

Context-Driven Testing and Model-Checking for Embedded Context-Aware and Adaptive Systems

Michele Sama

David S. Rosenblum and Cecilia Mascolo (Supervisors)
Dept. of Computer Science
University College London
London, UK
m.sama@cs.ucl.ac.uk
<http://www.cs.ucl.ac.uk/people/M.Sama.html>

ABSTRACT

The increasing context-awareness and adaptivity of embedded software systems expose them to faulty adaptation caused by erroneous context's interpretations. This thesis address erroneous adaptations by considering sequences of context variations as driving input. The two suggested solution address the problem of how to identify faulty sequences and how to expose the faults that they are causing.

Categories and Subject Descriptors

D.2.4 [Software/Program Verification]: Model checking; D.2.5 [Testing and Debugging]: Symbolic execution; I.5.1 [Pattern Recognition]: Models—*Faults model*

General Terms

Verification

Keywords: adaptation, context-awareness, mobile computing, model-based analysis, testing, ubiquitous computing

1. INTRODUCTION

Smart phones are used in multiple situations in which their usability depends from their capability to speed up users' interaction by adapting to the surrounding environment and by predicting users' needs. As a consequence mobile applications become *context-aware* and *adaptive*.

This thesis addresses the problem of faulty/incorrect adaptations caused by erroneous context's interpretations in *Context Aware and Adaptive Applications (CAAA)*. Once identified, a faulty adaptation, is hardly reproducible as it depends on multiple, fast going streams of context. Moreover faulty adaptations do not imply the presence of a bug, but more likely an interference between different components, which to be prevented may require changes in the application's logic.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ISSA Seattle, WA, July 20-24 2008.

The two suggested approaches address this new category of fault by analyzing dependencies between the code and perceived representation of the surrounding environment. The process consists of three stages: 1) identification of context variables, 2) analysis of commutation's sequences and their impact on the code and 3) reproduction of faults at run-time. With a context-driven model based analysis which checks all the possible variation of context in all the possible configuration of the CAAA it is possible to expose faulty sequences. Such sequences can be reproduced with a context injection (in a context-emulator or with Mock objects on real a device) capable of emulating sequential context variations [19]. The novelty of this work is in considering sequences of context commutations to predict bugs and in using them to reduce the test suite. By flagging context variables in stage (1) and by checking them against a model in stage (2) testers identify a set of faulty sequences small enough to be reproduced at run-time and which can be used by developers to prevent and fix faults.

This context-driven approaches can be compared against other techniques in terms of detected faults, evaluation time. In terms of exposed fault it will be possible to perform a cross comparison between explored contexts and their effectiveness.

2. SOFTWARE CHALLENGES

Issues in Adaptive Applications: adaptive applications modify their behavior according to a set of rules which can be implicit in the code, as a sequence of conditions and branches, or explicit as in a rule-based system. After a trigger event or when a specific instruction is reached, conditions are evaluated and the application eventually adapts. Faults in these systems are caused by mistakes in rules' definition or by interferences between different rules. Testing rule-based systems focus on covering single rules or rule chains [7, 2]. This adequacy criteria can be used to detect which part of the adaptation has been covered during a test and also which condition would allow to covered the missing parts.

The correctness of adaptive systems can be tested with a pseudo oracle able providing a twin implementation of the rule-base [4], or can be verified by analyzing the automaton [10] or the Finite State Machine *FSM* model [6] of the system. Tse et al. suggest a formal language to create a constraints between input and output and they apply meta-morphic testing on those constraint in order to find out how

the program deal with that [16]. Specifying a detailed formal description for the system means to design a twin implementation of the system itself with a different grammar and it is not feasible in terms of required efforts from the developers. Moreover a formal description can become very complicated and it can even contain bugs.

The coverage with a pseudo oracle is limited by the ability of understanding and reproducing all relevant inputs. FSM validation suffers from scalability problems on the number of variables. These kind of systems also present the non trivial problems of: *how long will it takes to adapt*; and *which behavior will the system have during the adaptation*.

Issues in Context Aware Applications: context-aware applications provide information of a specific context to the user or to other components. Context's detection implies a blocking system call to a sensor or to an internal device. These queries (e.g. "What is the amount of free memory in that disk?", or "Which networks are available?") stop the execution while waiting for the result. The response time can also be a problem, which is usually solved by dedicating a thread for context handling. Systems depending from context information also suffer from hardware failures or noises in the read data.

Issues in Context Aware Adaptive Applications: applications which are both context-aware and adaptive, adapt their behaviour according to the surrounding environment. All the context variables which the rule-based adaptation logic requires to make a decision are constantly refreshed by dedicated threads. Each context variables has a different inquiry time with which it will be refreshed and which will affect the liveness of the sensed value.

