constraints may lead to a complete failure of the respective software. This, in turn, may have serious consequences. For instance, consider the fall-detection application. The computation of a potential *fall* should have real-time constraints. More precisely, the time-frame between the sampling of sensor inputs and triggering an alarming situation should have strict timing constraints. Violation of such constraints may lead to the possibility of detecting a fall *too late*, hence, making the respective software impractical. Therefore, it is crucial that the validation process explicitly targets to discover the violation of timing-related constraints. It is, however, challenging to determine the timing behavior of an application, as the timing critically depends on the execution platform. The execution platform, in turn, may not be available during the testing phase. As a result, the validation of timing-related constraints, may often

involve building a timing model of the underlying execution platform. Such a timing model should be able to estimate the time taken by each executed instruction. In general, building such timing models is challenging. This is because, the time taken by each instruction depends on the specific instruction set architecture (ISA) of the processor, as well as the state of different hardware components (eg, cache, pipeline, and interconnect). To show the interplay between the ISA and hardware components, let us consider the program fragment shown in Fig. 3.

In Fig. 3, the `true` leg of the conditional executes an `add` instruction and the `false` leg of the branch executes a `multiply` instruction. Let us assume that we want to check whether this code finishes within some given time budget. In other words, we wish to find out if the execution time of branch with the longer execution time is less than the given time budget. In a typical processor, a multiplication operation generally takes longer than an addition operation. However, if the processor employs a cache between the CPU and the memory, the variable `z` will be cached after executing the statement `z := 3`. Therefore, the statement `x := x*z` can be completed without accessing the memory, but the processor may need to access the memory to execute `x := x+y` (to fetch `y` for the first time). As a result, even though multiplication is a costly operation compared to addition, in this particular scenario, the multiplication may lead to a faster completion time. This example illustrates that a timing model for an execution platform should carefully consider such interaction between different hardware components.

Once a timing model is built for the execution platform, the respective software can be tested against the given timing-related constraints. Broadly, the validation of timing constraints may involve the following procedures:

- The testing procedure may aim to discover the violation of constraints. For instance, let us assume that for a fall-detection application to be

```
int x, y, z;

z := 3; //z is accessed and put into the cache

if (x > 0)
  x := x + y; //y needs to be fetched from the cache
else
  x := x * z; //all variables are cached
```

Figure 3 The timing interplay between hardware components (eg, caches) and instructions.

practical, the alarming situation must be notified within 1 ms (*cf.* Fig. 1). Such a constraint can be encoded via the assertion: *assert*(*time* <= 1*ms*), where *time* is the time taken by the fall-detection application to compute a potential fall. The value of *time* can be obtained by executing the application directly on the targeted platform (when available) or by using a timing model for the same. The testing process aims to find test inputs that may potentially invalidate the encoded assertions.

- It may, however, turn difficult for a designer to develop suitable assertions that capture timing constraints. In such cases, she might be interested to know the worst-case execution time (WCET) of the software. As the name suggests, WCET captures the maximum execution time of an application with respect to all inputs. Accurately determining the WCET of an application is extremely challenging, especially due to the complex interactions across different software layers (application, operating systems, and hardware) and due to the absence of (proprietary) architectural details of the underlying execution platform. However, WCET of an application can be approximated via systematically testing the software with appropriate inputs. For instance, we shall discuss in Section 3 about the progress in evolutionary testing to discover the WCET.

2.2.2 Energy Constraints

Like timing, energy consumption of embedded software may also need careful consideration. In particular, if the respective software is targeted for a battery-operated device, the energy consumption of the software may pose a serious bottleneck. For instance, if a fall-detection software is battery-operated, the power drained from the battery should be acceptable in a way to trigger the alarming situation. Like timing, the energy consumption of software is also highly sensitive to the underlying execution platform. Therefore, in the absence of the execution platform, an appropriate energy-model needs to be developed. Such an energy model can be used during the test time to estimate the energy consumption of software and to check whether the software satisfies certain energy constraints. Similar to timing constraints, energy constraints can be captured systematically via assertions or via computing the worst case energy consumption (WCEC) of the respective software. The computation of WCEC has similar challenges as the computation of the WCET and therefore, such computations might involve approximations via systematically generating test inputs.

2.2.3 Reliability Constraints

As embedded software often interacts with the physical environment, it needs to reliably capture the data acquired from the physical world. Usually, this is accomplished via sensors (eg, gyroscope and accelerometers), which interacts with the software via communicating the data from the physical world. For instance, in the fall detection application, the data read via the sensors are sent via wireless sensor network. In general, it is potentially infeasible to get the sensor data accurately. This might be due to the inaccuracy of sensor chips or due to potential packet drops in the network. Therefore, the reliability of different software components may pose a concern for a critical embedded software, such as a fall detector. Besides, the reliability of a component and its cost has nontrivial trade-offs. For instance, a more accurate sensor (or a reliable network) might incur higher cost. Overall, the designer must ensure that the respective software operates with an acceptable level of reliability. As an example, in the fall detector, the designer would like to ensure that a physical fall is alarmed with $x\%$ reliability. Computing the reliability of an entire system might become challenging when the system consists of several components and such components might interact with each other (and the physical world) in a fairly complex fashion.

To summarize, apart from the functionality, most embedded software have several non-functional aspects to be considered in the testing process. Such non-functional aspects include timing, energy, and reliability, among others. In general, the non-functional aspects of embedded software may lead to several complex trade-offs. For instance, an increased rate of sampling sensor inputs (which capture the data from the physical world) may increase energy consumption; however, it might increase the reliability of the software in terms of monitoring the physical environment. Similarly, a naive implementation to improve the functionality may substantially increase the energy consumption or it may lead to the loss of performance. As a result, embedded software are required to be systematically tested with respect to their non-functional aspects. In the next section, we shall discuss several testing methodologies for embedded software, with a specific focus on their non-functional properties.

3. CATEGORIZATION OF TESTING METHODOLOGIES

Real-time and embedded systems are used extensively in a wide variety of applications, ranging from automotive and avionics to entertainment and consumer electronics. Depending on the application, the constraints applicable on such systems may range from mission-critical to soft-real time

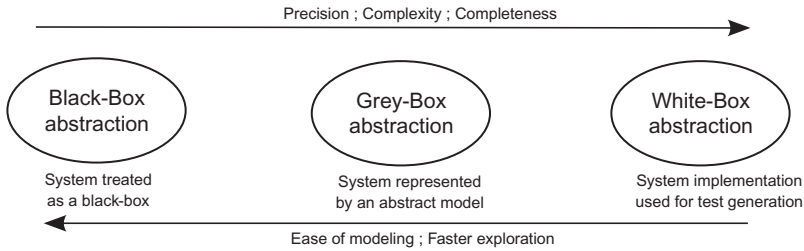

Figure 4 Classification of existing approaches for embedded software testing.

in nature. Additionally, embedded systems often have to interact with the physical environment that may be deterministic or non-deterministic. Such factors imply that embedded systems have to be designed and developed with varying operational requirements and no single testing technique is well suited to all systems. In some scenarios, the system under test (SUT) may be too complex to model and hence, approximate, yet fast sampling-based techniques are suitable. In other scenarios, where the SUT has mission-critical constraints and requires thorough testing, a fine-grained modeling of the system is crucial. In the following paragraphs, we shall categorize and discuss some of the existing works on testing embedded systems, with a specific focus on works being published in the past 5 years. In particular, we categorize all works into following three divisions (as shown in Figure 4):

Black-Box Abstraction : Such techniques often consider the SUT as a black-box. Test cases are generated by sampling, randomized testing techniques.

