Context-Driven Methodologies for Context-Aware and Adaptive Systems

Michele Sama

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

of the

University College London.

Department of Computer Science

University College London

July 12, 2011

"We can't solve problems by using the same kind of thinking we used when we created them."

Albert Einstein

Abstract

Applications which are both context-aware and adapting, enhance users' experience by anticipating their need in relation with their environment and adapt their behavior according to environmental changes. Being by definition both context-aware and adaptive these applications suffer both from faults related to their context-awareness and to their adaptive nature plus from a novel variety of faults originated by the combination of the two. This research work analyzes, classifies, detects, and reports faults belonging to this novel class aiming to improve the robustness of these Context-Aware Adaptive Applications (CAAAs).

To better understand the peculiar dynamics driving the CAAAs adaptation mechanism a general high-level architectural model has been designed. This architectural model clearly depicts the stream of information coming from sensors and being computed all the way to the adaptation mechanism. The model identifies a stack of common components representing increasing abstractions of the context and their general interconnections. Known faults involving context data can be re-examined according to this architecture and can be classified in terms of the component in which they are happening and in terms of their abstraction from the environment. Resulting from this classification is a CAAA-oriented fault taxonomy.

Our architectural model also underlines that there is a common evolutionary path for CAAAs and shows the importance of the adaptation logic. Indeed most of the adaptation failures are caused by invalid interpretations of the context by the adaptation logic. To prevent such faults we defined a model, the Adaptation Finite-State Machine (A-FSM), describing how the application adapts in response to changes in the context. The A-FSM model is a powerful instrument which allows developers to focus in those context-aware and adaptive aspects in which faults reside.

In this model we have identified a set of patterns of faults representing the most common faults in this application domain. Such faults are represented as violation of given properties in the A-FSM. We have created four techniques to detect such faults. Our proposed algorithms are based on three different technologies: enumerative, symbolic and goal planning. Such techniques compensate each other. We have evaluated them by comparing them to each other using both crafted models and models extracted from existing commercial and free applications. In the evaluation we observe the validity, the readability of the reported faults, the scalability and their behavior in limited memory environments. We conclude this Thesis by suggesting possible extensions.

Acknowledgements

These almost four years at the University College of London, starting from my first enrollment in November 2006 up to now, have been a very good and intense period of my live and have changed me forever. UCL made me grooving from the young student I was four years ago to the engineer and entrepreneur that I am now. In these four years I have met many great people. Talking with them, working with them and also having fun with them has been a privilege. Among them there few which I would like to thank personally.

First of all I want to thank my supervisor, David S. Rosenblum. I probably will not be here writing this acknowledgments if it was not for his lead. There is no need for me to mention how brilliant he is as a scientist and as a researcher. Most important, in all these years I have seen David not only as my supervisor but also as a devoted friend.

Sebastian Elbaum, with whom I had the privilege of writing many papers, is one of the brightest person I have ever met. Over the years I have deeply appreciated his critics and his comments about my work. He strongly contributed in improving my research.

Franco Raimondi, great researcher, amazing friend and now superb father. When Franco is around you the sun is always shining. Franco really charges you with his positive energy and with his love for living.

I want to give a special thank to my wife, Panteha Saeedi. When I have met Panteha she was also a PhD student at UCL. Over the years she has always been on my side, helping me, giving me advices and supporting me.

I want to thank all of my fellow (and former) PhD students, which now are spread around the globe. Andy Maule, Ilias Leontiadis, Dimitris Moustakas, Leonardo Mostarda, Afra Mashadi, Clovis Chapman, Elisa Rondini, Ettore Ferranti, Camillo Fitzgerald, Behzad Bezadan, Christian Wallenta, Mohammed Ahmed, Genaina Nunes Rodriguez, Leticia Duboc, Xueni (Sylvia) Pan.

During my PhD I had the opportunity of doing two internships in Google UK ltd. under the supervision of Julian Harty. To him I want to give a very special thank. He is the most devoted and caring person that I have ever met. If more people were like him the world would be a better place to live.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The growing popularity of handheld devices such as cellphones, PDAs and portable consoles, and the increasing availability of infrastructures that support mobility such as GPS satellites, WiFi networks and Bluetooth services, together create a market for new kinds of applications that constantly monitor and react to contextual information. Users start relying to portable and wearable devices in an increasing number of situations. Applications running on such portable devices need to be properly configured, to be able to react to different environmental inputs and to proactively adapt. For example, among the top ten awardees of Google's Android Developer Challenge conducted in 2008, five applications are heavily influenced by their environment: one application relies on ambient noise and on location obtained from WiFi and cell towers to adjust a phone's configuration [Two09]. Another uses acceleration to determine if a collision occurred [lif]. Still others use GPS location to measure how much a user has covered in a race [sof] and to estimate personal carbon footprint [eco]. The fifth relies on the detection of nearby users in order to establish social connections [wer]. Key characteristics of these emerging *Context-Aware Adaptive Applications* (CAAAs) are that they are intensely *context-aware* and continually *adaptive* to changes in context.

