Master in Informatics and Systems

^{m}Crash: a Framework for the Evaluation of Mobile Devices' Trustworthiness Properties

José Carlos Bregieiro Ribeiro

Departament of Informatics Engineering
School of Sciences and Technology
University of Coimbra, Portugal
October 2008

Organization:

Departament of Informatics Engineering, School of Sciences and Technology, University of Coimbra

Title:

^{m}Crash: a Framework for the Evaluation of Mobile Devices' Trustworthiness Properties

Author:

José Carlos Bregieiro Ribeiro

Supervising Teacher:

Mário Alberto Zenha-Rela

Period:

2005/2008

Abstract

Mobile devices, such as Smartphones, are being used virtually by every modern individual. Such devices are expected to work continuously and flawlessly for years, despite having been designed without criticality requirements. However, the requirements of mobility, digital identification and authentication lead to an increasing dependence of societies on the correct behaviour of these "proxies for the individual".

The Windows Mobile 5.0 release has delivered a new set of internal state monitoring services, centralized into the State and Notifications Broker. This API was designed to be used by context-aware applications, providing a comprehensive monitoring of the internal state and resources of mobile devices. We propose using this service to increase the dependability of mobile applications by showing, through a series of fault-injection campaigns, that this novel API is very effective for error propagation profiling and monitoring.

Keywords: robustness testing, dependability evaluation, State and Notifications Broker, Windows Mobile, COTS.

List of Publications

Publications originated from this Thesis:

- José Carlos Bregieiro Ribeiro and Mário Zenha-Rela. “mCrash: a Framework for the Evaluation of Mobile Devices’ Trustworthiness Properties”, in Proceedings of the Conference on Mobile and Ubiquitous Systems (CSMU 2006), pages 163–166. ISBN:972-8692-29-3. Guimarães, Portugal, June 2006.

- José Carlos Bregieiro Ribeiro, Bruno Miguel Luís, and Mário Zenha-Rela. “Error propagation monitoring on windows mobile-based devices”, in Proceedings of the Third Latin-American Symposium on Dependable Computing (LADC 2007), volume 4746/2007 of Lecture Notes in Computer Science, pages 111–122. ISBN:978-3-540-75293-6. Morelia, Mexico, September 2007.

Other Publications:

- José Carlos Bregieiro Ribeiro. “Search-Based Test Case Generation for Object-Oriented Java Software Using Strongly-Typed Genetic Programming”, in Proceedings of the GECCO ’08, pp. 1819-1822, Graduate Student Workshop. Atlanta, Georgia, USA, July 2008

- José Carlos Bregieiro Ribeiro, Mário Zenha-Rela and Francisco Fernández de Vega. “Strongly-Typed Genetic Programming and Purity Analysis: Input Domain Reduction for Evolutionary Testing Problems”, in Proceedings of the GECCO ’08, pp. 1783–1784, 10th Annual Conference on Genetic and Evolutionary Computation. Atlanta, Georgia, USA, July 2008.

- José Carlos Bregieiro Ribeiro, Mário Zenha-Rela and Francisco Fernández de Vega. “A strategy for evaluating feasible and unfeasible test cases for the evolutionary testing of object-oriented software”. in Proceedings of the 3rd International Workshop on Automation of Software

Test (AST '08), pp. 85-92, 30th International Conference on Software Engineering (ICSE '08), Leipzig, Germany, May 2008.

- José Carlos Bregieiro Ribeiro, Mário Zenha-Rela and Francisco Fernández de Vega. "An Evolutionary Approach For Performing Structural Unit-Testing On Third-Party Object-Oriented Java Software", in Proceedings of the NICSO 2007, pp. 379-388, Vol. 129/2008, Studies in Computational Intelligence, International Workshop on Nature Inspired Cooperative Strategies for Optimization. Acireale, Italy, November 2007.

- José Carlos Bregieiro Ribeiro, Mário Zenha-Rela and Francisco Fernández de Vega. "eCrash: a Framework for Performing Evolutionary Testing on Third-Party Java Components", in Proceedings of the I Jornadas sobre Algoritmos Evolutivos y Metaheuristicas (JAEM), pp. 137-144, II Congreso Español de Informática (CEDI). ISBN: 978-84-9732-593-6. Zaragoza, Spain, September 2007.

- José Carlos Bregieiro Ribeiro, Francisco Fernández de Vega and Mário Zenha-Rela. "Using Dynamic Analysis of Java Bytecode for Evolutionary Object-Oriented Unit Testing", in Proceedings of the 8th Workshop on Testing and Fault Tolerance (WTF), pp. 143-156, 25th Brazilian Symposium on Computer Networks and Distributed Systems (SBRC). ISBN: 85-766-0119-1. Belém, Brazil, May 2007.

Acknowledgements

I would like to start expressing my deepest gratitude to Professor Mário Zenha-Rela, for his availability, his support, his enthusiasm.

I would also like to thank Bruno Miguel Luis for his companionship, and Professor Francisco Fernández de Vega for his contribution to our latest research.

Finally, I also take this opportunity for thanking mum, dad, and my little brother, and for sending a big kiss to my girlfriend Marta.

Contents

List of Figures

List of Tables

List of Abbreviations

API	Application Program Interface
SNB	State and Notifications Broker
OEM	Original Equipment Manufacturer
IDE	Integrated Development Environment
MUT	Module Under Test
OO	Object-Oriented
STGP	Strongly-Typed Genetic Programming

Chapter 1

Introduction

The philosophy for mobile devices has been evolving towards the "wallet" paradigm: they contain important personal information, and virtually every adult carries one. They are true "proxies for the individual" [16]. Additionally, people are getting used to take care of their business affairs on these pervasive devices, since they are becoming increasingly more sophisticated and are able to handle most basic tasks.

However, not all mobile devices were designed with enterprise class security in mind, and even components which were specifically designed for mission-critical applications may prove to have problems if used in a different context. Retrofitting trust in any technology is considerably harder than building it in from the start, especially when users have already perceived it as invasive, intrusive, or dangerous.

Software behaviour is a combination of many factors: which particular data states are created, what paths are exercised, how long execution takes, what outputs are produced, and so forth [14]. An operating system is, itself, a dynamic entity [32], as different services have diverse robustness properties; the way in which software makes use of those services will have impact on the robustness of their operations. What's more, mobile devices – such as Pocket PCs and Smartphones – are expected to work continuously and flawlessly for years, with varying energy and in harsh environmental conditions; this requires stringent internal state and resource monitoring.

One of the major problems in dependability evaluation is the difficulty of observing what happens inside the system that is submitted to stress. This problem is exacerbated when the source code of the system under evaluation is unavailable; alas, this is the most common situation.

The Windows Mobile 5.0 release has delivered a new Application Program Interface (API) with a set of services targeting context-aware applications, the State and Notifications Broker (SNB) [33], which aims to provide comprehensive monitoring of resources. It is clear that automated testing of black-box components requires (or, at least, can be greatly improved by) built-in system support; this service, while not providing true white-box testing tools, makes the system transparent enough to allow for a semantic-oriented monitoring of relevant state-variables.

One of the key ideas presented in this thesis is to use the internal monitoring services provided by the SNB for error propagation profiling and monitoring. Although most of the information provided by the SNB could be obtained by other means, this tool enables the monitoring of a standard set of relevant system variables defined by the API itself, in a straightforward manner. We also aim to contribute to the issue of interpreting the raw data produced into useful information – i.e., into insight.

Chapter 2

Background and Related Work

We propose using Windows Mobile 5.0's SNB API for error monitoring and propagation profiling; the SNB centralizes system state in documented locations and distributes change notifications to interested parties using a publish-subscribe model. It provides built-in monitoring services to a large number of internal state variables, which constitutes a means for keeping an eye on undesirable state value modifications.

The ideas that lead to this approach were inspired by previous works of other and ourselves: the methodology for automated testing of component interfaces, based on parameter data types instead of component functionality, was inspired on Ballista [15]; the error propagation analysis follows the work of [14], by extending the degree of observability available; the fault-injection execution manager was based on the RT-Xception [10].

The following sections provide an overview of relevant concepts and resources.

2.1 Mobile Devices and Context-Awareness

Nowadays, personalization focuses on creating information services that deliver "the right information at the right time, in the right place, in the right way, to the right person" [28].

Mobile information services must be designed, specifically, for the context of their use and to meet mobile users' needs. The current design philosophy for mobile devices has been described as having the "Swiss Army knife" [29] approach: include as much functionality as possible into a single device. Popular usage scenarios include navigational assistance, task-specific cognitive assistance, access to messaging information, access to schedule information, memory aids and reminders, and meeting and experience sharing tools.

Mobile devices function in a very fluid environment, subject to frequent changes. It thus becomes important for an application to know what state the device is in at any one time. Context information allows for minimization of the number of steps required to carry out a specific action and this, in turn, depends on adapting content and application functionality. Unlike desktop systems, where functionality often takes precedence over interaction time, mobile devices' small size and limited input facility lower the chances that users will complete tasks requiring a long interaction sequence. Context-awareness aims to make interaction with devices easier.

Context can be categorized according to the following items [8]:

- Computing context – e.g., network connectivity, communication costs, and communication bandwidth, and nearby resources such as printers, displays, and workstations.
- User context – e.g., as the user's profile, location, people nearby, even the current social situation.
- Physical context – e.g., lighting, noise levels, traffic conditions, and temperature.
- Time context – e.g., time of a day, week, month, and season of the year.

Among the problems that may hinder context-aware devices and applications, Pascoe [22] identifies resource hungriness, high development cost and the diversity of computing environments as being the most common.

2.2 Dependability and Software Fault Injection

Dependability can be defined as the trustworthiness of a computing system, which allows reliance to be justifiably placed on the service it delivers [4]; it is related not only to correct behaviour under normal circumstances, but also – and most importantly – to reliability in the presence of errors.

The applications envisaged by our approach are not mission-critical – these are not the targets of the Windows Mobile platform. In fact, not all mobile devices are designed with enterprise class security in mind, and even components which were specifically designed for mission-critical applications may prove to have problems if used in a different context.