Grey-Box Abstraction : Such techniques do not treat the SUT as a black-box. The SUT is represented by a model, which captures only the information related to the property of interest. Test cases are generated by exploring the search space of the model.

White-Box Abstraction : Techniques in this category often require the source code or binary of the implemented system for the testing process. In other words, the source code and binary serves as the model of the system. Test cases are generated by searching the input space of the implemented system.

In subsequent sections, we shall elaborate on each of the categorization as described in the preceding paragraphs.

4. BLACK-BOX ABSTRACTION

One of the most simple (but not necessarily effective) approaches of testing complex systems is to uniformly sample its input space. The goal

of such sampling is to generate test inputs. As exceedingly simple as such a method might seem, the effectiveness of such uniform (or unguided) sampling remains questionable. When testing a system, in general, the objective is to produce test inputs that bears witnesses to failure of the system. Such a failure might capture the violation of a property of interest. Besides, such violations should be manifested within a certain time budget for testing.[1] Testing approaches, which are purely based on uniform random sampling, clearly do not adhere to the aforementioned criteria. For example, consider a system that expects an integer value as an input. For such a system uniform random sampling may blindly continue to generate test inputs forever without providing any information about the correctness (or in-correctness) of the system. However, there will be systems in the wild that are too complex to model. Such systems require some sort of mechanism by which they can be tested to some extent. For such systems, the sampling based technique, as discussed in the following paragraphs, might be useful.

The work in [2, 3] proposes sampling based techniques to generate failure-revealing test inputs for complex embedded systems. In particular, they focus on generating test inputs that lead to violation of timing-related properties. For these techniques to work, the essential timing-related properties of the system must be formulated via Metric Temporal Logic (MTL). An MTL formula can be, in a broad way, described as a composition of propositional as well as temporal operators. Common examples of propositional operators are *conjunction*, *disjunction*, and *negation*, whereas some example of temporal operators would be *until*, *always*, and *eventually*. Besides, MTL extends the traditional linear temporal logic (LTL) with timing constraints. For instance, consider our example in Fig. 1. Let us consider that a potential fall of the patient must be reported within 100 time units. Such a criteria can be captured via the following MTL formula:

$$\Box(\mathit{fall} \rightarrow \Diamond_{(0,100)}\mathit{alarm})$$

`fall` captures the event of a potential fall and `alarm` captures the event to notify the health care providers. Besides, the temporal operators $\Box$ and $\Diamond$ capture *always* and *eventually*, respectively. Once the timing-related properties of the system have been identified and encoded as MTL formulas, the next step is to identify test inputs (as shown in Fig. 5), for which the aforementioned formula do not hold true (ie, the system fails).

[1] Otherwise, the testing process should terminate with assurance that the system functionality is expected under all feasible circumstances.

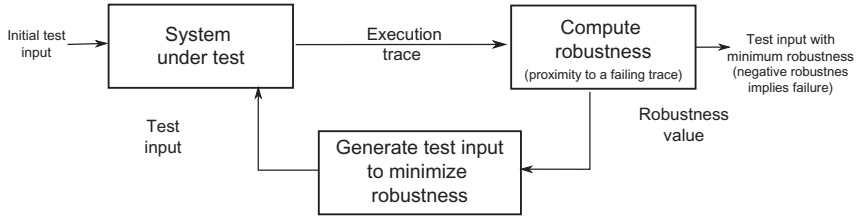

Figure 5 Overview of sampling based test-generation techniques.

The cornerstone of sampling-based approaches lies in the definition of a metric, as often called *robustness metric*. Such a metric represents the distance of a given execution trace (of the SUT, for a given input) from a failure revealing execution trace. The metric is designed in such a manner that if an execution trace has a negative value for the *robustness metric*, then it implies that the respective execution has lead to a violation of some timing-related property. Similarly, a positive value for a robustness metric signifies that the execution satisfies the MTL formulas. In general, the *robustness metric* provides a measure of how robustly an execution trace satisfies the encoded MTL formulas. Once such a metric has been defined, it needs to be decided *whether there exists an input that leads to the violation of the given property*. This decision problem can be transformed into an optimization problem. For instance, this optimization problem might aim to discover the execution with the lowest robustness value. Existing works have discussed a number of ways of solving the optimization (minimizing robustness) problem. For example, the technique of [2] uses Monte-Carlo simulations to solve this optimization problem. An obvious drawback being that the technique of [2] can only give probabilistic guarantees to find failure inducing test inputs. At the same time, an advantage of such a technique is to find execution where the timing-related property was the closest to being violated. Subsequent work in this direction have experimented with other optimization techniques, such as [3] uses Cross-entropy method based optimization and [4] uses ant-colony based optimization, in trying to improve the efficiency of the test-generation process.

5. GREY-BOX ABSTRACTION

This class of techniques work by creating an abstract model of the SUT. As shown in Fig. 6, in general, frameworks discussed in this category require three key components as follows:

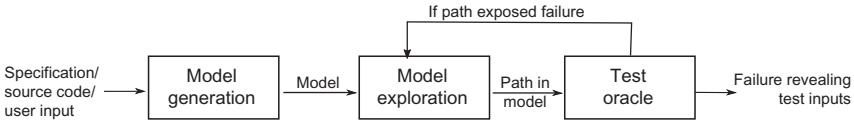

Figure 6 Overview of grey-box abstraction based testing techniques.

- A technique for model generation
- A technique for model exploration (to generate test cases), and
- An oracle for identifying failure-revealing tests

Once the property of interest has been identified, the model of the SUT can be generated through an automatic, semi-automatic or manual approaches. The model can be generated by analyzing the system specification, the source code or the environment. The generated model is then explored using a wide variety of techniques, ranging from random walk of the model to evolutionary or genetic algorithms. Test oracle is a critical component of the framework and it is used to differentiate between the correct and incorrect system execution. A test oracle is used to identify failure-revealing test inputs, while exploring the model of the SUT. The efficacy of the test-generation technique largely depends on the level of abstraction of the model and the efficiency of the exploration algorithm. A coarse-grained model is relatively easy to create and explore, but it may miss some of the important (failure-revealing) scenarios. On the contrary, a very detailed and fine-grained model is difficult to create and explore. However, such a fine-grained model is likely to discover more failure revealing test inputs. Considering the accuracy and precision of abstraction, we further classify the techniques in this category, based on the respective models used for testing. In the following sections, we describe each such model in more details.