This modern applications present a unique characteristic: alongside with the normal execution flow, in which the application interacts with the user, there is an additional, often independent, execution flow in which the application monitors the surrounding environment looking for certain situations. This parallel execution waits for updates from one of the monitoring sensors. By means of dedicated routines, APIs or middlewares refreshed values are read and computed. Such a computation aims to chose whether the application has to adapt or not. If not, nothing happens and the application continues its current behavior, otherwise an adaptation is performed. During each adaptation the application applies a new behavior by changing its features, by modifying part of its configuration or by loading or unloading required components.

Adaptations resulting from environmental changes create new challenges in terms of testing and validation due to a novel class of failures from which they suffer. Since the application behavior depends on adaptations, unpredicted or incorrect adaptations may apply behaviors which, for a certain environment, may prove undesirable, faulty or even dangerous. The perception of the surrounding environment depends on sensors and components which can be enabled or disabled both by adaptations and users.

Certain behaviors may fail because the user interfered with one of the required components. Sequential behaviors may perceive the same environment differently and may start interfering with each other. Developers design adaptations to be triggered when certain conditions are met. However, there may exist borderline environmental conditions that create interferences between multiple adaptations resulting in undesired or incorrect adaptations.

In this Thesis we describe, compare and evaluate validation methodologies for *Context-Aware Adaptive Applications* (CAAAs). The acronym CAAA has been introduced in one of our initial publications to identify applications which constantly monitor and react to their environment [SRWE08a]. The nature of such applications may vary from small standalone mobile applications to peer interconnected nodes of a pervasive system. Our main focus is on standalone applications but the techniques we propose also apply to single nodes of infrastructured systems.

Independently from its purpose, each CAAA adapts as a result of computations based on values read from its *context* of execution. Such *context* includes various kinds of informations spacing from sensor readings from the surrounding environments to internal configuration parameters. In this Thesis we aim to validate the correctness of such adaptations by means of novel context-driven techniques. We define our techniques as *context-driven*, meaning that the validation is driven by context information also available to the application itself. Although some of the presented techniques may be successfully applied to different problems [CSRR09], they are not meant to be general purpose validation techniques, therefore they may not be effective in validating other aspects. In particular, even if an application has been flagged as correct, it does not mean that the application is bug-free but simply that its adaptation mechanism has passed certain fault detection criteria.

The remaining of this Chapter fixes boundaries and assumptions on which this research is based and enumerates all the contributions produced by this work. A running example, *PhoneAdapter*, is also presented to help the reader through the rest of the manuscript.

1.1 Background

Sensor readings are discretization of physical values sampled with a certain refresh rate depending on the sensor or on its configurations. Following the notation introduced in [LCT06, WER07a] we name *context variables* those holding values obtained by sensor readings. The refresh rate with which the context variables are updated vary from sensor to sensor, from few milliseconds to few minutes. Normally a CAAA relies on multiple sensors to better understand its surrounding environment. Each of such sensors will read the context and notify the application asynchronously. Information provided by different sensors are normally redundant in order to avoid errors in sensor readings and in order to be able to recognize certain situations even if some of the used sensors are off. If both the sensors are on and their readings are similar then the application can consider the reading safe. Contrarily if they show discordant values, because the context is changed and not all of them have refreshed or because they have reading errors, the CAAA may misinterpret such readings and adapt incorrectly.

Beside asynchronous refresh problems, which are an intrinsic issue of this category of applications, CAAAs also suffer from faults caused by problems in their logic. Such problems normally are caused by human mistakes in defining the application's reaction to contextual changes and, in most of the situations, are implementation independent, meaning that any implementation using the same logic will suffer from the same fault.

As any other class of application, CAAAs have evolved through the time following the increase of their popularity and the availability of new technologies. We have identified five different evolutionary steps which still coexist depending on the application domain. Knowing and understanding these evolutionary steps is important to identify potential sources of faults.

1. At first, coping with proprietary and custom hardware, CAAAs were implemented as hard-coded applications targeting exactly the device in which they were being executed. Developers were implementing all the components starting from drivers through the adaptation

   logic to the user interface. This is still the case for some industrial applications or for small embedded proprietary applications such as simple house monitoring systems. In this Thesis, in Section 4.1.4 we consider one such application, the Conveyor Belt [LCT08].