Retrofitting trust in any technology is considerably harder than building it in from the start [16], especially when users have already perceived it as invasive, intrusive, or dangerous. This work's focus is precisely on trustworthiness – i.e., on contributing for the reliable and secure behaviour of standard personal applications such as those used by mobile devices for e-commerce or personal identification. The key dependability attribute we are interested in is the robustness of software, formally defined as the degree to which a software component functions correctly in the presence of exceptional inputs or stressful environmental conditions [1].

One of the main causes of failure is the interactions of different software components in the system which are often unknown at development time, such as the specific operating system under which the application is running, the set of services available, the drivers, and other applications.

The reliability of software is tested by exercising it with a consensually agreed workload when benchmarking different systems [3]. When absolute reliability figures are required, a tailored workload is selected to perform robustness tests to evaluate the different services offered by the system. There are two main categories of software testing techniques used to pinpoint faults [15]:

- White-box, or structural, testing – useful for attaining high test coverage of programs. Typically focuses on the control flow of a program, rather than on the handling of exceptional data values.

- Black-box, or behavioural, testing – designed to demonstrate correct response to various input values regardless of the software implementation. It is more appropriate for robustness testing, and is the preferred approach whenever the source code is not available [7].

The approach we are interested in is a black-box one, as we aim to test the robustness of the software. Robustness testing does not claim to report the number of software defects; rather, it intends detect opportunities for obtaining faulty responses due to robustness problems within the software being tested.

The key idea behind black-box testing is that tests are based on the values of parameters – test cases are built according to their data types – and not on the implementation details, which may even be unavailable because access to source code is required.

There are several research works on the evaluation the robustness of operating systems, with drivers being identified as the major source of OS failures ([9, 21, 30]). The effects of driver errors were also studied in [2, 11, 14].

Such was also the goal of the work presented in [3], where bit-flips were used to emulate errors; instead, we focus on data level errors flowing through the different module interfaces. This is also the approach followed in [14].

Of particular interest is the work in [15], where the Ballista methodology for the automated testing of component interfaces is presented. Its main contribution is the proposal for an object-oriented approach based on parameter data types instead of component functionality, thus eliminating the need for function-specific test scaffolding.

Our work is in line of the work performed by [14], but extending the observability available: while in their work the error propagation analysis was limited to the interface between components, we delve deeper into the system's internal state by using the information made available by the SNB.

2.3 The State and Notifications Broker

The focus of our research is on the monitoring and profiling of system behaviour under abnormal circumstances and, in particular, error propagation. It is clear that automated testing of black box components requires – or can be greatly improved – with built-in system support.

The Windows Mobile 5.0 operating system has centralised its state information into a single entity, the State and Notifications Broker (SNB)[1] – whether that information is related to the device itself or to the standard Windows Mobile 5.0 applications. It provides a standard architecture for monitoring state values for changes and for distributing change notifications to the interested parties using a publish-subscribe model, thus making it unnecessary to hunt down a separate function or API for each individual state value. Also, prior to the introduction of the SNB API, determining a specific state value often required several function calls and additional logic.

Error monitoring and profiling based on reports from the Windows SNB can greatly reduce the difficulty of collecting the internal system state in this platform. This service, while not providing a true white-box approach, such as control flow, makes the system transparent enough to allow for a semantically-oriented monitoring of relevant state-variables.

Each state value is available either through native or managed code: native code provides direct access to the behaviours and capabilities of the platform using the C or C++ languages, but the developer is responsible for handling the details involved in interacting with the platform; managed code puts a greater focus on development productivity by encapsulating details within class libraries.

[1]http://msdn2.microsoft.com/en-us/library/aa455748.aspx [cited: 2008/10/02]

For the managed code developers, the .NET Compact framework includes more than a hundred pre-defined static base State and Notification Properties[2] (Table 2.1) that represent the available state values; in addition, Original Equipment Manufacturers (OEMs) are free to add more values, as the underlying implementation of the SNB uses the registry as the data store.

The base State and Notification Properties encompass information on the system state, phone, user, tasks and appointments, connections, messages, media player and time. To access the present value of a given property, managed-code developers simply access the `SystemState` property that corresponds to the state value of their interests: to receive state value change notifications, an application must simply create an instance of the `SystemState` class and pass the appropriate `SystemProperty` enumeration that identifies the value of interest, and attach a delegate to the new `SystemState` instance's `Changed` event.

Still, some problems persist. Firstly, there is no standard way for third-party software companies to expose their own properties in the SNB. Secondly, not all the device's properties are exposed, although registry-based custom-made states can be implemented to extend the default functionality. Thirdly, even though C# managed code is easier to use, it includes reduced functionality when compared to native C++ code.

[2]http://msdn2.microsoft.com/en-us/library/aa455750.aspx [cited: 2008/10/02]

Information Types	**Base State and Notification Properties**
System State	ActiveApplication; ActiveSyncStatus; CameraPresent; CarKitPresent; CradlePresent; DisplayRotation; HeadsetPresent; KeyboardPresent; PowerBatteryBackupState; PowerBatteryBackupStrength; PowerBatteryState; PowerBatteryStrength
Message	MessagingActiveSyncAccountName; MessagingActiveSyncEmailUnread; MessagingLastEmailAccountName; MessagingMmsAccountName; MessagingMmsUnread; MessagingOtherEmailUnread; MessagingSmsAccountName; MessagingSmsUnread; MessagingTotalEmailUnread; MessagingVoiceMail1Unread; MessagingVoiceMail2Unread; MessagingVoiceMailTotalUnread
Calendar	CalendarAppointment; CalendarAppointmentBusyStatus; CalendarAppointmentCategories; CalendarAppointmentEndTime; CalendarAppointmentHasConflict; CalendarAppointmentLocation; CalendarAppointmentStartTime; CalendarAppointmentSubject; CalendarEvent; CalendarEventBusyStatus; CalendarEventCategories; CalendarEventEndTime; CalendarEventHasConflict; CalendarEventLocation; CalendarEventStartTime; CalendarEventSubject; CalendarEventSubject; CalendarHomeScreenAppointment; CalendarHomeScreenAppointmentBusyStatus; CalendarHomeScreenAppointmentCategories; CalendarHomeScreenAppointmentEndTime; CalendarHomeScreenAppointmentHasConflict; CalendarHomeScreenAppointmentLocation; CalendarHomeScreenAppointmentStartTime; CalendarHomeScreenAppointmentSubject; CalendarNextAppointment; CalendarNextAppointmentBusyStatus; CalendarNextAppointmentCategories; CalendarNextAppointmentEnd; CalendarNextAppointmentHasConflict; CalendarNextAppointmentLocation; CalendarNextAppointmentStart; CalendarNextAppointmentSubject; TasksActive; TasksDueToday; TasksHighPriority; TasksOverdue
Media	MediaPlayerAlbumTitle; MediaPlayerTrackArtist; MediaPlayerTrackBitrate; MediaPlayerTrackGenre; MediaPlayerTrackNumber; MediaPlayerTrackTimeElapsed; MediaPlayerTrackTitle
Phone	Phone1xRttCoverage; PhoneActiveCallCount; PhoneBlockedSim; PhoneCallCalling; PhoneCellBroadcast; PhoneLine1Selected; PhoneLine2Selected; PhoneMultiLine; PhoneNoService; PhoneProfileName; PhoneSearchingForService; PhoneInvalidSim; PhoneNoSim; PhoneRadioOff; PhoneRadioPresent; PhoneRingerOff; PhoneRoaming; PhoneSignalStrength; PhoneSimFull; SpeakerPhoneActive; PhoneCallBarring; PhoneCallForwardingOnLine1; PhoneCallForwardingOnLine2; PhoneCallOnHold; PhoneCallTalkingv; PhoneConferenceCall; PhoneIncomingCall; PhoneMissedCall; PhoneMissedCalls; PhoneOperatorName; PhoneProfile;
Connections	ConnectionsBluetoothDescriptions; ConnectionsCellularCount; ConnectionsCellularDescriptions; ConnectionsCount; ConnectionsDesktopCount; ConnectionsDesktopDescriptions; ConnectionsModemCount; ConnectionsModemDescriptions; ConnectionsNetworkAdapters; ConnectionsNetworkCount; ConnectionsNetworkDescriptions; ConnectionsProxyCount; ConnectionsProxyDescriptions; ConnectionsUnknownCount; ConnectionsUnknownDescriptions; ConnectionsVpnCount; ConnectionsVpnDescriptions; PhoneActiveDataCall; PhoneGprsCoverage; PhoneHomeService
Time	Date; Time
User	OwnerEmail; OwnerName; OwnerNotes; OwnerPhoneNumber; PhoneIncomingCallerContact; honeIncomingCallerContactPropertyID; PhoneIncomingCallerContactPropertyName; PhoneIncomingCallerName; PhoneIncomingCallerNumber; PhoneLastIncomingCallerContact; PhoneLastIncomingCallerContactPropertyIDv; PhoneLastIncomingCallerContactPropertyName; PhoneLastIncomingCallerName; PhoneLastIncomingCallerNumber; PhoneTalkingCallerContact; PhoneTalkingCallerContactPropertyID; PhoneTalkingCallerContactPropertyName; PhoneTalkingCallerName; PhoneTalkingCallerNumber

Table 2.1: Base State and Notification Properties, categorized by Information Type.

Chapter 3

Methodology and Framework

In order to access the usefulness of the SNB for error propagation monitoring and profiling, we have developed a prototype general-purpose software testing tool – ^{m}Crash – that allowed us to automate the testing process. This section describes this framework and contextualizes the use of the SNB API.

Presently, ^{m}Crash allows automatic testing of classes, methods, parameters and objects in the .NET framework. In order to achieve this, several .NET framework APIs were employed, such as the `System.Reflection` and `System.CodeDom` namespaces, and the Microsoft Excel Object Library. This tool is meant to dynamically generate a test script, compile it into a .NET assembly, and invoke the test process. Many ideas of this approach were inspired by previous work of others and ourselves. This tool was presented in [23, 24], and its design closely follows the guidelines proposed by [17, 32].

Four fundamental modules embody our tool: the Faultload Database; the Input Generation and Fault Injection Module; the Postcondition Checker; and the Execution Manager. These modules are schematically represented in Figure 3.1, and will be discussed in further detail in the following subsections.