5.1 Timed State Machines

Modelings tools, such as Markov chains have been used to model and test systems for a long time. To be more specific, Markov Chain Usage Models (MCUM) can be described as directed graphs, where the nodes of the graph represent the states of the SUT. The nodes of the system are connected by edges, representing events (inputs) that may arrive at a given state of the system. Additionally, edges are annotated with the probability of the occurrence of an event, when the system is in a given state. However MCUMs, by themselves, do not provide a suitable way of representing the timing-related properties of the SUT. Such timing-related properties may require certain events to happen before, after or within a specific deadline. Since timing-related requirements are often an integral part of real-time embedded

system (eg, in automotive applications), MCUMs were extended to capture such requirements. One of the earliest such extensions of MCUMs was proposed in [5], where the extended MCUMs are referred to as Timed Usage Models (TUMs). Similar to the conventional MCUMs, all paths, from the start state to the end state in a TUM, represent feasible executions of the SUT. Figure 7A provides a simple example of a timed usage model. However, there also exists some key differences between an MCUM and a TUM that are listed in the following:

- Similar to the conventional MCUMs, a TUM has a set of states to capture the feasible usage of the system. However, in TUM, an additional probability distribution function (*pdf*) is associated with each state. This *pdf* encodes the time, for which the SUT will be in the respective state.
- In TUM, each transition between two states is triggered by a stimulus. Additionally, edges connecting the states are associated with two variables, a transition probability and a probability distribution function (*pdf*) of stimulus time. As the name suggests, the transition probability captures the probability of the respective transition between two states. Therefore, the transition probability has a similar role to that of conventional MCUMs. The *pdf* of the stimulus time represents the duration of execution of the stimulus on the system, at a given state.
- In a deterministic MCUM, there could be at most one transition (from a given state) for a given stimulus. However, in a TUM, the next state not only depends on the stimulus, but also on the duration of the execution of the stimulus. This feature is required to capture timing-related dependencies in the system. Additionally, to maintain consistency, the *pdfs* of stimulus time, originating from a state, do not overlap.

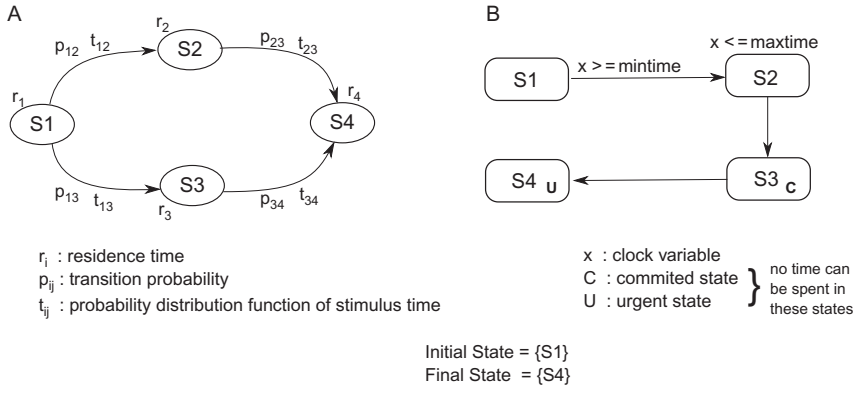

Figure 7 Simple example showing (A) timed usage model and (B) timed automata.

Once the model of the system has been created, a variety of model-exploration techniques can be used to generate test cases. For instance [5] and [6] perform a simple random walk of the TUM model to generate test cases while other works such as [7] and [8], have designed coverage metrics to guide the test-generation process. In particular, works in [7] and [8], combine the usage of TUMs with dependencies between the different components of the SUT. This allows them to generate test cases that not only represent different timing scenarios, but also capture dependencies between critical system components.

Another line of work [9] propose to extend finite state machines model (FSM) to incorporate timing-related constraints. Such a model is most commonly known as *timed automata* (TA). In timed automata, an FSM is augmented with a finite number of clocks. These clocks are used to generate boolean constraints and such constraints are labeled on the edges of the TA. Additionally, clock values can be manipulated and reset by different transitions of the automata. The boolean constraints succinctly captures the criteria for the respective transition being triggered. Timed automata also has the feature to label time-critical states. For instance, states marked as *Urgent* or *Committed* imply that no time can be spent in these states. Besides, while exploring the model, certain states (such as states marked as *Committed*), have priority over other states. These additional features make the process of modeling intuitive and also make the model easier to read. Figure 7B provides a simple example of timed automata. A major difference between the works (eg, works in [5–8]) that use TUM as a modeling approach as compared to works (eg, work in [9]) that use timed automata, is in the model exploration. Whereas the former use either random or guided walks of the model to generate test cases, the later use evolutionary algorithms to explore the model and generate test cases.

5.2 Markov Decision Process

One of the key assumptions, which were made while designing TUMs (as described in the preceding section), was that the probability distributions for transitions were known *a priori*. This is usually true for deterministic systems. However, as argued by the work in [10], such transition probabilities are often unavailable for non-deterministic systems. Therefore, when testing non-deterministic systems for non-functional properties, such as reliability, TUMs do not present a suitable approach. For such systems, the work of [10] proposes an approach based on Markov-Decision Process (MDP). In particular, the system-level modeling is performed via MDPs. Since MDPs

can support non-determinism, it is a suitable platform for capturing the non-determinism in a system. Once an MDP model is created for the system, the work in [10] uses a combination of hypothesis testing and probabilistic model checking to check reliability constraints. Hypothesis testing is a statistical mechanism, in which, one of many competing hypothesis are chosen based on the observed data. In particular, [10] uses hypothesis testing to obtain the reliability distribution of the deterministic components in the system. Such reliability distribution are computed within the specific error bounds that the user needs to provide. Subsequently, probabilistic model checking is used on MDPs to compute the overall reliability of the system.

The work of [11] uses a similar technique to obtain reliability distribution for a real life, healthcare system. The system tested in [11] is an ambient-assisted-living-system for elderly people with dementia. Such embedded systems must function reliably under all operating scenarios. Failure to do so may cause serious injuries to the respective patients. For such systems, the non-determinism in the environmental factors (eg, the input from the sensors and human behavior) makes the system complex and make it challenging to produce the required reliability assurances. However, the work of [11] has shown that an MDP-based approach can be effectively used to test complex, real life systems in a scalable and efficient manner.