2. With the standardization of certain sensors and with the rise of wireless communication protocols such as WiFi, Bluetooth and Zigbee, sensor manufacturers have embedded their sensors in small multipurpose external units, such as handsfree or GPS. Those external units were then connected to industrial handheld devices such as PDAs. This change in the hardware platform lead to a second stage in the development of CAAAs. Developers were provided with proprietary API to control available sensors, and with frameworks for the user interface. At this stage the complexity moved from directly controlling the sensors to implementing the application logic, leading to more complex logics. Applications belonging to this second stage are commonly proprietary and tightly bounded to a certain driver.

3. With the advent of modern Smartphones, such as Android [All09], iPhone [App09], OpenMoko [Ope09], Blackberries [Res09] and modern PalmOS devices [Pal09], application developers could benefit from large frameworks encapsulating all the sensors with standard APIs [Sun09, Pro06a, Pro06b, Jav00, Ope08b, Ope08a]. This increased further more the portability of CAAAs and contributed to their diffusion to the public. As an example of this in Section 4.1.2 we introduce *TourApp* [SDA99].

4. The availability of standard framework built with high level languages made it possible the rise of context-awareness middlewares [BC04, CEM03a, FC04, Flo06, GPZ04, RC03, SRa]. These context-awareness middleware took care of most of the synchronization and adaptation problems and let the developer focus on the logic behind the adaptations instead of in its implementation. CAAAs' developers stopped designing the adaptation logic with imperative languages and started describing them in terms of *adaptation rules*. Such rules define triggers to which the application reacts and adapts. Rule based middlewares are the most diffuse, however other implementations exist, for instance middlewares using neural networks or support vector machines. For instance,

in Section 4.1.3 we discuss SeNIE which provides a gesture recognition tool based on a neural networks.

Adaptation rules can be simple enough to be understood and designed by end users. In fact adaptation rules can also be represented as a human readable description of the application's behavior (e.g. "if the battery is low turn off bluetooth"). With the increase of popularity of Smartphones and with the need to provide more ubiquitous and pervasive applications, developers allowed end users to specify custom configurations by providing graphical user interface to redefine at runtime the adaptation logic. This describes the current state of the art of CAAAs. The increased number of scenarios in which CAAAs are now used and the complexity which such adaptation rules can reach also increase the number of faults caused by an incorrect or fault prone adaptation logic. This is caused by two factors: first users are not really aware of problems behind rules definition, second, the adaptation logic which was once tested by application developers before releasing the product, is now not tested because it is defined at runtime. In Section 2.5 we mention *Locale* [Two09], a CAAA belonging to this category. In Locale, for instance, a set of rules is used to raise events that change the phone's behavior or notify external applications based on predefined situations (e.g., low battery) and on situations that a user can define (e.g., being in a meeting).

5. Since now CAAAs are distributed to multiple platforms the next evolutionary step would be the definition of a standard language to define such application rules. *PhoneAdapter* [SRWE08a], described in Section 1.2 is an example of how a platform independent, in this case, XML based, definition of a CAAA may look like.

In middleware based applications CAAAs' validation should be limited to rules validation, leaving the middleware testing to middleware developers. When an incorrect rule is triggered, or the correct one is not, a CAAA fails to adapt properly or behaves improperly. Again, in applications like *Locale*, failure reports often refer to issues associated with rules that trigger undesired behavior (e.g., the wrong location is set when sensor data is not available, leading to a phone ringing loudly in a meeting) or fail to trigger appropriate behavior (e.g., not turning GPS off, which may quickly drain the battery).

Discovering such *adaptation faults* in CAAAs is challenging because of various confounding factors:

1. The space of rules becomes complex to analyze in the presence of shared context variables, concurrent triggering of rules, and priority ordering of rules.

2. The context variables are refreshed asynchronously at different rates by the middleware, causing artificial, transient inconsistencies between the external physical context and its internal representation within the application.

3. It is becoming increasingly common for CAAAs to let their users configure their behavior; this can lead to runtime failures due to buggy user-defined configurations.

In practice, we have observed that developers attempt to control these factors and the associated introduction of faults by constraining the rule space (e.g., disallowing disjunctions in rules), enforcing stronger priority orderings (e.g, requiring that each rule has a unique priority), restricting the adaptation actions that can be taken based on rules (e.g., disabling the power-off feature on GPS because it may be used by other rules), and reducing the number or type of sensor information that can be considered (e.g., not letting the end user turn off particular services).

All of these practices are attempts to address an unsolved problem: human defined rules may be faulty and, before they can be applied, they need to be validated.