3.1 Faultload Database

The process of building the Faultload Database precedes the actual testing phase, as a set of test cases must be created for each unique public constructor, method and property of each class made available by the Module Under Test (MUT). The first step is to catalogue all the MUT's information – including input and output parameters, their data types and error codes. Most of these tasks are achieved automatically by means of the `Reflection` API; alas, some of the information (e.g., the expected return values) must be manually defined by the software tester.

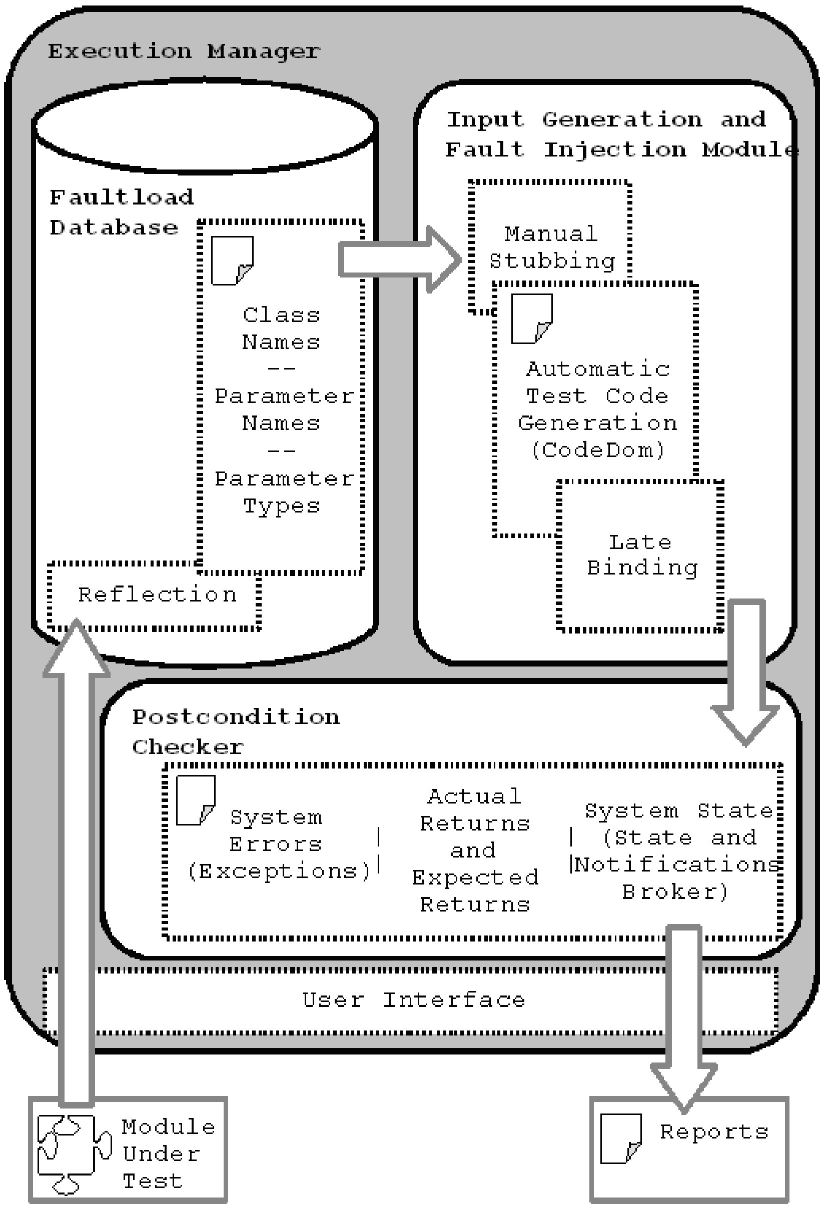

Execution Manager
Faultload Database
Class Names -- Parameter Names -- Parameter Types
Reflection
Input Generation and Fault Injection Module
Manual Stubbing
Automatic Test Code Generation (CodeDom)
Late Binding
Postcondition Checker
System Errors (Exceptions)
Actual Returns and Expected Returns
System State (State and Notifications Broker)
User Interface
Module Under Test
Reports

Figure 3.1: ^{m}Crash's framework scheme.

The following step involves performing a domain analysis for each individual data type in order to establish the faultload. Test cases encompass valid, boundary and invalid conditions for the different data types; this allows the coverage of a vast array of erroneous inputs, and also enables the tester to obtain a reference execution (i.e., the gold run).

Finally, all this information is inserted in an Excel spreadsheet – using Excel API Programming in the case of the automated tasks, and manually in the case of the values that must be defined by the software tester. This spreadsheet holds an ordered list of the API calls that will be used to test the MUT.

3.2 Input Generation and Fault Injection Module

The Input Generation component dynamically generates test cases for a given set of constructors, methods and properties; the Fault Injection component automatically executes the test cases, and collects the information returned by a particular function call.

The test cases' source code is generated using the `CodeDom` API, and is based on the parameters defined in the Excel spreadsheet during the Faultload Database building process. Additionally, the necessary code for logging any events detected by SNB is included in the test cases' source code. Any changes to a monitored property are logged to a text file. If any parameters were left blank during the Faultload Database definition, the user is given the option of either allowing the application to insert random values and "dummy" objects, or entering a "manual stub" himself. The ability to use late binding, provided by the `Reflection` API, is employed to dynamically invoke the test cases; using this technique enables the ^{m}Crash tool to resolve the existence and name of a given type and its members at runtime (rather than compile time).

In short, a reference execution is run first; then, all the boundary and invalid test cases defined for a given function are executed. The Postcondition Checker is in charge of comparing these executions and presenting reports to the user. This methodology automates the test case generation process, hence avoiding the need to write source code, and it even allows for a considerable amount of system state to be set.

3.3 Postcondition Checker

The Postcondition Checker monitors the environment for unacceptable events. Assertions are put in two main places: at the system level and at the output. All of these values are recorded in a Microsoft Excel spreadsheet.

At the system level, global environmental events are tracked using the SNB. Two distinct categories of values are logged: those incoming from the notifications received, and those of the properties being monitored – the Base State and Notification Properties. The latter are logged before and after the fault injection process takes place. At the output level, the tool validates return values (by comparing them with the expected returns defined during the Faultload Database definition) and checks if exceptions were thrown – and where they were thrown.

Finally, the results yielded by the boundary and invalid test values are automatically compared with the gold run, and any discrepancies will be recorded in the results spreadsheet.

3.4 Execution Manager

The Execution Manager provides the visual interface between the user and the software testing tool. It allows for the definition of the parameters used during a given software testing campaign, such as the location of the .NET Integrated Development Environment (IDE) and of the MUT. It is also responsible for dealing with the complexity of creating the three other modules, and for feeding each one of them with the necessary incoming data.

Until now, this tool was only tested using Microsoft Visual Studio as the IDE. During the fault injection process, the IDE is automatically started and the code produced by the Fault Injection component is executed.

At the end of the software testing campaign, the results spreadsheet, containing all the results gathered by the Postcondition Checker, is presented to the user.

Chapter 4

Experimental Studies

In the experiments described in this study, we employed the ^{m}Crash tool to conduct a software testing campaign with the purpose of assessing Windows Mobile 5.0's trustworthiness properties and uncovering faults.

4.1 Targets and Methodology

The targets of this experiment were the public properties made available by the Microsoft Windows Mobile 5 `Microsoft.WindowsMobile.PocketOutlook` namespace. We chose to target the `PocketOutlook` namespace in this study because it is a productivity package used, essentially, by programmers that develop mobile and context-aware applications, and also because its complexity is adequate for research and demonstration purposes. The rationale for focusing our study on the public properties is related with the extended insight that the SNB allows.

We started by using ^{m}Crash to extract the list of public properties available in all the classes made available by the `PocketOutlook` namespace. During the Faultload Database building process, 9 distinct classes, including 96 distinct public properties, were identified and catalogued. These 96 distinct public properties encompassed 13 different data types, including primitive data types (`bool`, `int`, `string`), enumerations (`WeekOfMonth`, `TimeSpan`, `Sensitivity`, `RecurrenceType`, `Month`, `Importance`, `DaysOfWeek`, `DateTime`, `BusyStatus`) and objects (`Uri`).

The methodology followed was that of performing fault-injection by changing the target public properties' values. Valid, boundary and invalid test values were defined for each of the data types, except for `bool` properties, to which only true or false values can be assigned. Manual stubbing was employed to instantiate an object and to set the minimum amount of state needed for each individual test case. In the majority of the cases, creating a "dummy" object sufficed but, in some situations, additional complexity was required; these special situations were individually addressed in order to create the state needed.

Preliminary experiments showed that some errors were only uncovered by the operating system when the object carrying the faulty property was used as an input parameter in a method call. In order to pinpoint such situations, we tested all of the abovementioned objects as input parameters in a method belonging to the same class.

Finally, we analysed the results collected by ^{m}Crash in order to draw conclusions. The logs generated by the Postcondition Checker were automatically compared to the previously recorded gold run; all the exceptions thrown (and the phase of the testing process in which they were thrown) were annotated; the values the properties assumed (in the cases in which no exception was thrown) after the fault injection process were compared to those that were expected. The results of this comparison were thoroughly analysed, and will be discussed in the following subsection.

4.2 Results and Observations

As a result of our experiments, we were able to categorize the exceptions thrown during the fault injection procedure in two types, according to their latency:

- if the exception is thrown during the process of assigning an erroneous value to a property (i.e. if the assertion is located in the property's setter method) the exception is considered to be an **immediate exception**;

- if the exception is thrown by the method that receives the object containing the faulty property as an input parameter (i.e. the assertion is located in the method called) the exception is considered to be a **late exception**.

Late exceptions are more problematic, due to the high probability of error propagation. In fact, objects containing "faulty" properties could linger in the system indefinitely, until they are used as an input parameter and the exception is triggered. Late exception statistics are depicted in Table 4.1.