5.3 Unified Modeling Language

A different line of work [12–14] uses Unified Modeling Language (UML) to model and test real-time systems for timing-related and safety-related properties. UML provides a well known (and well accepted) standard for software modeling and it is used in different dimensions of software testing. In UML, the structure of a system under test can easily be represented via the utilities provided by UML, such as object diagrams and components diagrams. Additionally, the behavior of the modeled system can be represented by use cases, state charts or message-sequence charts. However, for modeling embedded, real-time systems, UML needs to be extended with additional packages, such as packages for non-functional properties and scheduling or management of resources. These packages can be availed through Modeling and Analysis of Real time Embedded Systems Extension (MARTE) of UML. In particular, constraints (such as timing-related constraints) on real-time system can be captured through a standard language known as Object Constraint Language (OCL). Once the system is modeled with appropriate constraints, failure-inducing test cases can be generated by exploring the model. For instance, the search techniques in [12, 13]

compares the effectiveness of their test-generation process for random testing, adaptive random testing and evolutionary algorithms, while other works [14] experiment with the effectiveness of genetic algorithms as a search strategy. These works observe that, at least for the evaluated case studies, none of the search strategies (for test generation) were definitively superior than others. However, subsequent works [13] have claimed better efficiency, when searching for failure-inducing test cases, through hybrid search strategies.

5.4 Event Flow Graph

Systematic testing of event-driven applications for non-functional properties, such as energy consumption, is a challenging task. This is primarily because of the fact that like any other non-functional property, information related to energy consumption is seldom present in the source code. Additionally, such information may differ across different devices. Therefore, generating energy-consumption annotation, for each application and device, is definitely time consuming and error-prone. A real-life scenario of such event-driven systems is mobile applications. Mobile applications are usually executed on battery-constrained systems, such as smartphones. Smartphones, in turn, are equipped with energy-hungry components, such as GPS, WiFi and display. This necessitates the development of efficient and automated testing technique to stress energy consumption. One such technique has been presented in [15]. It automatically generates the Event Flow Graph (EFG) [16] of the application under test. An EFG can be described as a directed graph, where the nodes of the graph represent events and the edges capture the *happens-after* relationship between any two events. It is possible (and often the case) that EFGs of mobile applications have cycles (such as the example shown in Fig. 8). Such cycles typically do not have an explicit iteration bounds. Therefore, although an EFG has a finite number of events, an unbounded number of event sequences can be generated from the same. This further complicates the process of test generation, as any effective testing technique should not only be able to generate all failure-revealing test cases, but also do so in a reasonable amount of time.

The framework presented in [15] has two key innovations that helps it to tackle the challenges described in the preceding paragraph. The first of those two innovations being the definition of a metric that captures the energy inefficiency of the system, for a given input. To design such a metric, it is important to understand what exactly qualifies as energy-inefficient

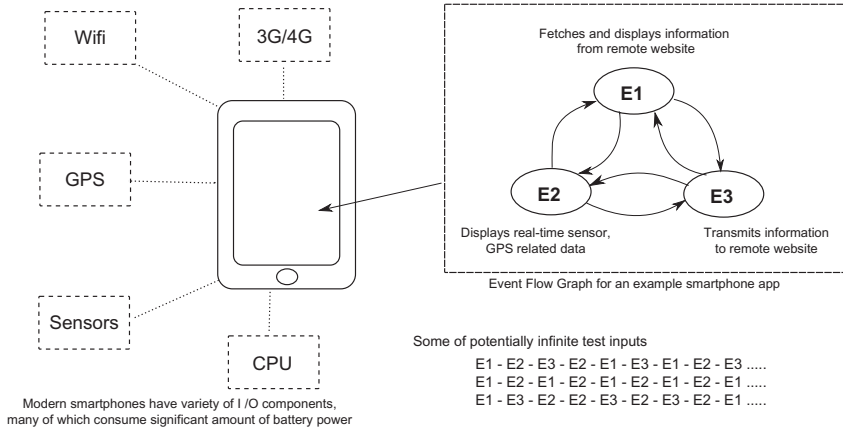

Figure 8 Modern smartphones have a wide variety of I/O and power management utilities, improper use of which in the application code can lead to suboptimal energy-consumption behavior. Smartphone application are usually nonlinear pieces code, systematic testing of which requires addressing a number of challenges.

behavior. In other words, let us consider the following question: *Does high-energy consumption always imply higher energy-inefficiency?* As it turns out [15], the answer to this question is not trivial. For instance, consider a scenario where two systems have similar energy-consumption behavior but one is doing more work (has a higher utilization of its hardware components) than the other. In such a scenario, it is quite intuitive that the system with higher utilization is the more energy-efficient one. Taking inspiration from this observation, the work in [15] defines the metric of *E/U ratio* (energy consumption vs utilization) to measure the energy inefficiency of a system. For a given input, the framework executes the application on a real hardware device and analyses the *E/U ratio* of the device at runtime. An anomalously high *E/U ratio*, during the execution of the application, indicates the presence of an energy hotspot. Additionally, a consistently high E/U ratio, after the application has completed execution, indicates the presence of an energy bug. In general, energy bugs can cause more wastage of battery power than energy hotspots and can drastically reduce the operational time of the smartphone. With the metric of *E/U ratio*, it is possible to find energy-inefficient behavior in the SUT, for a given input. However, another challenge is to generate inputs to stress energy behavior of a given application, in a reasonable amount of time. Interestingly, for smartphone applications, a number of previous studies (primarily based on Android operating system) have observed that most of the energy-hungry components can only be

accessed through a predefined set of system calls. The work in [15] uses this information to prioritize the generation of test inputs. In particular, [15] uses a heuristic-based approach. This approach tries to explore all event traces that may invoke system calls to energy-hungry components. Besides, the work also prioritizes inputs that might invoke a similar sequence of system calls compared to an already discovered, energy-inefficient execution.

6. WHITE-BOX ABSTRACTION

In this section, we shall discuss software testing methodologies that are carried out directly on the implementation of an application. Such an implementation may capture the source code, the intermediate code (after various stages of compilation) or the compiled binary of an embedded software. Whereas we only specialize the testing procedures at the level of abstractions they are carried out, we shall observe in the following discussion that several methodologies (eg, evolutionary testing and symbolic execution) can be used to test the implementation of embedded software. The idea of directly testing the implementation is promising in the context of testing embedded software. In particular, if the designer is interested in accurately evaluating the non-functional behaviors (eg, energy and timing) of different software components, such non-functional behaviors are best observed at the level of implementation. On the flip side, if a serious bug was discovered in the implementation, it may lead to a complete redesigning of the respective application. In general, it is important to figure out an appropriate level of abstraction to run the testing procedure. We shall now discuss several works to test the implementation of embedded software and reason about their implications. In particular, we discuss testing methodologies for timing-related properties in Section 6.1 and for functionality-related behaviors in Section 6.2. Finally, in Section 6.3, we discuss challenges to build an appropriate framework to observe and control test executions of embedded software and we also describe some recent efforts in the software engineering community to address such challenges.