In addition to faults related to the rule's definition and to noises in read values, these systems suffer from wrong adaptations as a result of a misinterpreted read of the environment. In the first place, since there are multiple threads updating the sensed context with different refresh rates, it is not clear when to evaluate the collected information [15]. Moreover multiple Context Variables collected asynchronously can have discordant values which generates wrong or invalid inferred contexts. This is similar to the design issue in self-timing circuit design issues in which outputs are the result of the evaluation of a chain of gate, are similar to asynchronous context variable refresh within multiple boolean adaptation predicates. Unger [17] and Hauck [8] detected timing problems in sequential circuits. Similarly faults in CAAAs can be identified from a software point of view as a result of asynchronous context updates.

For instance consider an handset which is using a bi-axial accelerometer to adapt the display between portrait and landscape mode according to the device's orientation. Regarding the implementation this adaptive feature can be too sensitive and be affected by noises, or can react too slowly and lose some adaptation. Assuming readings correct, if the handset is rotated of 270 degrees, the application is supposed to change orientation twice. This can hypothetically cause side effects if the multiple rotations are not handled properly or if the second rotation is started when the initial one is still being performed.

Traditional code-based adequacy criteria, such as code, statement or branch coverage when applied to CAAAs have a weaker meaning and leave most of the faults uncaught. A modern approach consist in performing a code-analysis in order to define variables or instructions which are dependent

from the context Context Aware Program Points ("CAPP") and their execution paths ("Driver") [12, 18]. Valid adequacy criteria are then taking in account concurrent executions of specific sets of drivers. However the number of drivers, in order to cover multiple combinations of several CAPPs, badly scale. Extracting CAPPs from the code, and preempt their execution in order to cover all the possible drivers is also not trivial. With the same principle also a set of adequacy criteria has been defined by focusing on Context Variables by assuming that they will be accessed by multiple threads and by testing if they can be correctly defined and accessed [12], or if specific Context Aware Program Points (CAPP) [18] could be preempted correctly.

3. RESEARCH HYPOTHESIS

The hypothesis underlying this thesis is that context-driven model-based analysis, testing methods and adequacy criteria are more effective in exposing adaptation faults in context-aware and adaptive systems than other general purpose or issue-specific techniques in terms of detected faults and execution time. In other terms separating the problem of analysis/testing of context-based adaptations from general analysis/testing allows to address the problem more effectively.

Faulty adaptation are caused by a discrepancy between the real environment and its representations (sensed context) on which the application bases the decision whether to adapt or not [15, 14]. This inconsistency is produced by interferences between multiple threads storing different context variables asynchronously and the decision-maker controlling adaptations. In the execution flow of CAAAs, an adaptation is the direct consequence of a context variation. By using contexts as inputs and adaptations as outputs it is possible to expose faulty adaptations and to isolate their causes. This context-driven approach is more effective in detecting faulty adaptations rather than other general purpose approaches because it underlines dependencies between sensed contexts and inferred adaptations [15]. By (1) identifying context variable and by (2) analyzing their impact on the execution flow with a specific model it is possible to predict weaknesses in the implementation and it will be possible to (3) address them with a reduced test suite containing only test cases selected/generated during the analysis.

Moreover, as explained in Section 2, CAAAs have a common structure which can be modelled with a FSM by mapping each configuration with a state in which an event-driven rule-based systems is using context information to choose if adapting. The cause of a faulty adaptation can be modelled as well into a pattern of fault. Once a set of pattern has been identified it is possible to check the SUT's model against it [14] and to expose for each state sequence of context variation which may lead to one of those patterns.

Once a faulty combination has been exposed it can be reproduced at run-time by emulating or mocking the context. The capability of reproducing those faults which have been identified on the model is useful to remove false positives and to fix real faults. From the pattern analysis it is also possible to create more robust architectures capable of avoiding/preventing faults.

This thesis can be proved by comparing context-driven approaches against other techniques. The validity of these approaches can be proved directly by detecting a set of known faults. The effectiveness can be evaluated by comparing both the model based context driven verification and the

context emulation with classical techniques in terms of false negatives, false positives, evaluation time and implementation time. They can also be compared to each other by comparing result from the model based analysis of the whole context space against the emulation of multiple sets of context variations. In term of efficiency (detected faults per execution time) these approaches can be compared against the work of Lu [12] and Wang [18] which search for similar faults but at a lower level and with a more instruction-based analysis. Moreover this work focus on sequences of commutation which are selected from an initial stage of analysis and only a subset is reproduced.