The vast majority of the test values that threw late exceptions were of the `string` data type; the property can be assigned an invalid value, but

Late Exceptions	
Data Types	**Test Cases**
string	string with 4096 characters; "\\\0066n"; string.Empty; null
DateTime	DateTime.MaxValue
EmailMessage.Importance	(Importance)1000; (Importance)(-1); Importance.Low; Importance.High
EmailMessage.Sensitivity	(Sensitivity)int.MaxValue; (Sensitivity)(-1); Sensitivity.Confidential; 0

Exception types	**Ocurrences**
System.ComponentModel.Win32Exception	60
System.InvalidCastException	17

Table 4.1: Data Types and corresponding Test Cases that threw late exceptions. Late Exception Types and corresponding number of occurrences.

when the object is used as an input in a method an assertion existed to make sure that the `string` could not exceed the maximum length.

Actually, the maximum length of `string` objects is defined in the documentation, but nothing is mentioned on when the check is made. What's more, this limit is documented in the property's entry; hence the programmer has no reason to assume that the check won't be done immediately. The `DateTime` data type is also problematic in terms of latency; the `DateTime.MaxValue` test value (which we considered to be a boundary value) often generated a late exception. Such was also the case of some of the enumeration types associated to the `EmailMessage` class.

Immediate exceptions included `null`, range and format exceptions. Table 4.2 resumes the data for these categories of exceptions. The analysis of the exceptions' data does not allow the typification the data types according to category of exception generated – there is no coherent behaviour or pattern that allows us to conclude that a particular data type or a particular test case always have the same exception latency. Similar invalid test values generate both immediate and late exceptions, which can only be explained by the API's internal structure (of which no source code is available).

It is at this point that the extended insight provided by the SNB can prove to be invaluable; this API can be used to monitor properties continuously. The software tester will thus be able to assert properties' values all the way through – and early on – the software testing process.

Immediate Exceptions	
Data Types	**Test Cases**
string	String with 4096 characters
DateTime	DateTime.MaxValue; DateTime.MinValue; new DateTime(int.MaxValue, int.MaxValue, int.MaxValue); new DateTime(int.MinValue, int.MinValue, int.MinValue);
TimeSpan	TimeSpan.MaxValue; new TimeSpan(int.MaxValue, int.MaxValue, int.MaxValue)
Uri	new Uri(null); new Uri("dei.uc.pt")
EmailMessage.Importance	(Importance)1000; (Importance)(-1)
EmailMessage.Sensitivity	(Sensitivity)int.MaxValue; (Sensitivity)(-1)
Appointment.BusyStatus	(BusyStatus)(-1)

Exception types	**Ocurrences**
System.ArgumentOutOfRangeException	23
System.ComponentModel.Win32Exception	16
System.UriFormatException	1
System.NullReferenceException	1
System.ArgumentNullException	1

Table 4.2: Data Types and corresponding Test Cases that threw immediate exceptions. Immediate Exception Types and corresponding number of occurrences.

With this in mind, we devoted special attention to the time frame between the contamination of the property with an erroneous value and the usage of the "faulty" object as an input parameter in a method (error latency). The measurements made to the `Appointment` class were especially interesting, since the SNB monitors an extensive set of properties regarding Calendar information.

For instance, we observed that when the `Appointment.Start` property was set to a value below the allowed range, an immediate "Argument Out Of Range" exception was thrown; nevertheless the Postcondition Checker received a notification of the property being set to its lower bound – i.e. some of the properties values are changed even though an exception is thrown. What's more, in a similar situation – when the `Appointment.Start` property was set to a value above its upper bound – an immediate exception of the type `System.ComponentModel.Win32Exception` was thrown, and the property kept its previous value. This irregular behaviour requires distinct handling of similar situations.

Other anomalous behaviour observed using the SNB included receiving notifications of changes to properties other than those directly disturbed. The following observations are typical of this situation:

- when the `Appointment.Start` property was set to an invalid value, the `Appointment.End` property was set to its default value;
- when the `Appointment.End` property was set to an invalid value the `Appointment.Start` property was set to its default value.

Although this behaviour is not completely unreasonable – the `Start` and `End` properties of the `Appointment` class are obviously related – it does constitute a means for error propagation. It also provides a clear sign that to increase the effectiveness of the postcondition checking the system must me monitored as a whole.

In some circumstances, we were also able to detect the contamination of objects before the errors were detected by the runtime environment. For instance, in the `Appointment.Subject` property, the "String with 4096 characters" boundary test case (the documentation explicitly refers that an appointment's subject is limited to 4096 characters) generated a late exception when the object was used as an argument in a method call. Nevertheless, by means of the State and Notification Broker, it was possible to observe that this property assumed a `null` value immediately after the erroneous value was assigned to the property; it issued a notification for the change of the base State and Notification Property `CalendarAppointmentSubject`, and the logs also showed that the property was reset to `null` – its default value.

It must be stressed that this anomalous behaviour was unveiled by the SNB – it published a notification of the property change – before the runtime environment threw an exception.

Chapter 5

Conclusions

This thesis proposes using a custom-tailored framework for accessing Windows Mobile 5.0's trustworthiness properties. To achieve this, we employed the SNB API for error monitoring and propagation profiling, and presented an experimental study illustrating the feasibility of the approach.

The SNB centralizes system state information in documented locations, and distributes change notifications to interested parties using a publish-subscribe model. It provides built-in monitoring services to internal system variables, which constitutes a means for keeping an eye on undesirable state value modifications.

The experimental observations show that system built-in assertions are sparsely distributed and less than thoroughly documented, and that errors can remain dormant in the system until they are detected and dealt with – e.g., with an exception. This behaviour renders the SNB particularly useful for detecting erroneous internal states. Interesting observations include:

- receiving notifications of changes to properties other than those disturbed;
- receiving notification of a property being changed, even though an exception was immediately thrown after an invalid value was assigned to it;
- receiving notification of invalid values being assigned to a property; an exception was only triggered when the faulty property's instance was used as an argument in a method call.

Even thought this API is not enough to prevent the contamination of internal objects with erroneous values, we believe it represents an opportunity for enhancing dependability in large-scale, not limited to mission-critical applications.

Our work so far was limited to the base State and Notification Properties defined by default; nevertheless, these are clearly insufficient to cover the system as a whole. Future work includes extending the set of properties exposed, with the purpose of broadening the range of relevant system variables being monitored by our tool.

Along this work, we realized that the current fault-injection paradigm is still much too centred on the stimulus-response functional model. However, a growing number of real-world mission-critical applications are now based on the object-oriented model; nonetheless, tools for dependability evaluation are seldom used in this context.

5.1 On-Going and Future Work

Recently, the focus of our research has been mostly focused on employing Evolutionary Algorithms for the structural unit-testing of Object-Oriented (OO) programs. In fact, a large amount of the resources spent on testing are applied on the difficult and time consuming task of locating quality test data [6]; automating this process is thus vital to advance the state-of-the-art in software testing. So far, automation in this area has been quite limited, mainly because the exhaustive enumeration of a program's input is unfeasible for any reasonably-sized program, and random methods are unlikely to exercise "deeper" features of software [19].

Meta-heuristic search techniques, like Evolutionary Algorithms (high-level frameworks which utilise heuristics, inspired by genetics and natural selection, in order to find solutions to combinatorial problems at a reasonable computational cost [5]), are natural candidates to address this problem, since the input space is typically large but well defined, and test goal can usually be expressed as a fitness function [12]. The application of Evolutionary Algorithms to test data generation is often referred to as *Evolutionary Testing* [31] or *Search Based Testing* [19].

Our Evolutionary Testing approach involves representing and evolving test cases using the Strongly-Typed Genetic Programming (STGP) technique [20]. The methodology for evaluating the quality of test cases includes instrumenting the program under test, and executing it using the generated test cases as inputs with the intention of collecting trace information with which to derive coverage metrics. The aim is that of efficiently guiding the search process towards achieving full structural coverage of the program under test. These concepts have been implemented into the *eCrash* automated test case generation tool [26].

Our main goals are those of defining strategies for addressing the challenges posed by the OO paradigm and of proposing methodologies for enhancing the efficiency of search-based testing approaches. Relevant contributions presented so far include the following:

- presenting a strategy for test case evaluation and search guidance, which involves allowing unfeasible test cases (i.e., those that terminate prematurely due to a runtime exception) to be considered at certain stages of the evolutionary search – namely, once the feasible test cases that are being bred cease to be interesting [25];

- introducing a novel Input Domain Reduction methodology, based on the concept of Purity Analysis, which allows the identification and removal of entries that are irrelevant to the search problem because they do not contribute to the definition of test scenarios [27].

Evolutionary Testing is an emerging methodology for automatically generating high quality test data. It is, however, a difficult subject, especially if the aim is to implement an automated solution, viable with a reasonable amount of computational effort, which is adaptable to a wide range of test objects. Significant success has been achieved by applying this technique to the automatic generation of unit-test cases for procedural software [18, 19]. The application of search-based strategies for OO unit-testing is, however, fairly recent [31] and is yet to be investigated comprehensively [13].

Bibliography

[1] Ieee standard glossary of software engineering terminology. Technical report, 1990.

[2] Arnaud Albinet, Jean Arlat, and Jean-Charles Fabre. Characterization of the impact of faulty drivers on the robustness of the linux kernel. In *DSN*, pages 867–876, 2004.

[3] Jean Arlat, Jean-Charles Fabre, Manuel Rodríguez, and Frédéric Salles. Dependability of cots microkernel-based systems. *IEEE Trans. Comput.*, 51(2):138–163, 2002.

[4] A. Avizienis, J. Laprie, and B. Randell. Fundamental concepts of dependability, 2001.

[5] Thomas Back, David B. Fogel, and Zbigniew Michalewicz, editors. *Handbook of Evolutionary Computation.* IOP Publishing Ltd., Bristol, UK, UK, 1997.

[6] Boris Beizer. *Software Testing Techniques.* John Wiley & Sons, Inc., New York, NY, USA, 1990.

[7] Boris Beizer. *Black-box testing: techniques for functional testing of software and systems.* John Wiley & Sons, Inc., New York, NY, USA, 1995.

[8] Guanling Chen and David Kotz. A survey of context-aware mobile computing research. Technical report, Hanover, NH, USA, 2000.

[9] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. *SIGOPS Oper. Syst. Rev.*, 35(5):73–88, 2001.