6.1 Testing Timing-related Properties

The work in [17] shows the effectiveness of evolutionary search for testing embedded software. In particular, this work targets to discover the maximum delay caused due to *interrupts*. In embedded software, interrupts are common phenomenon. For instance, the incoming signals from sensors or network events (eg, arrival of a packet) might be captured via interrupts.

Besides, embedded systems often consist of multiple tasks, which share resources (eg, CPU and memory). As a result, switching the CPU from a task t to a task t' will clearly induce additional delay to the task t. Such switching of resources are also triggered via interrupts. Therefore, the delay caused due to interrupts might substantially affect the overall timing behavior. For instance, in the fall detection application, each sensor might be processed by a different task and another task might be used to compute a potential fall of the patient. If all these tasks share a common CPU, a particular task might be delayed due to the switching of CPU between tasks.

Fig. 9 illustrates scenarios where the task to compute a fall is delayed by interrupts generated from the accelerometer and the gyroscope. In particular, Fig. 9A demonstrates the occurrence of a single interrupt. On the contrary, Fig. 9B illustrates nested interrupts, which prolonged the execution time of the computation task. In general, arrival of an interrupt is highly non-deterministic in nature. Moreover, it is potentially infeasible to test an embedded software for all possible occurrences of interrupts.

The work in [17] discusses a genetic algorithm to find the maximum interrupt latency. In particular, this work shows that a testing method based on genetic algorithm is substantially more effective compared to random testing. This means that the interrupt latency discovered via the genetic algorithm is substantially larger than the one discovered using random testing. An earlier work [18] also uses genetic algorithm to find the WCET of a program. In contrast to [17], the work in [18] focuses on the uninterrupted execution of a single program. More specifically, the testing method, as proposed in [18], aims to search the input space and more importantly, direct the search toward WCET revealing inputs.

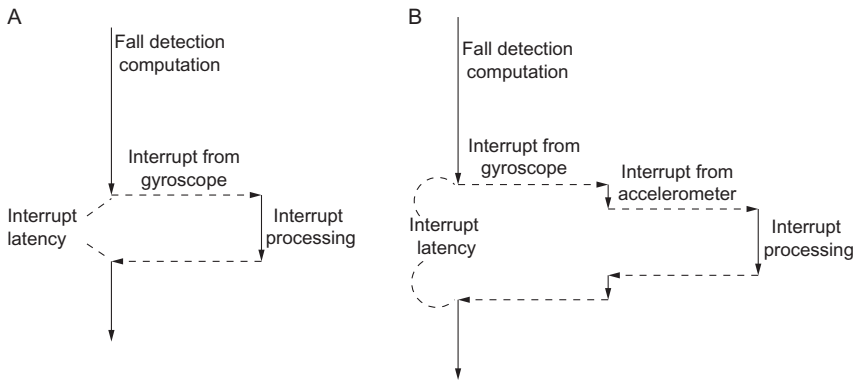

Figure 9 Interrupt latency, (A) single interrupt and (B) Nested interrupts.

It is well known that the processing power of CPUs have increased dramatically in the last few decades. In contrast, memory subsystems are several order of magnitudes slower than the CPU. Such a performance gap between the CPU and memory subsystems might be critical for embedded software, when such software are restricted via timing-related constraints. More specifically, if the software is spending a substantial amount of time in accessing memory, then the performance of an application may have a considerable slowdown. In order to investigate such problems, some recent efforts in software testing [19, 20] have explicitly targeted to discover memory bottlenecks. Such efforts directly test the software binary to accurately determine requests to the memory subsystems. In particular, requests to the memory subsystems might be reduced substantially by employing a cache. Works in [19, 20] aim to exercise test inputs that lead to a poor usage of caches. More specifically, the work in [19] aims to discover *cache thrashing scenarios*. A cache thrashing scenario occurs when several memory blocks replace each other from the cache, hence, generating a substantial number of requests to the memory subsystems. For instance, the code fragment in Fig. 10 may exhibit a cache thrashing when the cache can hold exactly one memory block. In the code fragment, `m1` and `m2` replace each other from the cache, leading to a *cache thrashing*. This behavior is manifested only for the program input `'t'`.

The work in [19] shows that the absence of such cache thrashing scenarios can be formulated by systematically transforming the program with assertions. Subsequently, a search procedure on the software input space can be invoked to find violation of such assertions. Any violation of an assertion, thus, will produce a cache thrashing scenario. The methodology proposed in [19] uses a combination of static analysis and symbolic execution to search the input space and discover inputs that violate the formulated assertions.

```
int n = 100;

while (n-- >= 0) {
  if (input == 't') {
     //access memory block m1
     //access memory block m2
  } else {
     //access memory block m1
  }
```

Figure 10 Input dependent cache thrashing.

The work in [20] lifts the software testing of embedded software for massively parallel applications, with a specific focus on general-purpose graphics processing units (GPGPU). It is well known that future technology will be dominated by parallel architectures (eg, multicores and GPGPUs). For such architectures, software testing should take into account the input space of the application, as well as the non-deterministic nature of scheduling multiple threads. The work in [20] formally defines a set of scenarios that capture memory bottlenecks in parallel architectures. Subsequently, a search procedure is invoked to systematically traverse the input space and the space consisting of all possible scheduling decisions among threads. Like the approach in [19], the work in [20] also uses a combination of static analysis and symbolic execution for the search. In summary, both the works [19, 20] revolve around detecting fine-grained events such as memory requests. In general, such fine-grained events are appropriate to test only at the implementation level (eg, software binary). This is because the occurrence of such events would be significantly difficult to predict at intermediate stages of the development.

6.2 Testing Functionality-related Properties

In the preceding Section, we have discussed software testing methodologies that focus on validating timing-related constraints of embedded software. In contrast to such methodologies, the work in [21] primarily targets functional properties of embedded software. In particular, authors of [21] discuss some unique challenges that might appear only in the context of testing embedded software and systems. The key observation is that embedded systems often contain different layers of hardware and software. Besides, an embedded application may contain multiple tasks (eg, programs) and such tasks might be active simultaneously. For instance, in our fall-detection application, the access to hardware components (eg, gyroscope and accelerometers) might be controlled by a supervisory software, such as operating systems (OS). Similarly, sampling signals from sensors and computation of a potential fall might be accomplished by different tasks that run simultaneously in the system. The work in [21] argues the importance of testing interactions between different hardware/software layers and different tasks. Fig. 11 conceptually captures such interactions in a typical embedded system.