1.2 A Running Example: PhoneAdapter

In this section we present *PhoneAdapter*, an application that suffers from the kinds of faults peculiar to CAAAs that our approach is able to detect. To help the reader, *PhoneAdapter* will be used as running example across the manuscript and as a benchmark in the evaluations.

PhoneAdapter adapts a phone's *profile* according to context information. Phone profiles are a set of parameters that determine a phone's behavior, such as settings for display intensity, ring tone volume and vibration. Instead of requiring the user to select a profile manually, *PhoneAdapter* uses a set of adaptation rules to trigger automatic selection of a profile suitable to the user's current context. The selected profile prevails until a more suitable one is chosen through the triggering of other rules. The rule predicates are expressed over context readings from Bluetooth and GPS sensors on the phone plus the phone's internal clock. As we shall see, however, the selected profile is not always the most appropriate one.

The application is implemented on top of ContextNotifier, a J2ME rule-based adaptation framework and middleware we have built for CAAAs [SRa], and targeted for deployment on the Nokia N95 mobile phone. Originally we run the application and its adaptation rules within an emulation environment called TestingEmulator [SRb].

By means of a user created XML configuration file, *PhoneAdapter* defines which behavior exists and under which circumstances it should be applied.

The setup used in the remaining of this thesis defines nine profiles:

1. *General*: the initial profile, which defines a user-specified default configuration, and which is applied by default when the phone's sensors are unable to detect any activity related to one of the remaining profiles;

2. *Home*: increases the ring tone volume and removes vibration when the user is at home;

3. *Office*: mutes the ring tone and activates vibration when the user is in his office;

4. *Meeting*: mutes the ring tone and disables vibration when the user is in a meeting;

5. *Outdoor*: increases the backlight intensity and speaker volume when the user is outdoors;

6. *Jogging*: increases the backlight intensity and speaker volume and also activates vibration when the user is jogging;

7. *Driving*: connects to the car's handsfree communication system when the user is driving;

8. *DrivingFast*: diverts calls when the user is driving fast;
9. *Sync*: periodically synchronizes personal information on the phone with the user's home or office PC when the phone is not in use and the PC is discovered via Bluetooth.

Some profiles are more important than others for safety or social reasons, so it is possible to sort the order with which they will be applied with a weak priority order. In this scenario *DrivingFast* and *Driving* are applied with high priority, *Meeting*, *Home*, *Outdoor*, *Jogging*, and *Office* with medium priority, and, since synchronization can be performed after other activities have been accounted for, *Sync* is applied with low priority.

Over several executions, we observed a number of non-obvious problems with *PhoneAdapter*. For instance, the profile *Sync* is never applied when the phone is adapted to *Home* or *Office*. Also, the rules that trigger adaptation to *Home* and *Office* can be satisfied simultaneously—which is possible if the user's office PC is discovered in the home location, or vice versa—causing nondeterministic adaptation to one of the two profiles.

But there are even more subtle problems. While the phone is in the process of adapting according to one rule, if some other rules are satisfied, the phone can pass through a sequence of different profiles within the same context. This chain of adaptation causes multiple problems. In particular, the user can be annoyed by the multiple adaptations, and through the sequence of adaptations the desired profile can become unreachable. For instance, when the user has left his office or house and has entered his car, the phone is supposed to adapt to *Driving*. However, if the Bluetooth sensor does not detect the handsfree system fast enough, the phone can adapt to *General*, and then to *Outdoor*. Then when the user starts driving, the speed increases and the phone adapts from *Outdoor* to *Jogging*. From *Jogging* the phone cannot adapt to *Driving* even when the handsfree system finally is detected, because the application cannot adapt from *Jogging* to *Driving* directly according to the rules.

It can also happen that, through a chain of adaptations, the predicates of contradictory rules are satisfied and keep activating each other. For instance, from *Meeting*, when the meeting is over, the application adapts to *Office*, in which another rule restores *Meeting*, leading to a loop, because there exist particular inputs that can satisfy the necessary predicates simultaneously. For instance, the predicate $time > meeting\_start$ is always true after the meeting.

The timing of context updates can affect the triggering of rules in other ways. Since context updates occur asynchronously, the internal view of the context can become inconsistent temporarily, which causes the evaluation of rules to produce incorrect results or to trigger in a manner that violates their priorities. For instance, if a meeting is scheduled but the user is going from the office to his car, the higher refresh rate of time relative to Bluetooth can force an adaptation to *Meeting* instead of *Driving*.