[10] João Carlos Cunha, Ricardo Maia, Mário Zenha Rela, and João Gabriel Silva. A study of failure models in feedback control systems. In *DSN '01: Proceedings of the 2001 International Conference on Dependable*

Systems and Networks (formerly: FTCS), pages 314–326, Washington, DC, USA, 2001. IEEE Computer Society.

[11] João Durães and Henrique Madeira. Multidimensional characterization of the impact of faulty drivers on the operating systems behavior. *IEICE transactions on information and systems*, 86(12):2563–2570, 20031201.

[12] Mark Harman. Automated test data generation using search based software engineering. In *AST '07: Proceedings of the Second International Workshop on Automation of Software Test*, page 2, Washington, DC, USA, 2007. IEEE Computer Society.

[13] Mark Harman. The current state and future of search based software engineering. In *FOSE '07: 2007 Future of Software Engineering*, pages 342–357, Washington, DC, USA, 2007. IEEE Computer Society.

[14] Andreas Johansson. Error propagation profiling of operating systems. In *DSN '05: Proceedings of the 2005 International Conference on Dependable Systems and Networks*, pages 86–95, Washington, DC, USA, 2005. IEEE Computer Society.

[15] Nathan P. Kropp, Philip J. Koopman Jr., and Daniel P. Siewiorek. Automated robustness testing of off-the-shelf software components. In *Symposium on Fault-Tolerant Computing*, pages 230–239, 1998.

[16] Marc Langheinrich. Privacy by Design – Principles of Privacy-Aware Ubiquitous Systems. In G. D. Abowd, B. Brumitt, and S. Shafer, editors, *Ubicomp 2001 Proceedings*, volume 2201 of *Lecture Notes in Computer Science*, pages 273–291. Springer, 2001.

[17] Kanglin Li and Mengqi Wu. *Effective Software Test Automation: Developing an Automated Software Testing Tool.* SYBEX Inc., Alameda, CA, USA, 2004.

[18] Timo Mantere and Jarmo T. Alander. Evolutionary software engineering, a review. *Appl. Soft Comput.*, 5(3):315–331, 2005.

[19] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, 2004.

[20] David J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230, 1995.

[21] B. Murphy and Levidow B. Windows 2000 dependability. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks*, New York, NY, USA, 2000.

[22] Jason Pascoe, Nick Ryan, and David Morse. Issues in developing context-aware computing. In *HUC '99: Proceedings of the 1st international symposium on Handheld and Ubiquitous Computing*, pages 208–221, London, UK, 1999. Springer-Verlag.

[23] José Carlos Bregieiro Ribeiro, Bruno Miguel Luís, and Mário Zenha-Rela. Error propagation monitoring on windows mobile-based devices. In *LADC 2007: Third Latin-American Symposium on Dependable Computing*, volume 4746/2007 of *Lecture Notes in Computer Science*, pages 111–122. Springer Berlin / Heidelberg, 2007.

[24] José Carlos Bregieiro Ribeiro and Mário Zenha Rela. mcrash: a framework for the evaluation of mobile devices' trustworthiness properties. In *CSMU 2006: Conference on Mobile and Ubiquitous Systems.*

[25] José Carlos Bregieiro Ribeiro, Mário Zenha Rela, and Francisco Fernandéz de Vega. A strategy for evaluating feasible and unfeasible test cases for the evolutionary testing of object-oriented software. In *AST '08: Proceedings of the 3rd international workshop on Automation of software test*, pages 85–92, New York, NY, USA, 2008. ACM.

[26] José Carlos Bregieiro Ribeiro, Mário Zenha-Rela, and Francisco Fernandez de Vega. An evolutionary approach for performing structural unit-testing on third-party object-oriented java software. In *Nature Inspired Cooperative Strategies for Optimization (NICSO 2007)*, volume Volume 129/2008 of *Studies in Computational Intelligence*, pages 379–388. Springer Berlin / Heidelberg, 11 2007.

[27] José Carlos Bregieiro Ribeiro, Mário Alberto Zenha-Rela, and Francisco Fernandéz de Vega. Strongly-typed genetic programming and purity analysis: input domain reduction for evolutionary testing problems. In *GECCO '08: Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation*, pages 1783–1784, New York, NY, USA, 7 2008. ACM.

[28] George Roussos, Andy J. Marsh, and Stavroula Maglavera. Enabling pervasive computing with smart phones. *IEEE Pervasive Computing*, 4(2):20–27, 2005.

[29] M. Satyanarayanan. Swiss army knife or wallet? *IEEE Pervasive Computing*, 4(2):2–3, 2005.

[30] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. *ACM Trans. Comput. Syst.*, 23(1):77–110, 2005.

[31] Paolo Tonella. Evolutionary testing of classes. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 119–128, New York, NY, USA, 2004. ACM Press.

[32] Jeffrey M. Voas and Gary McGraw. *Software fault injection: inoculating programs against errors.* John Wiley & Sons, Inc., New York, NY, USA, 1997.

[33] Jim Wilson. The state and notifications broker part i. Electronic Article, February 2006. http://msdn2.microsoft.com/en-us/library/aa456240.aspx.

Appendix A

Publications

A.1 “mCrash: a Framework for the Evaluation of Mobile Devices’ Trustworthiness Properties” (CSMU 2006)

^{m}Crash: a Framework for the Evaluation of Mobile Devices' Trustworthiness Properties

José Ribeiro[1], Mário Zenha-Rela[2]

[1] Department of Informatics Engineering, Polytechnic Institute of Leiria,
2411-901 Leiria
jose.ribeiro@estg.ipleiria.pt
[2] Department of Informatics Engineering, University of Coimbra,
3030-290 Coimbra
mzrela@dei.uc.pt

Abstract. A rationale and framework for the evaluation of mobile devices' robustness and trustworthiness properties using a Windows Mobile 5.0 testbed is presented. The methodology followed includes employing software fault-injection techniques at the operating system's interface level and customising tests to the behaviour of the software.

1 Introduction

The philosophy for mobile devices has been evolving towards the "wallet" [1] paradigm: they contain important personal information, and virtually every adult carries one. They are true "proxies for the individual" [2]. Additionally, people are getting used to take care of their business affairs on these pervasive devices, since they are becoming increasingly more sophisticated and are able to handle most basic tasks. But not all mobile devices were designed with enterprise class security in mind, and even components which were specifically designed for mission-critical applications may prove to have problems if used in a different context. Retrofitting trust in any technology is considerably harder than building it in from the start [3], especially when users have already perceived it as invasive, intrusive, or dangerous. More emphasis should be placed on building robust systems that can adapt to aberrant behaviour.

However, software behaviour is a combination of many factors: which particular data states are created, what paths are exercised, how long execution takes, what outputs are produced, and so forth [4]. An operating system is, itself, a dynamic entity [5], as different services have diverse robustness properties; the way in which software makes use of those services will have impact on the robustness of their operations.

We believe that a tool for assessing robustness properties during the development phase by employing software fault-injection techniques will contribute decisively for the trustworthiness of the resulting software, as it is known that most of the computing devices' breakdowns are caused by residual software faults.

This framework’s approach is that of integrating a fault-injection tool into the IDE – e.g. Microsoft Visual Studio 2005 – as a plug-in. The availability of the code under development allows for a profiling phase to take place, hence creating ground for making use of software fault-injection techniques to test the code in a tailored way and according to the developers’ requirements. The methodology we will be following is that of applying fault injection instrumentation directly at the Windows Mobile 5.0 API - it is at the operating system’s interfaces that corrupt data coming from the application under development will be simulated.
The next chapter outlines the proposed framework and provides an overview of its architecture; the third and last chapter presents topics for discussion and sets ground for future work.

2 Description of the Proposed Framework

Four fundamental modules embody this framework:

- The Faultload Database.
- The Input Generation and Fault Injection Module.
- The Postcondition Checker.
- The Execution Manager.

The Faultload Database

The process of building the Faultload database is executed offline, and must precede the actual testing phase, as a set of test values must be created for each unique parameter data type of every function made available by the Windows Mobile 5.0 SDK API. Test cases encompass both valid and exceptional values for the parameter data types, in order to mimic all sorts of events.
The first step is to catalogue all the API information, including all the functions, input and output parameters and their data types, and error codes. A parsing application that extracts information from the Windows Mobile SDK documentation automates this process.
The following step includes performing a domain analysis for each individual parameter in order to establish the Faultload. The DWORD data type, for example, is a typedef for an unsigned long. Therefore, test values could include its boundaries, and some randomly selected valid and invalid values. For pointer types, values such as NULL, -1 (cast to a pointer), pointer to freed memory, and pointers to malloc’ed buffers of various powers of two could be used [6].

Input Generation and Fault Injection Module

The Input Generator Component dynamically generates test cases for a given set of functions and their parameter data types. The function lists are fed to this component

by the Execution Manager as a result of a profiling phase; test cases are built by drawing values from the pre-defined Faultload Database.
The Fault Injection Component's approach is that of automatically and exhaustively testing combinations of parameter test cases by nested iteration. The testing code is generated given just the function name and a listing of parameter types. A test object is instantiated for each of the function's parameters data types, and is responsible for creating all testing infrastructure; the constructor runs the instructions needed to generate the test case. This methodology uses test objects to encapsulate all the test case generation complexity - hence avoiding the need to adapt test cases to a particular function - and allows for a considerable amount of system state to be set.

Postcondition Checker

The Postcondition Checker monitors the environment for unacceptable events. Assertions are put in two main places: the system level and the output level. The results yielded by the testing phase will be mapped to low-level categories (similar to those depicted in the C.R.A.S.H. Scale [7]).

Table 1. C.R.A.S.H. scale.

Value	Description
Catastrophic	the system crashes or hangs
Restart	the test process hangs
Abort	the test process terminates abnormally
Silent	the test process exits without an error code, but one should have been returned
Hindering	the test process exits with an error code not relevant to the situation
Pass	the module exits properly, possibly with an appropriate error code

Additionally, at the system level, global environmental events will be tracked using the State and Notifications Broker API [8] - Windows Mobile 5.0's state information store, which provides a standard architecture for monitoring state values for changes - which is a valuable tool for increasing the system's observability and for detecting actions that were uncalled for.
At the output level, Silent and Hindering failures values will be considered; this is why it is necessary - during the APIs cataloguing phase - for the output parameters and for the error codes to be identified.