In order to exercise interactions between tasks and different software layers, authors of [21] have described a suitable coverage criteria for testing embedded systems. For instance, the interaction between application layer

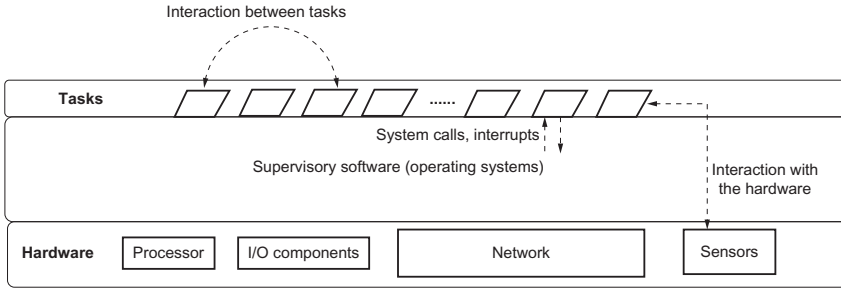

Figure 11 Interaction among different tasks and hardware/software layers.

```
f = 0;
if (input = 'x') {
  g := g + 1;
  f := f + 1;
}
if (f != 0)
  syscall(g);
else
  syscall(0);
```

Figure 12 Interaction between application layer variable and operating system.

and OS layer can happen via system calls. Similarly, the application might directly access some hardware components via a predefined set of application programmer interfaces (APIs). The work in [21] initially performs a static analysis to infer data dependencies across different layers of the embedded system. Besides, if different tasks of the system use shared resources, such an analysis also tracks the data dependencies across tasks. For instance, consider the piece of code fragment in Fig. 12, where `syscall` captures a system call implemented in the kernel mode. In the code shown in Fig. 12, there exists a data dependency between application layer variable `g` and the system call `syscall`. As a result, it is important to exercise this data dependency to test the interaction between application layer and OS layer. Therefore, the work in [21] suggests to select test cases that can manifest the data dependency between variable `g` and `syscall`. To illustrate the dependency between multiple tasks, let us consider the code fragment in Fig. 13.

The keyword `__shared__` captures shared variables. In Fig. 13, there is a potential data dependency between `Task 1` and `Task 2`. However, to exercise this data dependency, the designer must be able to select an input that satisfies the condition `input == 'x'`. The work in [21] performs static analysis to discover the data dependencies across tasks, as shown in this example. Once

```
__shared__ int s = 0;

Task 1:                    Task 2:
int y;
y := y + 2;                if (s > 0)
if (input = 'x')             //do something
  s = y;
else
  y = y - 1;
```

Figure 13 Interaction between tasks via shared resources (shared variable `s`).

all data dependencies are determined via static analysis, the chosen test inputs aim to cover these data dependencies.

6.3 Building Systematic Test-execution Framework

So far in this section, we have discussed test input generation to validate either functional or non-functional properties of embedded software. However, as discussed in [22, 23], there exists numerous other challenges for testing embedded software, such as *observability* of faulty execution. *Test oracles* are usually required to observe faulty execution. Designing appropriate oracles is difficult even for traditional software testing. In the context of embedded software, designing oracles may face additional challenges. In particular, as embedded systems consist of many tasks and exhibit interactions across different hardware and software layers, they may often have non-deterministic output. As a result, oracles, which are purely based on output, are insufficient to observe faults in embedded systems. Moreover, it is cumbersome to build output-based oracles for each test case. In order to address these challenges, authors in [22] propose to design property-based oracles for embedded systems. Property-based oracles are designed for each execution platform. Therefore, any application targeting such execution platform might reuse the oracles and thereby, it can avoid substantial manual efforts to design oracles for each test case. The work in [22] specifically targets concurrency and synchronization properties. For instance, test oracles are designed to specify proper usage of binary semaphores and message queues, which are used for synchronization and interprocess communication, respectively. Such synchronization and interprocess communication APIs are provided by the operating system. Once test oracles are designed, a test case can be executed, while instrumenting the application, OS and hardware interfaces simultaneously. Each execution can subsequently be checked for violation of properties captured by an oracle. Thus property-based test

oracles can provide a clean interface to observe faulty executions. Apart from test oracles, authors in [23] discuss the importance of giving the designer appropriate tools that control the execution of embedded systems. Since the execution of an embedded system is often non-deterministic, it is, in general difficult to reproduce faulty executions. For instance, consider the fall detection application where a task reads sensor data from a single queue. If new data arrives, an interrupt is raised to update the queue. It is worthwhile to see the presence of a potential data race between the routine that services the interrupt and the task which reads the queue. Unfortunately, the arrival of interrupts is highly non-deterministic in nature. As a result, even after multiple test executions, the testing may not reveal a faulty execution that capture a potential data race. In order to solve this, authors in [23] design appropriate utilities that gives designer the power to raise interrupts explicitly. For instance, the designer might choose a set of locations where she suspects the presence of data races due to interrupts. Subsequently, a test execution can be carried out that raise interrupts exactly at the locations specified by the designer.

Summary

To summarize, in this section, we have seen efforts to generate test inputs and test oracles to validate both functional and non-functional aspects of embedded software. A common aspect of all these techniques is that the testing process is carried out directly on the implementation. This might be appealing in certain scenarios, for instance, when the designer is interested in events that are highly sensitive to the execution platform. Such events include interrupts, memory requests and cache misses, among others.

7. FUTURE DIRECTIONS

As discussed in this chapter, analysis of non-functional properties is crucial to ensure that embedded systems behave as per its specification. However, there exists an orthogonal direction of work, where analysis of non-functional properties, such as power consumption, memory accesses and computational latencies, have been used for security-related exploits. Such exploits are commonly referred to as side-channel attacks and are designed to extract private keys[2] from cryptographic algorithms, such as

[2] Cryptographic algorithms such as AES and DES are used to encrypt a message in a manner such that only the person having the private key is capable of decrypting the message.