Existing analysis techniques do not differentiate predicates based on asynchronous input signals such as GPS and Bluetooth. Such predicates could cause abnormal adaptation when updated asynchronously. Also, the space of rules becomes complex and non-trivial to analyze in the presence of shared context variables, of rules that can be concurrently triggered, and of rules with priorities. Although these problems can be fixed manually, we have no guarantee that we have discovered all possible faults of this nature. We therefore need systematic ways of discovering adaptation faults like the kinds described above. Our approach aims to help software engineers (especially rule designers) analyze rules and detect faults in them automatically.

1.3 Contributions

This section summarizes the achievements and contributions produced by this Thesis. The manuscript starts from a general analysis of CAAAs' architectures and of their related failures. Common causes of faults are generalized and techniques to detect them automatically, are proposed.

1.3.1 Architecture and Failure Taxonomy

Due to their context-aware and adaptive nature CAAAs' implementation follows a general architecture which became more evident with the rise of context-awareness middlewares. In Chapter 3 this architecture is modelled in detail with particular focus on how context variables are computed and evolve inside the application layers which we have identified.

Once a high level architectural model is defined, known failures can be re-examined in terms of which layer fails and which contextual variable is faulty. Such context-related failures can hardly be classified with existing bug reporting tools. In fact their error reports miss those context-aware information which would provide a better support to software developers. In Chapter 4 we describe such taxonomy and we use it to classify instances of failures which we have identified in various applications.

1.3.2 A-FSM Model

The diffusion of context-awareness middleware moves the complexity of CAAAs from the context-awareness to the adaptation logic. Nowadays developers mainly focus on defining their own adaptation logic or in providing design tools to let each user to configure its own configuration.

To support this new trend in Chapter 5 we propose the *Adaptation Finite State Machine* model, a model for the application logic of CAAAs. This model uses a Finite State Automation (FSA) also known as Finite State Machine (FSM) to model the adaptive behaviour of a CAAA, and can be considered one of the main contributions of this Thesis as all the following validation techniques are built on top of that.

Given that CAAAs are more and more taking advantage of context-awareness middlewares and their implementation is often reduced to the definition of behaviours and adaptation rules, the definition of a standard model acquires more and more relevance and becomes crucial in a scenario in which such configuration needs to be portable from one implementation to the other.

1.3.3 Fault Patterns

As described in Chapter 3 most of the failures occurring in CAAAs can be reduced to a set of common causes. By means of the A-FSM such common causes have been encapsulated in a set of properties which, if violated, may lead to a failure. We name such violations *fault patterns* and we describe their effect in Chapter 5.

Being able to identify properties and faults' patterns is a big step in CAAAs validation because: It allows their automatic verification; It abstracts the problem from testing if an implementation is correct to defining what correct means and letting automated validation techniques to prove such correctness.

1.3.4 Validation Techniques

Given an A-FSM Model describing a CAAA, such model can be validated against our Fault Patterns. Even if such validation can be performed in several ways in Chapter 6 we present three different implementations by defining for each of them a set of unique validation algorithms.

At first, in Section 6.1 we describe an enumerative technique which is not scalable but which provides a detailed error report. In Section 6.2 we solve the scalability issues by means of a more optimized BDD-based (Binary Decision Diagram-based) context representation. Both these approaches rely on the definition of specific pattern detection algorithms. At last, in Section 6.3, we show how violations can also be detected by converting the definition of their violation in a planning task and by using a planner to find instances of such violations. This third approach does not require the definition of a detection algorithm and supports more complex properties and provides a fully detailed trace starting from an initial configuration till a detected fault instance.

1.3.5 Evaluations

The techniques described in Chapter 6 are capable of validating CAAAs against known faults and additional techniques could be defined. By examining them we found that there is a trade off between the details of the error report, the ways faults are aggregated and the scalability of the technique. Moreover the computation time is not always the issue, and the application may need to be validated on devices where the limited memory is a stricter constraint. In Chapter 8 we compare our three techniques and discuss their strengths and weaknesses.

This evaluation is important because not only it shows that no technique is superior to the others but also puts the basis for a more sophisticated validation in which at first the whole application is validated using approaches which prioritize scalability to error report's details and then only the faulty application subsets are revalidated with other less scalable but more accurate approaches.

1.4 Outline

The rest of this Thesis is organized as follows: in Chapter 3 we introduce a high level architectural model, on which the fault taxonomy presented in Chapter 4 is based. In Chapter 5 we present our A-FSM Model based on which, we define a set of properties which, if validated may lead to failures. In Chapter 6 we introduce three techniques capable of validating applications against the patterns of faults which we have presented. Such techniques are compared with each other in Chapter 8. Finally Chapter 9 presents conclusions and future works.