Execution Manager

The Execution Manager is responsible for profiling the code - in order to correctly setup the Input Generation and Fault Injection Module with adequate usage scenarios - and for automating the collection and analysis of vulnerability information provided by the Postcondition Checker.

The profiling phase aims to identify the operating system's services used by the application under development – by means of an API tracing tool [9] - in order to generate ordered lists of the API functions; once the software behaviour is analysed, it is possible to make use of software fault-injection techniques to test the code in a customized way. This phase is of paramount importance, as the number of test cases is determined by the number and type of input parameters, and is thus exponential with the number of functions.

The collection and analysis phase allows for the outputs monitored by the Post-condition Checker to be properly logged and mapped to the inputs previously produced by the Input Generation and Fault Injection Module.

3 Conclusions and Future Work

We believe that the presented framework will allow the detection robustness problems all the way through – and early on – the software development process, hopefully decreasing the need for adding software "wrappers" to baselined software, and thus reducing the possibility of introducing additional faults and lowering maintenance costs. However, several issues are still subject to discussion. Topics include:

- The feasibility of including all of Windows Mobile 5.0's APIs in the scope of the proposed framework.
- The relevance of implementing a parsing application to extract information from the Windows Mobile SDK documentation.
- The methodology for the generation of test cases.
- The tools to be employed during the profiling phase.
- The drawbacks and advantages of covering Silent and Hindering failures.

References

1. Satyanarayanan, M.: Swiss Army Knife or Wallet?. IEEE Pervasive Computing, vol. 4, number 2, pp. 2-3 (2005)
2. Abowd, D.: The Smart Phone: A First Platform for Pervasive Computing. IEEE Pervasive-Computing, vol. 4, number 2, pp. 18-19 (2005)
3. Langheinrich, M.: Privacy by Design - Principles of Privacy-Aware Ubiquitous Systems. Swiss Federal Institute of Technology, ETH Zurich, ch. 3. ACM UbiComp (2001)
4. Johansson, A., Suri, N.: Error Propagation Profiling of Operating Systems. International Conference on Dependable Systems and Networks (2005)
5. Voas, V., McGraw, G.: Software Fault Injection: Inoculating Programs Against Errors. John Wiley & Sons, Inc. (1998)
6. Kropp, N., Koopman, P., Siewiorek, D.: Automated Robustness Testing of Off-the-Shelf Software Components. Fault Tolerant Computing Symposium, Munich (1998)
7. Biyani, M., Santhanam, P.: TOFU: Test Optimizer for Functional Usage. Software Engineering Technical Brief (1997)
8. Wilson. J.: The State and Notifications Broker Part I. MSDN Library Online (2006)
9. Durães, J., Madeira, H.: Generic Faultloads Based on Software Faults for Dependability Benchmarking. International Conference on Dependable Systems and Networks (2004)

A.2 "Error propagation monitoring on windows mobile-based devices" (LADC 2007)

Error Propagation Monitoring on Windows Mobile-based Devices

José Carlos Bregieiro Ribeiro[1], Bruno Miguel Luís[2], Mário Zenha-Rela[2]

[1] Polytechnic Institute of Leiria (IPL), Morro do Lena, Alto do Vieiro,
Leiria, Portugal
jose.ribeiro@estg.ipleiria.pt
[2] University of Coimbra (UC), CISUC, DEI, 3030-290,
Coimbra, Portugal
mzrela@dei.uc.pt

Abstract. Mobile devices, such as Smartphones, are being used virtually by every modern individual. Such devices are expected to work continuously and flawlessly for years, despite having been designed without criticality requirements. However, the requirements of mobility, digital identification and authentication lead to an increasing dependence of societies on the correct behaviour of these 'proxies for the individual'. The Windows Mobile 5.0 release has delivered a new set of internal state monitoring services, centralized into the State and Notifications Broker. This API was designed to be used by context-aware applications, providing a comprehensive monitoring of the internal state and resources of mobile devices. In this paper we propose using this service to increase the dependability of mobile applications by showing, through a series of fault-injection campaigns, that this novel API is very effective for error propagation profiling and monitoring.

Keywords: Robustness Testing, Dependability Evaluation, State and Notifications Broker, Windows Mobile, COTS.

1 Introduction

The philosophy for mobile devices has been evolving towards the ‘wallet’ paradigm: they contain important personal information, and virtually every adult carries one. They are true “proxies for the individual” [1]. Additionally, people are getting used to take care of their business affairs on these pervasive devices, since they are becoming increasingly more sophisticated and are able to handle most basic tasks. But not all mobile devices were designed with enterprise class security in mind, and even components which were specifically designed for mission-critical applications may prove to have problems if used in a different context. Retrofitting trust in any technology is considerably harder than building it in from the start [1], especially when users have already perceived it as invasive, intrusive, or dangerous.

Software behaviour is a combination of many factors: which particular data states are created, what paths are exercised, how long execution takes, what outputs are produced, and so forth [2]. An operating system is, itself, a dynamic entity [3], as different services have diverse robustness properties; the way in which software makes use of those services will have impact on the robustness of their operations. What’s more, mobile devices – such as Pocket PCs and Smartphones – are expected to work continuously and flawlessly for years, with varying energy and in harsh environmental conditions; this requires stringent internal state and resource monitoring. One of the major problems in dependability evaluation is the difficulty of observing what happens inside the system that is submitted to stress. This problem is exacerbated when the source code of the system under evaluation is unavailable; alas, this is the most common situation.

The Windows Mobile 5.0 release has delivered a new API with a set of services targeting context-aware applications, the State and Notifications Broker (SNB) [4], which aims to provide comprehensive monitoring of resources. This service, while not providing true white-box testing tools, makes the system transparent enough to allow for a semantically-oriented monitoring of relevant state-variables.

One of the key ideas presented in this paper is to use the internal monitoring services provided by the State and Notifications Broker for error propagation profiling and monitoring. Although most of the information provided by the State and Notifications Broker could be obtained by other means, this tool enables the monitoring of a standard set of relevant system variables defined by the API itself, in a straightforward manner. We also aim to contribute to the issue of interpreting the raw data produced into useful information, into insight. It is clear that automated testing of black-box components requires (or, at least, can be greatly improved by) built-in system support.

2 Background

Computer dependability can be defined as the trustworthiness of a computing system, which allows reliance to be justifiably placed on the service it delivers [5]. The applications envisaged by our approach, however, are not mission-critical – actually, this is not the target of the Windows Mobile platform. This work’s focus is trustworthiness – i.e. reliable and secure behaviour of standard personal applications – such as those used by mobile devices for e-commerce or personal identification. In fact, the key dependability attribute we are interested in is the robustness of software, formally defined as the degree to which a software component functions correctly in the presence of exceptional inputs or stressful environmental conditions [6]. The robustness of software is tested by exercising it with a tailored workload. Black-box or behavioural testing [7] is the preferred approach whenever the source code is not available – as is the case of a proprietary operating system. There are several research works on the evaluation of the robustness of operating systems [8-13]. Drivers were identified as a major source of OS failures, and its effects were studied in [2, 14, 15].

The works based on the Ballista methodology [16-18] interested us particularly, due to the possibility of automating the testing of component interfaces. Its main contribution was the proposal of an object-oriented approach based on parameter data types instead of component functionality, thus eliminating the need for function-specific test scaffolding. Since we are emulating software errors, we focus on data level errors flowing through the different module interfaces and on the evaluation of the impact of these errors on the overall system dependability. This is also the approach followed in [2, 19-21]; however, in [20] the study of the impact of data errors is focused on the consequences of error propagation in control applications. The experiments presented in this paper closely follow the line of the work presented in [2] by extending the observability; while in their work the error propagation analysis is limited by the observation at the interface between components, we delve deeper into the system internals, as this was made feasible by the State and Notifications Broker of the Windows Mobile 5.0 platform. Johansson and Suri's work has the added interest in that they present a case study based on Windows CE.net, the platform from which Windows Mobile (our testbed) derives.

Thus, the main focus of this paper is on presenting, employing, and discussing the usefulness of this service to increase the dependability of mobile applications by showing, through a series of fault-injection campaigns, that this novel API is very effective for error propagation profiling and monitoring. The rationale behind this study falls into the 'callee interface fault injection' as defined in [22].

3 State and Notifications Broker Overview

The recent Windows Mobile 5.0 operating system has centralised its state information into a single entity, the State and Notifications Broker[1] – whether that information is related to the device itself or to the standard Windows Mobile 5.0 applications. It provides a standard architecture for monitoring state values for changes and for distributing change notifications to the interested parties using a publish-subscribe model, thus making it unnecessary to hunt down a separate function or API for each individual state value. Also, prior to the introduction of the State and Notifications Broker API, determining a specific state value often required several function calls and additional logic.

Each state value is available either through native or managed code: native code provides direct access to the behaviours and capabilities of the platform using the C or C++ language, but the developer is responsible for handling the details involved in interacting with the platform; managed code puts a greater focus on development productivity by encapsulating details within class libraries. For the managed code developers, the .NET Compact framework includes more than a hundred pre-defined static *base State and Notification Properties*[2] that represent the available state values; in addition, original equipment manufacturers (OEMs) are free to add more values, as the underlying implementation of the State and Notifications Broker uses the registry as the data store. The base State and Notification Properties encompass information on the system state, phone, user, tasks and appointments, connections, messages, media player and time. To access the present value of a given property, managed-code developers simply access the `SystemState` property that corresponds to the state value of their interests: to receive state value change notifications, an application must simply create an instance of the `SystemState` class and pass the appropriate

[1] http://msdn2.microsoft.com/en-us/library/aa455748.aspx [cited: 2007/03/03]

[2] http://msdn2.microsoft.com/en-us/library/aa455750.aspx [cited: 2007/04/03]

`SystemProperty` enumeration that identifies the value of interest, and attach a delegate to the new `SystemState` instance's `Changed` event.

Still, some problems persist. Firstly, there is no standard way for third-party software companies to expose their own properties in the State and Notifications Broker. Secondly, not all the device's properties are exposed, although registry-based custom-made states can be implemented to extend the default functionality. Thirdly, even though C# managed code is easier to use, it includes reduced functionality when compared to native C++ code.