algorithms used in smart cards and smart tokens. The intention of the attacker is not to discover the theoretical weaknesses of the algorithm. Instead, the attacker aims to break the *implementation* of the algorithms through side channels, such as measuring execution time or energy consumption. In particular, the attacker tries to relate such measurements with the secret key. For instance, if different secret keys lead to different execution time, the attacker can perform statistical analysis to map the measured execution time with the respective key. In general, any non-functional behavior that has a correlation with cryptographic computation, is capable of leaking information, if not managed appropriately. For example, the differential power attack, as proposed in [24], uses a simple, yet effective statistical analysis technique to correlate the observed power-consumption behavior to the private key. Since then, a number of subsequent works have proposed counter-measures (eg, [25]) against side-channel vulnerabilities and bypasses to those counter-measures (eg, [26]). Similarly, researchers have also studied side-channel attacks (and their counter-measures) based on other non-functional behaviors, such as computational latency [27, 28] and memory footprint [29]. Even though works on side-channel attacks have a very different objective compared to those on non-functional testing, there exists a number of commonalities. In essence, both lines of work are looking for test inputs that lead to *undesirable non-functional behavior*. The definition of the phrase *undesirable non-functional behavior* is based on the system under test (SUT). For instance, in an embedded system that has hard timing-related constraints, an undesirable input would be the violation of such constraints. On the contrary, for a cryptographic algorithm, such as implemented in a smart card, an undesirable input may lead to information leaks via side channels. Undesirable non-functional behavior in one scenario may lead to performance loss, sometimes costing human lives (such as in an anti-lock braking system), whereas, in the other scenario undesirable non-functional behavior may cause information leaks, which, in turn may often lead to financial losses. It is needless to motivate the fact that testing embedded cryptographic systems for such undesirable non-functional behaviors is crucial. More importantly, testing methodologies for detecting side-channel attacks need to be automated. However, as of this writing, this line of research is far from being solved. New works on this topic could draw inspiration from earlier works on non-functional testing, such as works described in Section 3.

Another more generic direction is related to the detection of root cause and automatic repair of non-functional properties in embedded systems. In general, the purpose of software testing is to expose suboptimal or unwanted

behavior in the SUT. Such suboptimal behaviors, once identified, should be rectified by modifying the system. More specifically, the rectification process can be subdivided into two parts: fault-localization[3] and root-cause detection, followed by debugging and repair. Fault-localization is, in general, the more time-consuming (and expensive) phase of this modification process and therefore, there is a huge demand for effective, automated techniques for fault-localization. Over the past years, several works have proposed methodologies for fault-localization. However, most of these works have focused on the functionality of software. As of this writing, there exists a lack of efforts in fault-localization techniques for non-functional properties. One plausible explanation can be that designing such a framework, for non-functional properties, is significantly more challenging. This is because the non-functional behavior of system depends not only on the source code, but also on the underlying execution platform. Some of the well known techniques [30] for fault-localization include comparing a set of failed execution to a set of passing executions and subsequently, deriving the root cause for the fault. Such works narrow down the search space for the root cause by assigning suspiciousness values to specific regions of source code. As is the case with fault-localization, considerable research needs to be performed on automated debugging and repair of non-functional software properties. As a matter of fact, automated program repair, even in the context of software functionality, is far from being matured, not to mention the lack of research for non-functional software properties. As for embedded software and systems, both functional and non-functional behaviors play crucial roles in validating the respective system. We hope that future works in software testing will resolve these validation challenges faced by embedded system designers.

8. CONCLUSION

Embedded systems are ubiquitous in the modern world. Such systems are used in a wide variety of applications, ranging from common consumer electronic devices to automotive and avionic applications. A property common to all embedded systems is that they interact with the physical environment, often deriving their inputs from the surrounding environment. Due to the application domains such systems are used in, their behavior is often constrained by functional (such as the input–output relationship) as well as

[3] In this context, the word "fault" implies all type of suboptimal, non-functional behavior.

non-functional properties (such as execution time or energy consumption). This makes the testing and validation of such systems a challenging task. In this chapter, we discussed a few challenges and their solutions in the context of testing embedded systems. In particular, we take a closer look into existing works on testing non-functional properties, such as timing, energy consumption, reliability, for embedded software. To put the existing works in perspective, we classify them in three distinct categories, based on the level of system abstraction used for testing. These categories include,*black-box*, *grey-box* and *white-box abstraction* based testing approaches. In general, *black-box abstraction* based testing methods use sampling based techniques to generate failure-revealing test cases for the system under test. Such methods consider the system as a black-box and hence are equally applicable to simple and complex systems alike. However, such ease of use usually comes at the cost of effectiveness. In particular, these methods often cannot provide completeness guarantees (ie, by the time the test-generation process completes, all failure revealing test inputs must have been uncovered). The *grey-box abstraction* based approaches are usually more effective than the *black-box abstraction* based approaches. This is because such methods often employ an abstract model of the system under test to generate failure-revealing test cases. Effectiveness of these test-generation methodologies is often dictated by the level of system abstraction being used. *White-box abstraction* based testing approaches use the actual system implementation to generate failure revealing test cases and hence are capable of providing maximum level of guarantee to discover failure revealing inputs. We observe that existing techniques vary hugely in terms of complexity and effectiveness. Finally, we have discussed future research directions related to embedded software testing. One of which was automated fault-localization and repairing of bugs related to non-functional properties. Another direction was related to the development of secure embedded systems. In particular, we explored the possibility of testing techniques to exploit the vulnerability toward side-channel attacks. Over the recent years, there have been a number of works, which analyze non-functional behavior to perform side-channel (security related) attacks. It would be appealing to see how existing testing methodologies can be adapted to test and build secure embedded software.

ACKNOWLEDGMENT

The work was partially supported by a Singapore MoE Tier 2 grant MOE2013-T2-1-115 entitled "Energy aware programming" and the Swedish National Graduate School on Computer Science (CUGS).

REFERENCES

[1] A wearable miniaturized fall detection system for the elderly. http://www.fallwatch-project.eu/press_release.php.

[2] T. Nghiem, S. Sankaranarayanan, G. Fainekos, F. Ivancić, A. Gupta, G.J. Pappas, Monte-carlo techniques for falsification of temporal properties of non-linear hybrid systems, in: Proceedings of the 13th ACM International Conference on Hybrid Systems: Computation and Control, HSCC '10, 2010.

[3] S. Sankaranarayanan, G. Fainekos, Falsification of temporal properties of hybrid systems using the cross-entropy method, in: Proceedings of the 15th ACM International Conference on Hybrid Systems: Computation and Control, HSCC '12, 2012.

[4] Y.S.R. Annapureddy, G.E. Fainekos, Ant colonies for temporal logic falsification of hybrid systems, in: IECON 2010–36th Annual Conference on IEEE Industrial Electronics Society, 2010.

[5] S. Siegl, K. Hielscher, R. German, Introduction of time dependencies in usage model based testing of complex systems, in: Systems Conference, 2010 4th Annual IEEE, 2010, pp. 622–627.

[6] S. Siegl, K. Hielscher, R. German, C. Berger, Formal specification and systematic model-driven testing of embedded automotive systems, in: 4th Annual IEEE Systems Conference, 2010, 2011.

[7] S. Siegl, P. Caliebe, Improving model-based verification of embedded systems by analyzing component dependences, in: 2011 6th IEEE International Symposium on Industrial Embedded Systems (SIES), 2011, pp. 51–54.

[8] P. Luchscheider, S. Siegl, Test profiling for usage models by deriving metrics from component-dependency-models, in: 2013 8th IEEE International Symposium on Industrial Embedded Systems (SIES), 2013, pp. 196–204.