4. APPROACHES

The most important aspect of context-aware testing is the capacity to expose and to reproduce a fault with the minimal impact in the original implementation. In Section 4.1 this problem is addressed by executing the SUT into a controlled environment. Even if effective in reproducing faults this approach requires to execute the SUT, which is unfeasible for large context-spaces. Ideally it should be executed directly with contexts producing faults. This can be obtained by exploring the context space searching for contexts causing malfunctions. In Section 4.2 the proposed validation generates sets of context variations which failed to respect given properties against a model of the SUT. Those faulty set are a good candidate as input for the execution into a controlled environment. In Section 4.2 two main classes of faults have been identified. These are related to the model used. In future studies more detailed models will be applied to expose additional classes of fault. Moreover the two suggested approaches, even if meant to be executed iteratively, are fully independent, therefore they could be replaced with future works.

4.1 Context Emulation

Faulty adaptations are nasty to be detected because they do not interrupt directly the normal computation-flow, they just evolve the system incorrectly. Since they happen as a result of a sequence of context variations an effective way to detect them is to drive tests with different sequences of variations as input. This *context-driven testing* can be performed by executing the CAAA under test by emulating/mocking context information, which can be used also used by a partial pseudo oracle to verify the correctness of each performed adaptation. From the same context-oriented prospective coverage criteria can be expressed in terms of reached behaviors, triggered adaptations and emulated context variations.

The validity of this approach depends from how effective is the emulated context in exposing faults. A random set of contexts can be easily generated according to the nature of each context variable but it does not guarantee any kind of coverage, and may require to be applied for a very long time. Moreover randomly generated contexts can violate physical constraints and produce false positives. To avoid this issue it is also possible to use traces directly collected by sensors. Traces are expensive in terms of time and effort to be collected, but also more effective in reproducing real conditions. The acquisition problem can be solved by using a synthetic trace generator as the one proposed by Calegari which is capable of generating realistic values from the analysis of a set of real ones [3].

Unfortunately this approach, regardless from how context is generated, has some limitations. First of all, traces synchronization is not trivial, especially if the system is using multiple sensors it is not easy to emulate a consistent environment. Moreover reaching coverage criteria such as *all the contexts*, which tries to cover all the combination of context variables, or *all the adaptations*, which tries to cover all the possible adaptations, require several time to be achieved.

4.2 Context-Driven Validation

System's evolution can be analyzed and predicted with a model. We used a finite state machine (FSM) in which behaviors are mapped as states and adaptations as edges (*Adaptation-FSM* or *A-FSM*) [14]. An adaptation can be represented as $(state, predicate) \rightarrow state$ in which the predicate is a compositions of boolean conditions based on context variables. In each state it is possible to predict how the SUT will react to a context variation according to the conditions which are composing its adaptation predicates.

From an analysis of manually detected faults it has been possible to identify common patterns and to design their formal representation in the A-FSM as a set of properties which must be respected. Context variations violating one or more of those properties are defined *faulty against the pattern*. This model based approach is extensible in terms of detectable faults: testers can isolate new patterns and detect them during the analysis. So far two different families of faults have been detected: (1) faults causing unwanted/unexpected behaviors generated by interferences between multiple rules within a certain set of states with certain inputs, and (2) faults related to inconsistencies in the internal representation of the context caused by multiple commutation of variables which are loaded asynchronously.

As all the model checking techniques, this approach suffers from scalability due to the explosion of the number of possible configurations which is 2^b where b is the number of boolean conditions on context variables. A first solution to this problem is using a discretization mechanism in which continuous values of context variable can be discretized by considering boolean values of predicates in order to reduce the number of combination from continuous to be exponential on the number of boolean conditions. This is similar to the clock in time automata but extended to all the context variables [1, 9, 5, 13, 11]. A further reduction can be obtained by a symbolic analysis and by the application of a set of constraints which can strongly reduce the number of possible combinations of inputs. Constraints can be found by analyzing physical and mathematical properties of context variables and by reducing the possible combination of boolean conditions using those variables. In the applied case study 25% of the possible combination of boolean inputs can be pruned by constraints.

For each specific fault, the number of combination can be reduced by determining exactly which set of variables need to be explored and which can be pruned. This approach, together with the constraint reduction, applied to a set of 5 different algorithms to detect pattern of fault that we have identified so far, produced, in a case study with 9 states and 12 variables (with a total of $9 * 2^{12} = 36864$ different combinations of inputs) a reduction of the input space between 80% to 99% depending on the nature of each fault.

5. FUTURE EVALUATIONS

Both *Context emulation* and *Context-driven validation* are trying to expose context dependent faults, but at different levels of abstraction. Therefore their correctness can be evaluated by checking if their output is consistent. Given a CAAA, all the faults matching known patterns are easily detected in its model, but not other faults, errors and exceptions. The same CAAA can also be executed for a fixed amount of time, or in order to reach one of the adequacy criteria defined in Section 4.1 with mocked/emulated context generated: (1) randomly, (2) from real traces or (3) from synthetic traces. These three context generation methods can be compared in terms of detected fault, generation times and false positives.