4 Framework Description

In order to access the usefulness of the State and Notifications Broker for error propagation monitoring and profiling, we've developed a prototype general-purpose software testing tool – ^{m}Crash – that allowed us to automate the testing process. This section describes this framework and contextualizes the use of the State and Notifications Broker API.

Presently, ^{m}Crash allows automatic testing of classes, methods, parameters and objects in the .NET framework. In order to achieve this, several .NET framework APIs were employed, such as the `System.Reflection` and `System.CodeDom` namespaces, and the Microsoft Excel Object Library. This tool is meant to dynamically generate a test script, compile it into a .NET assembly, and invoke the test process. Many ideas of this approach were inspired by previous work of others and ourselves. This tool was first presented in [23], and its design closely follows the guidelines proposed by [3, 24].

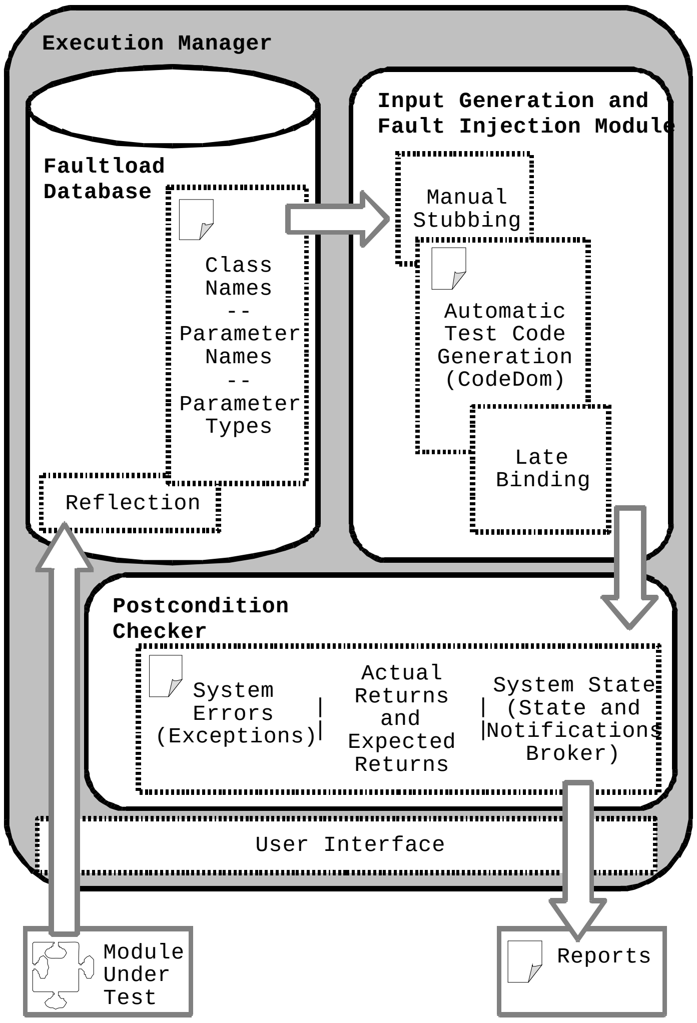
Four fundamental modules embody our tool: the Faultload Database; the Input Generation and Fault Injection Module; the Postcondition Checker; and the Execution Manager. These modules are schematically represented in Figure 1, and will be discussed in further detail in the following subsections.

4.1 Faultload Database

The process of building the Faultload Database precedes the actual testing phase, as a set of test cases must be created for each unique public constructor, method and property of each class made available by the Module Under Test (MUT). The first step is to catalogue all the MUTs information – including input and output parameters, their data types and error codes. Most of these tasks are achieved automatically by means of the Reflection API; alas, some of the information (e.g. the expected return values) must be manually defined by the software tester.

The following step involves performing a domain analysis for each individual data type in order to establish the faultload. Test cases encompass valid, boundary and invalid conditions for the different data types; this allows the coverage of a vast array of erroneous inputs, and also enables the tester to obtain a reference execution (i.e. the *gold run*).

Finally, all this information is inserted in an Excel spreadsheet – using Excel API Programming in the case of the automated tasks, and manually in the case of the values that must be defined by the software tester. This spreadsheet holds an ordered list of the API calls that will be used to test the MUT.



Execution Manager
Faultload Database
Class Names -- Parameter Names -- Parameter Types
Reflection
Input Generation and Fault Injection Module
Manual Stubbing
Automatic Test Code Generation (CodeDom)
Late Binding
Postcondition Checker
System Errors (Exceptions)
Actual Returns and Expected Returns
System State (State and Notifications Broker)
User Interface
Module Under Test
Reports

Fig. 1. Framework scheme.

4.2 Input Generation and Fault Injection Module

The Input Generation component dynamically generates test cases for a given set of constructors, methods and properties; the Fault Injection component automatically executes the test cases, and collects the information returned by a particular function call.

The test cases' source code is generated using the `CodeDom` API, and is based on the parameters defined in the Excel spreadsheet during the Faultload Database building process. Additionally, the necessary code for logging any events detected by State and Notifications Broker is included in the test cases' source code. Any changes to a monitored property are logged to a text file. If any parameters were left blank during the Faultload Database definition, the user is given the option of either allowing the application to insert random values and "dummy" objects, or entering a "manual stub" himself. The ability to use late binding, provided by the Reflection API, is employed to dynamically invoke the test cases; using this technique enables the ^{m}Crash tool to resolve the existence and name of a given type and its members at runtime (rather than compile time).

In short, a reference execution is run first; then, all the boundary and invalid test cases defined for a given function are executed. The Postcondition Checker is in charge of comparing these executions and presenting reports to the user. This methodology automates the test case generation process, hence avoiding the need to write source code, and it even allows for a considerable amount of system state to be set.

4.3 Postcondition Checker

The Postcondition Checker monitors the environment for unacceptable events. Assertions are put in two main places: at the system level and at the output. All of these values are recorded in a Microsoft Excel spreadsheet.

At the system level, global environmental events are tracked using the State and Notifications Broker. Two distinct categories of values are logged: those incoming from the notifications received, and those of the properties being monitored – the Base State and Notification Properties. The latter are logged before and after the fault injection process takes place. At the output level, the tool validates return values (by comparing them with the expected returns defined during the Faultload Database definition) and checks if exceptions were thrown – and where they were thrown.

Finally, the results yielded by the boundary and invalid test values are automatically compared with the gold run, and any discrepancies will be inserted in the results spreadsheet.

4.4 Execution Manager

The Execution Manager provides the visual interface between the user and the software testing tool. It allows for the definition of the parameters used during a given software testing campaign, such as the location of the .NET IDE and of the MUT. It is also responsible for dealing with the complexity of creating the three other modules, and for feeding each one of them with the necessary incoming data.

Until now, this tool was only tested using Microsoft Visual Studio 2005 as the IDE. During the fault injection process, the IDE is automatically started and the code produced by the Fault Injection component is executed.

At the end of the software testing campaign, the results spreadsheet, containing all the results gathered by the Postcondition Checker, is presented to the user.

5 Experimental Observations

In the experiment described in this paper, we employed the ^{m}Crash tool to conduct a software testing campaign with the purpose of accessing Windows Mobile 5.0's trustworthiness properties and uncovering faults.

5.1 Targets and Methodology

The targets of this experiment were the public properties made available by the Microsoft Windows Mobile 5 `Microsoft.WindowsMobile.PocketOutlook` namespace. We chose to target the `PocketOutlook` namespace in this study because it is a productivity package used, essentially, by programmers that develop mobile and context-aware applications, and also because its complexity is adequate for research and demonstration purposes. The rationale for focusing our study on the public properties is related with the extended insight that the State and Notifications Broker allows..

We started by using ^{m}Crash to extract the list of public properties available in all the classes made available by the `PocketOutlook` namespace. During the Faultload Database building process, 9 distinct classes, including 96 distinct public properties, were identified and catalogued. These 96 distinct public properties encompassed 13 different data types, including primitive data types (`bool`, `int`, `string`), enumerations (`WeekOfMonth`, `TimeSpan`, `Sensitivity`, `RecurrenceType`, `Month`, `Importance`, `DaysOfWeek`, `DateTime`, `BusyStatus`) and objects (`Uri`).

The methodology fallowed was that of performing fault-injection by changing the target public properties' values. Valid, boundary and invalid test values were defined for each of the data types, except for `bool` properties, to which only true or false values can be assigned. Manual stubbing was employed to instantiate an object and to set the minimum amount of state needed for each individual test case. In the majority of the cases, creating a "dummy" object sufficed but, in some situations, additional complexity was required; these special situations were individually addressed in order to create the state needed.

Preliminary experiments showed that some errors were only uncovered by the operating system when the object carrying the faulty property was used as an input parameter in a method call. In order to pinpoint such situations, we tested all of the abovementioned objects as input parameters in a method belonging to the same class.

Finally, we analysed the results collected by ^{m}Crash in order to draw conclusions. The logs generated by the Postcondition Checker were automatically compared to the previously recorded *gold run*; all the exceptions thrown (and the phase of the testing process in which they were thrown) were annotated; the values the properties assumed (in the cases in which no exception was thrown) after the fault injection process were compared to those that were expected. The results of this comparison were thoroughly analysed, and will be discussed in the following subsection.

5.2 Results and Observations

As a result of our experiments, we were able to categorize the exceptions thrown during the fault injection procedure in two types, according to their *latency*:

- if the exception is thrown during the process of assigning an erroneous value to a property (i.e. if the assertion is located in the property's setter method) the exception is considered to be *immediate*;
- if the exception is thrown by the method that receives the object containing the faulty property as an input parameter (i.e. the assertion is located in the method called) the exception is considered to be *late*.

Late exceptions are more problematic, due to the high probability of error propagation. In fact, objects containing "faulty" properties could linger in the system indefinitely, until they are used as an input parameter and the exception is triggered. Late exception statistics are depicted in Table 1.