1.5 Publications

Initial results in the definition of an architecture for CAAAs have been published as a short paper in the 1st ICSE 2008 International Workshop on Software Architectures and Mobility (SAM) [SRWE08b]. Such work has then been revised, extended and published in a special issue on mobile architecture of the IEEE Journal on Software Systems (JSS) [SRWE10].

Behavioural faults, hazards and their related fault patterns have been presented at the 16th ACM SIGSOFT International Symposium of Software Engineering [SRWE08a] in which, faults were only detected using the enumerative approach. We introduced a first symbolic implementation in the IEEE 1st International Workshop on Automated engineeRing of Autonomous and run-tiMe evolvIng Systems (ARAMIS). The two improved symbolic approaches, which are presented in this thesis, as well as their evaluation have been published on a special issue of the IEEE Transactions on Software Engineering (TSE) [SER$^+$10]. The latest approach using planners to detect behavioural faults is still under review.

ContextNotifier and TestingEmulator, a framework and a testing tool which we have used to detect faults on CAAAs and which we have used to test manually PhoneAdapter have been presented on the demo session of the 5th ACM International Conference on Mobile Systems, Applications, and Services (MOBISYS).

The A-FSM model has been applied to Webservice composition in a work presented in the IEEE International Conference on Services Computing 2009 (SCC).

Chapter 2

Preliminaries

Software testing is any activity, that aims to evaluate an attribute or capability of a program (or even a system) and determines if it meets its required results [HH91]. One part of the software testing validation is the activity of checking that a software system meets its specifications and that it fulfills its intended purpose. To validate applications monitoring the environment in order to adapt and to provide their user a more suitable behavior, we need to understand what is their intended purpose. In this thesis we believe that their intended purpose is to always show the user the most suitable behavior according to the environment in which they are being executed. Then the problem is: how do we establish which behavior is the correct one? The best answer to this question is that it depends on the application's purpose. In the presence of a formal detailed specification of how the application should behave it would be possible to verify if such specifications are respected. However, in the presence of a variety of inputs it is not trivial for developers to define a detailed specification which would take in account all the possible situations. In this thesis we look at the problem from the opposite direction. Indeed, we can easily say that a CAAA is violating its purpose if, in a certain context, it exposes any undesired behavior. Then deciding which behavior can be considered undesired in a certain environment is a much simpler task which we solved by defining a set of properties and by considering all those undesired behaviors that are violating at least one property. To define such a set of properties we followed previous works on testing and validation for context-aware and adaptive applications and we attempt to isolate and classify the various fault they found. Then to apply such properties, we were inspired by the existing model checking and validation techniques.

The rest of this Chapter is organized as follows: Section 2.1 gives an overview of testing and validation techniques applied to context-aware applications; Section 2.2 introduces related work in model checking and validation techniques designed for a more general class of applications; Section 2.3 describes Ordered Binary Decision Diagrams, a data structure with which a Boolean representation of CAAA can be encoded; Finally Section 2.4 introduces the Planning Domain Definition Language, which has been used to convert CAAA validation into planning tasks.

2.1 Testing and Validation Techniques related to Context-Aware Applications

In the literature there are various work designing and testing middleware-centric context-aware systems [LCT06]. Middleware-centric context-aware applications are part of a centralized system in which a centralized coordinator controls several distributed applications using the contextual information they provide. The communication between the coordinator and the distributed applications requires an infrastructure, such as a WiFi, a 3G or a wired network. In terms of context-awareness this architecture provides various benefits. In particular, sensor reading errors are identified by comparing readings from different sources, and interferences between different applications are prevented by having a centralized coordinator. With a certain approximation, middleware centric systems can be considered an evolution of client server distributed systems using also contextual information. Note that in this architecture the context-aware clients tend to be as simple as possible, their coordinator is delegated for all the computation by forwarding all the information and waiting for a command. In this thesis we do not focus on context-aware applications which are part of a middleware-centric system, because their implementation tend to be trivial. Instead we focus on those stand alone context aware applications which gather and compute environmental information by themselves. When such applications are part of a distributed system they consider information coming from the system, as part of their context, and therefore they are still able to take decisions independently from the rest of the system. Please note that with a certain abstraction the coordinator of a middleware-centric system can also be considered as a standalone application receiving data from remote clients, therefore they can also be validated with the techniques described in this thesis.