[9] J. Hansel, D. Rose, P. Herber, S. Glesner, An Evolutionary algorithm for the generation of timed test traces for embedded real-time systems, in: 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation (ICST), 2011.

[10] L. Gui, J. Sun, Y. Liu, Y.J. Si, J.S. Dong, X.Y. Wang, Combining model checking and testing with an application to reliability prediction and distribution, in: Proceedings of the 2013 International Symposium on Software Testing and Analysis, ISSTA 2013, 2013.

[11] Y. Liu, L. Gui, Y. Liu, MDP-based reliability analysis of an ambient assisted living system, in: FM 2014: Formal Methods, Lecture Notes in Computer Science, vol. 8442, Springer International Publishing, 2014.

[12] A. Arcuri, M.Z. Iqbal, L. Briand, Black-box system testing of real-time embedded systems using random and search-based testing, in: Proceedings of the 22Nd IFIP WG 6.1 International Conference on Testing Software and Systems, ICTSS'10, 2010, pp. 95–110.

[13] M.Z. Iqbal, A. Arcuri, L. Briand, Combining search-based and adaptive random testing strategies for environment model-based testing of real-time embedded systems, in: Proceedings of the 4th International Conference on Search Based Software Engineering, SSBSE'12, 2012.

[14] M.Z. Iqbal, A. Arcuri, L. Briand, Empirical investigation of search algorithms for environment model-based testing of real-time embedded software, in: Proceedings of the 2012 International Symposium on Software Testing and Analysis, ISSTA 2012, 2012.

[15] A. Banerjee, L.K. Chong, S. Chattopadhyay, A. Roychoudhury, Detecting energy bugs and hotspots in mobile apps, in: Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2014.

[16] A.M. Memon, I. Banerjee, A. Nagarajan, GUI ripping: reverse engineering of graphical user interfaces for testing, in: Working Conference on Reverse Engineering, 2003, pp. 260–269.

[17] S. Weissleder, H. Schlingloff, An evaluation of model-based testing in embedded applications, in: 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation (ICST), 2014, pp. 223–232.
[18] P.P. Puschner, R. Nossal, Testing the results of static worst-case execution-time analysis, in: IEEE Real-Time Systems Symposium, 1998, pp. 134–143.
[19] A. Banerjee, S. Chattopadhyay, A. Roychoudhury, Static analysis driven cache performance testing, in: Real-Time Systems Symposium (RTSS), 2013 IEEE 34th, 2013, pp. 319–329.
[20] S. Chattopadhyay, P. Eles, Z. Peng, Automated software testing of memory performance in embedded GPUs, in: 2014 International Conference on Embedded Software (EMSOFT), 2014, pp. 1–10.
[21] T. Yu, A. Sung, W. Srisa-An, G. Rothermel, An approach to testing commercial embedded systems, J. Syst. Softw. 88 (2014).
[22] T. Yu, A. Sung, W. Srisa-an, G. Rothermel, Using property-based oracles when testing embedded system applications, in: 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation (ICST), 2011, pp. 100–109.
[23] T. Yu, W. Srisa-an, G. Rothermel, SimTester: a controllable and observable testing framework for embedded systems, in: Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments, VEE '12, London, England, UK, ISBN 978-1-4503-1176-2, 2012.
[24] P. Kocher, J. Jaffe, B. Jun, Differential power analysis, 1998. http://www.cryptography.com/public/pdf/DPA.pdf.
[25] M.-L. Akkar, C. Giraud, An implementation of DES and AES, secure against some attacks, in: Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems, CHES '01, 2001.
[26] S. Mangard, N. Pramstaller, E. Oswald, Successfully attacking masked AES hardware implementations, in: Cryptographic Hardware and Embedded Systems, CHES 2005, Lecture Notes in Computer Science, 2005.
[27] P. Kocher, Timing attacks on implementations of diffe-hellman, RSA, DSS, and other systems. http://www.cryptography.com/public/pdf/TimingAttacks.pdf.
[28] B. Köpf, L. Mauborgne, M. Ochoa, Automatic quantification of cache side-channels, in: Proceedings of the 24th International Conference on Computer Aided Verification, CAV'12, Berkeley, CA, Springer-Verlag, Berlin, ISBN 978-3-642-31423-0, 2012, pp. 564–580, http://dx.doi.org/10.1007/978-3-642-31424-7_40.
[29] S. Jana, V. Shmatikov, Memento: learning secrets from process footprints, in: Proceedings of the 2012 IEEE Symposium on Security and Privacy, SP '12, IEEE Computer Society, Washington, DC, ISBN 978-0-7695-4681-0, 2012, pp. 143–157, http://dx.doi.org/10.1109/SP.2012.19.
[30] J.A. Jones, M.J. Harrold, Empirical evaluation of the tarantula automatic fault-localization technique, in: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, ASE '05, Long Beach, CA, USA, ACM, New York, NY, ISBN 1-58113-993-4, 2005, pp. 273–282, http://dx.doi.org/10.1145/1101908.1101949.

ABOUT THE AUTHORS

Abhijeet Banerjee is a Ph.D. scholar at the School of Computing, National University of Singapore. He received his B.E. in Information Technology from Indian Institute of Engineering Science and Technology, Shibpur, India in 2011. His research interests include automated software testing, debugging, and re-factoring with specific emphasis on testing and verification of non-functional properties of software.

Sudipta Chattopadhyay is a Post-doctoral Research Fellow in the Center for IT-Security, Privacy, and Accountability (CISPA) in Saarbrücken, Germany. He received his Ph.D. in computer science from National University of Singapore (NUS) in 2013. His research interests include software analysis and testing, with a specific focus on designing efficient and secure software systems.

Abhik Roychoudhury is a Professor of Computer Science at School of Computing, National University of Singapore. He received his Ph.D. in Computer Science from the State University of New York at Stony Brook in 2000. Since 2001, he has been employed at the National University of Singapore. His research has focused on software testing and analysis, software security, and trust-worthy software construction. His research has received various awards and honors, including his appointment as ACM

Distinguished Speaker in 2013. He is currently leading the TSUNAMi center, a large 5-year long targeted research effort funded by National Research Foundation in the domain of software security. His research has been funded by various agencies and companies, including the National Research Foundation (NRF), Ministry of Education (MoE), A*STAR, Defense Research and Technology Office (DRTech), DSO National Laboratories, Microsoft, and IBM. He has authored a book on "Embedded Systems and Software Validation" published by Elsevier (Morgan Kaufmann) Systems-on-Silicon series in 2009, which has also been officially translated to Chinese by Tsinghua University Press. He has served in various capacities in the program committees and organizing committees of various conferences on software engineering including ICSE, ISSTA, FSE, and ASE. He is currently serving as an Editorial Board member of IEEE Transactions on Software Engineering (TSE).