It is also possible a comparison with other testing techniques. For instance a comparison against unit testing is possible by reaching one of the adequacy criteria defined in Section 4.1 and by comparing the number of detected faults. Moreover this work can be considered an high-level extension of the work of Lu [12] and Wang [18]. Wang, in particular, suggests to trigger all the possible preemption before accessing a context variable. This execution "driver" can be considered the code-level representation of how the application reacts to a context variation. However each single sequence of context variations correspond to multiple drivers, according to the number of CAPPs which the sequence is triggering and to the number of accessing threads. It will be interesting to evaluate if this set of driver can be subsumed in order to reach a relation 1:1 between an high-level context sequence and a low-level driver, and to evaluate how many fault will be undetected by doing this subsumption.

6. REMAINING WORK

Even if we assume the system's logic (configurations and rules) to be correct, because validated with the model and possible faults related to the implementation detected, because part of the code has been automatically generated from the model or because the whole system has been tested against an emulated context, the application can still fail. This can happen due to architectural problems mainly related to carriers and manufacturer. The robustness of the implementation to different devices should also be tested.

In terms of context dependencies manufacturers are implementing drivers and API following specifications, each implementation has its own characteristics, in particular where specifications are not detailed enough. In time critical operations, such as context variable refresh, each device will require a different calibration.

7. REFERENCES

- [1] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, April 1994.
- [2] V. Barr. Applications of rule-base coverage measures to expert system evaluation. In *National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference*, pages 411 – 416, July 1997.
- [3] R. Calegari, M. Musolesi, F. Raimondi, and C. Mascolo. CTG: a connectivity trace generator for testing the performance of opportunistic mobile systems. In *ESEC-FSE*, pages 415–424, New York, NY, USA, 2007. ACM.
- [4] M. D. Davis and E. J. Weyuker. Pseudo-oracles for non-testable programs. In *ACM 81: Proceedings of the ACM '81 conference*, pages 254–257, New York, NY, USA, 1981. ACM.
- [5] A. En-Nouaary, R. Dssouli, and F. Khendek. Timed Wp-method: Testing real-time systems. *IEEE Transactions on Software Engineering*, 28(11):1023–1038, November 2002.
- [6] W. F. *Modeling Software with Finite State Machines: A Practical Approach*. Auerbach Publications, 2006.
- [7] U. G. Gupta. Automatic tools for testing expert systems. *Communications of the ACM*, 5:179–184, May 1998.
- [8] S. Hauck. Asynchronous design methodologies: An overview. *Proceedings of the IEEE*, 83:69 – 93, January 1995.
- [9] T. Higashino, A. Nakata, K. Taniguchi, and A. R. Cavalli. Generating test cases for a timed I/O automaton model. In *International Workshop on Testing Communicating Systems: Method and Applications*, pages 197–214, September 1999.
- [10] J. E. Hopcroft, R. Motwani, and J. D. Ullman. Introduction to automata theory, languages, and computation, 2nd edition. *SIGACT News*, 32(1):60–65, 2001.
- [11] M. Krichen and S. Tripakis. Black-box conformance testing for real-time systems. *Model Checking Software*, 2989:109–126, February 2004.
- [12] H. Lu, W. K. Chan, and T. H. Tse. Testing context-aware middleware-centric programs: a data flow approach and an RFID-based experimentation. In *International Symposium on Foundations of Software Engineering*, pages 242–252, November 2006.
- [13] B. Nielsen and A. Skou. Automated test generation from timed automata. *International Journal on Software Tools for Technology Transfer*, 5(1):59–77, November 2003.
- [14] M. Sama, D. S. Rosenblum, Z. Wang, and S. Elbaum. Model-based fault detection in context-aware adaptive applications. In *The 16th International Symposium on the Foundations of Software Engineering (FSE 16)*, November 2008. To appear.
- [15] M. Sama, D. S. Rosenblum, Z. Wang, and S. Elbaum. Multi-layer faults in the architectures of mobile context-aware adaptive applications. In *The ICSE International Workshop on Software Architectures and Mobility*, May 2008.
- [16] T. Tse, S. Yau, W. Chan, H. Lu, and T. Chen. Testing context-sensitive middleware-based software applications. In *COMPSAC*, volume 1, pages 458–465, Los Alamitos, California, 2004. IEEE Computer Society Press.
- [17] S. H. Unger. Hazards, critical races, and metastability. *IEEE Transactions on Computers*, 44(6):754–768, June 1995.
- [18] Z. Wang, S. Elbaum, and D. S. Rosenblum. Automated generation of context-aware tests. In *International Conference on Software Engineering*, pages 406–415, May 2007.
- [19] Wikipedia. http://en.wikipedia.org/wiki/mock_object.