The basic way of validating a CAAA is to feed the application with various environmental inputs and to observe its resulting behavior. Depending on the knowledge that we have on the application there are three ways in which an application can be validated. If the application is a black box, meaning that we do not have any information about its internal structure, we cannot monitor its internal configuration and that we cannot change or instrument its source code, it can be validated by executing it inside a controlled environment. For instance, mobile applications can be executed in an emulator, simulating a certain environment and observing how the application reacts [SRb]. This approach has various limitations: it requires an oracle to detect failures; it is not trivial to decide which context need to be generated; user input may also need to be emulated; it requires an adequacy criterion to terminate the validation and it is not trivial to trace failures to their causing source. In addition, this approach can be used to validate user defined configurations only if those are known a priory and if they are part of the simulation. However, it cannot be used for runtime validation of user defined configurations. If it is possible to manipulate the application's code, for instance by instrumenting it, then we can validate the application by forcing the execution of certain execution path and by monitoring the results. This approach is discussed in Section 2.3. This approach is very effective in detecting bugs in the source code, but cannot detect issues related to user defined configurations. If we can design or extract a model describing how the application reacts to environmental changes then we can validate such model. A model can also be extracted from user configurations and those can also be validated. The latter approach is applied in this thesis because we want to focus on user defined configurations.

CAAAs suffer from concurrency issues as a result of the asynchronous updates from their connected sensors. Concurrency and race condition have been broadly studied in the literature. In his work Sen uses potential data race information obtained from a dynamic analysis technique to control a random scheduler of threads so that real race conditions get created with very high probability and those races get resolved randomly at runtime [Sen08]. Similarly Lei and Carver [LC06] use reachability testing to generate synchronization sequences automatically and on-the-fly. Although this potential data races also occurs in CAAAs the issues here is deeper. CAAAs use multiple sensor reading representing the same contextual information or information related to each other. Preventing data races solves issues caused by threads reading and writing the same variables but it does not prevent the application from computing various context information inconsistent between each other. This and similar techniques need not only to prevent race conditions but also understand the contextual-meaning of each variable. A race-safe execution can still lead to a fault due to the internal inconsistence between multiple variables representing the same information.

Previous works on testing context-aware applications have identified the existence of *context variables* and the need of sharing those variables between multiple threads both updating and reading them. This has also been identified as one of the major source for failures in CAAAs. In other terms, multiple sensors are read by multiple threads, which store the read values in variables which are representing the context inside the application [LCT06]. These context variables are then accessed by the application to perform some computation. The thread reading or depending on those variables are not aware of when those contextual values have been updated or when their value will be refreshed. Similarly, the threads looping to refresh such variables are not aware of the impact that overwriting that variable will have on the computation.

For instance, consider a scenario in which a thread U is updating a Boolean context variable b and another thread R will need to read that variable twice. In particular, consider the sequence in which b is false and R reads it, U updates it to true, and then R reads it again. This simple sequence generates a paradox in which b had two values during the same computation and in which the result computed by R may be inconsistent. To address this problem Wang et al. [WER07b], and Lu et al. [LCT06] use code analysis to detect context-dependent variables or codes attempting to identify and remove those execution paths in which the context has become inconsistent. In general, by observing in the execution graph of these *context variables* and *context-aware program point (CAPP)* it is possible to understand if the context has become inconsistent and to abort the computation.

From a testing point of view, by controlling the preemption of different threads, it is also possible to identify execution paths, in which those variables are used. Sequences of paths can be executed on the system under test until some adequacy criterion has been satisfied. Both these research works suggest their own set of adequacy criteria based on the execution of specific sets of drivers in order to cover all the possible paths in which contextual variables are assigned and used.

Lu et al. [LCT06] apply their work to an industrial application driving a conveyor belt. For such application in which the context-aware thread and the algorithm are part of the implementation, this and similar techniques work perfectly. The execution of all possible paths suffers from scalability issues on the number of threads and variables, which is another limitation of this approach. In addition code analysis requires the logic of the computation to be part of the code. In these modern applications the logic is not written in the application, but instead is loaded either from a configuration file or from a database.

The research work in this thesis focuses mainly on those applications built on top of existing frameworks and middlewares, with particular interest in those applications in which the logic is loaded and applied at runtime (i.e. user defined).

2.2 General Purpose Model Checking and Validation Techniques

In the field of software verification and testing the techniques that we apply in this thesis can be classified as model checking and software validation. Model checking is an automatic technique for verifying finite-state reactive systems, such as sequential circuit designs and communication protocols. Specifications are expressed in temporal logic, and the reactive system is modeled as a statetransition graph. An efficient search procedure is used to determine whether or not the state-transition graph satisfies the specifications [Cla97]. Those software capable of perform this automated verification are known as model checkers.