Table 1. Data Types and corresponding Test Cases that threw late exceptions. Late Exception Types and corresponding number of occurrences

Late Exceptions	
Data Types	Test Cases
`string`	`string with 4096 characters; "\\\u0066\n"; string.Empty; null`
`DateTime`	`DateTime.MaxValue`
`EmailMessage.Importance`	`(Importance)1000; (Importance)(-1); Importance.Low; Importance.High`
`EmailMessage.Sensitivity`	`(Sensitivity)int.MaxValue; (Sensitivity)(-1); Sensitivity.Confidential; 0`

Exception types	Ocurrences
`System.ComponentModel.Win32Exception`	`60`
`System.InvalidCastException`	`17`

The vast majority of the test values that threw late exceptions were of the string data type; the property can be assigned an invalid value, but when the object is used as an input in a method an assertion existed to make sure that the string could not exceed the maximum length. Actually, the maximum length of these strings is defined in the documentation, but nothing is mentioned on when the check is made. What's more, this limit is documented in the property's entry; hence the programmer has no reason to assume that the check won't be done immediately. The `DateTime` data type is also problematic in terms of latency; the `DateTime.MaxValue` test value (which we considered to be a boundary value) often generated a late exception. Such was also the case of some of the enumeration types associated to the `EmailMessage` class.

Immediate exceptions included null, range and format exceptions. Table 2 resumes the data for these categories of exceptions. The analysis of the exceptions' data doesn't allow us to typify the data types according to category of exception generated – there is no coherent behaviour or pattern that allows us to conclude that a particular data type or a particular test case always have the same exception latency. Similar invalid test values generate both immediate and late exceptions, which can only be explained by the API's internal structure (of which no source code is available).

Table 2. Data Types and corresponding Test Cases that threw immediate exceptions. Immediate Exception Types and corresponding number of occurrences.

Immediate Exceptions	
Data Types	Test Cases
`string`	`String with 4096 characters`
`DateTime`	`DateTime.MaxValue; DateTime.MinValue; new DateTime(int.MaxValue, int.MaxValue, int.MaxValue); new DateTime(int.MinValue, int.MinValue, int.MinValue)`
`TimeSpan`	`TimeSpan.MaxValue; new TimeSpan(int.MaxValue, int.MaxValue, int.MaxValue)`
`Uri`	`new Uri(null); new Uri("dei.uc.pt")`
`EmailMessage.Importance`	`(Importance)1000; (Importance)(-1)`
`EmailMessage.Sensitivity`	`(Sensitivity)int.MaxValue; (Sensitivity)(-1)`
`Appointment.BusyStatus`	`(BusyStatus)(-1)`

Exception types	Ocurrences
`System.ArgumentOutOfRangeException`	23
`System.ComponentModel.Win32Exception`	16
`System.UriFormatException`	1
`System.NullReferenceException`	1
`System.ArgumentNullException`	1

It is at this point that the extended insight provided by the State and Notifications Broker can prove to be invaluable; this API can be used to monitor properties continuously. The software tester will thus be able to assert properties' values all the way through – and early on – the software testing process.

With this in mind, we devoted special attention to the time frame between the contamination of the property with an erroneous value and the usage of the "faulty" object as an input parameter in a method (error latency). The measurements made to the `Appointment` class were especially interesting, since the State and Notifications Broker monitors an extensive set of properties regarding Task and Appointment information. For instance, we observed that when the `Appointment.Start` property was set to a value below the allowed range, an immediate "Argument Out Of Range" exception was thrown; nevertheless the Postcondition Checker received a notification of the property being set to its lower bound – i.e. some of the properties values are changed even though an exception is thrown. What's more, in a similar situation – when the `Appointment.Start` property was set to a value above its upper bound – an immediate exception of the type `System.ComponentModel.Win32Exception` was thrown, and the property kept its previous value. This irregular behaviour requires distinct handling of similar situations.

Other anomalous behaviour observed using the State and Notifications Broker included receiving notifications of changes to properties other than those directly disturbed. The following observations are typical of this situation:

- when the `Appointment.Start` property was set to an invalid value, the `Appointment.End` property was set to its default value;
- when the `Appointment.End` property was set to an invalid value the `Appointment.Start` property was set to its default value.

Although this behaviour is not completely unreasonable – the `Start` and `End` properties of the `Appointment` class are obviously related – it does constitute a means for error propagation. It also provides a clear sign that to increase the effectiveness of the postcondition checking the system must me monitored as a whole. In some circumstances, we were also able to detect the contamination of objects before the errors were detected by the runtime environment. For instance, in the `Appointment.Subject` property, the "String with 4096 characters" boundary test case (the documentation explicitly refers that an appointment's subject is limited to 4096 characters) generated a late exception when the object was used as an argument in a method call. Nevertheless, by means of the State and Notification Broker, it was possible to observe that this property assumed a null value immediately after the erroneous value was assigned to the property; it issued a notification for the change of the base State and Notification Property `CalendarAppointmentSubject`, and the logs also showed that the property was reset to null – its default value.

It must be stressed that this anomalous behaviour was unveiled by the State and Notifications Broker – it published a notification of the property change – before the runtime environment threw an exception.

6 Conclusions and Future Work

This paper proposes using a custom-tailored framework for accessing Windows Mobile 5.0's trustworthiness properties. For this, we employed the State and

Notifications Broker API for error monitoring and propagation profiling, and presented an experimental study illustrating the feasibility of the approach.

The State and Notifications Broker centralizes system state information in documented locations, and distributes change notifications to interested parties using a publish-subscribe model. It provides built-in monitoring services to internal system variables, which constitutes a means for keeping an eye on undesirable state value modifications.

The experimental observations show that system built-in assertions are sparsely distributed and less than thoroughly documented, and that errors can remain dormant in the system until they are detected and dealt with e.g. by throwing an exception. This behaviour renders the State and Notifications Broker particularly useful for detecting erroneous internal states. Interesting observations include:

- receiving notifications of changes to properties other than those disturbed;
- receiving notification of a property being changed, even though an exception was immediately thrown after an invalid value was assigned to it;
- receiving notification of invalid values being assigned to a property; an exception was only triggered when the faulty property's instance was used as an argument in a method call.

Even thought this API is not enough to prevent the contamination of internal objects with erroneous values, we believe it represents an opportunity for enhancing dependability in large-scale, not limited to mission-critical applications.

Our work so far was limited to the base State and Notification Properties defined by default; nevertheless, these are clearly insufficient to cover the system as a whole. Future work includes extending the set of properties exposed, with the purpose of broadening the range of relevant system variables being monitored by our tool.

Along this work, we realized that the current fault-injection paradigm is still much too centred on the stimulus-response functional model. However, a growing number of real-world mission-critical applications are now based on the object-oriented model; nonetheless, tools for dependability evaluation are seldom used in this context.

References

1. M. Langheinrich, "Privacy by Design - Principles of Privacy-Aware Ubiquitous Systems.," presented at ACM UbiComp, 2001.
2. A. Johansson and N. Suri, "Error Propagation Profiling of Operating Systems," presented at DSN, 2005.
3. J. M. Voas and G. McGraw, Software fault injection: inoculating programs against errors. New York: Wiley Computer Pub, 1998.
4. J. Wilson, "The State and Notifications Broker Part I," MSDN Library, 2006.
5. A. Avizienis, J.-C. Laprie, and B. Randell, "Fundamental Concepts of Dependability," LAAS-CNRS N01145, 2001.
6. IEEE Standard Glossary of Software Engineering Terminology (IEEE Std610.12-1990), 1990.
7. B. Beizer, Black-box testing : techniques for functional testing of software and systems. New York ; Chichester: Wiley, 1995.
8. W. N. Gu, Z. Kalbarczyk, R. K. Lyer, and Z. Y. Yang, "Characterization of Linux kernel behavior under errors," presented at DSN, 2003.

9. B. Murphy and B. Levidow, "Windows 2000 Dependability," presented at Workshop on Dependable Networks and OS, 2000.
10. A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An Empirical Study of Operating System Errors," presented at SOSP, 2001.
11. J. Arlat, J.-C. Fabre, M. Rodriguez, and F. Salles, "Dependability of COTS Microkernel-Based Systems," IEEE Trans. on Computers, vol. 51, pp. 138–163, 2002.
12. M. M. Swift, B. N. Bershad, and H. M. Levy, "Improving the Reliability of Commodity OS's," Operating Systems Review, vol. 37, pp. 207-222, 2003.
13. X. Jun, K. Zbigniew, and K. I. Ravishankar, Networked Windows NT System Field Failure Data Analysis, 1999.
14. A. Albinet, J. Arlat, and J.-C. Fabre, "Characterization of the Impact of Faulty Drivers on the Robustness of the Linux Kernel," presented at DSN, 2004.
15. J. Durães and H. Madeira, "Multidimensional Characterization of the Impact of Faulty Drivers on the OS Behavior," IEICE, pp. 2563–2570, 2003.
16. N. P. Kropp, P. J. Koopman, and D. P. Siewiorek, "Automated Robustness Testing of Off the Shelf Software Components," presented at FTCS 98, IEEE, 1998.
17. P. Koopman and J. DeVale, "Comparing the robustness of POSIX operating systems," presented at FTCS 99, 1999.
18. C. P. Shelton, P. Koopman, and K. Devale, "Robustness testing of the Microsoft Win32 API," presented at DSN, 2000.
19. M. Hiller, A. Jhumka, and N. Suri, "PROPANE: An environment for examining the propagation of errors in software," Proceedings of the ACM SIGSOFT 2002 International Symposium on Software Testing and Analysis, pp. 81, 2002.
20. Ö. Askerdal, M. Gafvert, M. Hiller, and N. Suri, "Analyzing the Impact of Data Errors in Safety-Critical Control Systems," presented at IEEE Trans. Inf. Syst., 2003.
21. M. Hiller, A. Jhumka, and N. Suri, "EPIC: Profiling the propagation and effect of data errors in software," IEEE Trans. on Computers, vol. 53, pp. 512-530, 2004.
22. P. Koopman, "What's Wrong With Fault Injection As A Benchmarking Tool?," presented at DSN, Washington, 2002.
23. J. Ribeiro and M. Z. -Rela, "mCrash: a Framework for the Evaluation of Mobile Devices' Trustworthiness Properties," presented at CMUS, Portugal, 2006.
24. K. Li and M. Wu, Effective software test automation: developing an automated software testing tool. London: Sybex, 2004.