Among them SPIN [Hol03] is an examples of a model checker using properties expressed in Linear Temporal Logic (LTL). SPIN is based on a language called Promela which is a verification modeling language. However the abstractions and the structures of distributed systems which Promela provides have a different level of abstraction from the ones which are required to validate a CAAAs. Although it would be possible, it would require a serious effort to model a CAAA with Promela. In addition time is only one of the multiple context which CAAAs use, therefore LTL is also a limit. For instance consider the LTL predicate $P : f(t)$. If we consider time as part of the context (as it normally is) then we can also define $P\prime : f(C)$ where C represent the context. $P\prime$ is expressed in some sort of context-driven logic and it is more suitable to define conditions on a CAAAs. This kind of formalism is possible in other languages such as PDDL [GL97] which is unfortunately not supported by existing model checkers. Another interesting tool is Java Path Finder (JPF) [HP00] which translates a given Java program into a Promela model, which then can be model checked using Spin. The Java program may contain assertions, which are translated into similar assertions in the Promela model. The Spin model checker will then look for deadlocks and violations of any stated assertions. Eventually developers could use the Java assertion mechanism to write constraint in their CAAAs and use JPF to validate them. On the other hand this approach is only semi automated as it still requires developers to write their own assertions.

Other model checkers such as PRISM [KNP02], known as probabilistic model checkers are capable of validate probabilistic automata (PAs) and probabilistic timed automata (PAs) using a discrete-event simulation engine. This kind of model checkers not only can verify if a certain condition is met but also with which probability within a certain amount of time. Although this kind of information is useful our intent is not to predict what will happen but to verify what can happen wrong.

More sophisticated techniques such as static and dynamic source code analysis [Bin07] represent the internal execution of an application using data structures such as the Control-Flow Graph (CFG), the Value-Dependence Graph (VDG) or even a finite-state automata. All of such representations can also be used to validate and detect a given pattern. For instance Christodorescu and Jha [CJ03] used static analysis to detect malicious patterns making it possible to detect malicious code in the application under test. This together with symbolic execution [Kin76] could be eventually used to craft various combinations of inputs which can trigger certain fault conditions. However, the kind of fault condition that we are trying to validate in this thesis require an higher abstraction level, identifying context variables, adaptation and states which is hard to get using source code analysis or symbolic execution. It may be possible to use annotations and code instrumentation to provide the required abstraction level. For instance it could be possible to annotate all the context readings and to trace their path in the CFG as done by Wang et al. [WER07b].

2.3 OBDDs and Symbolic Computation

Most of the modern model checker and system using a large quantity of boolean variables handle them with the support of a data structure known as Ordered Binary Decision Diagram (OBDD). Ordered Binary Decision Diagrams have been particularly successful in the last two decades because they offer, a much more compact representation of Boolean functions with respect to other canonical forms (e.g., conjunctive/disjunctive normal forms) [Bry86]. During the implementation of the validation techniques presented in this Thesis we also used OBDDs to improve the scalability and the speed of our algorithms. Here we present a brief explanation of how OBDD works and how they can be used. The reader will need this knowledge to better understand the following chapters.

A *Boolean variable* x is a variable whose value is either 0 or 1. A *Boolean function* of n Boolean variables is a function $f : \{0,1\}^n \to \{0,1\}$. *Boolean formula* can be seen as Boolean functions. For instance, the Boolean formula $x_1 \wedge (x_2 \vee x_3)$ can be seen as the Boolean function $f(x_1, x_2, x_3) = x_1 \wedge (x_2 \vee x_3)$, with $x_1, x_2, x_3 \in \{0,1\}$.

A rooted, directed graph G can be associated to every Boolean function $f(x_1, \dots, x_n)$ by imposing an ordering on the variables $x_1, \dots, x_n$, and by reducing the graph (in the sense explained below) [Bry86]. The graph G is called the *Ordered Binary Decision Diagram* of f. For instance, the reduced graph associated with the Boolean function $f(x_1, x_2, x_3) = x_1 \wedge (x_2 \vee x_3)$ is depicted in Figure 2.1 (b), by "simplifying" the graph depicted in Figure 2.1 (a). Formally, a graph is reduced by iteratively eliminating the vertexes which are the root of two isomorphic subgraphs, and by merging isomorphic subgraphs. A graph is said to be *reduced* if it contains no isomorphic subgraphs and no vertexes v and v' such that the sub-graphs rooted at v and v' are isomorphic. We assume here that the left child of a vertex corresponds to the choice of the value 0 (i.e., *false*) for the variable preceding it, while the right child correspond to the choice of the value 1 (i.e., *true*). Thus, the leftmost path of Figure 2.1 (a) corresponds to an assignment of 0 to all variables and, consequently, to the value 0 to the expression $f(x_1, x_2, x_3) = x_1 \wedge (x_2 \vee x_3)$.

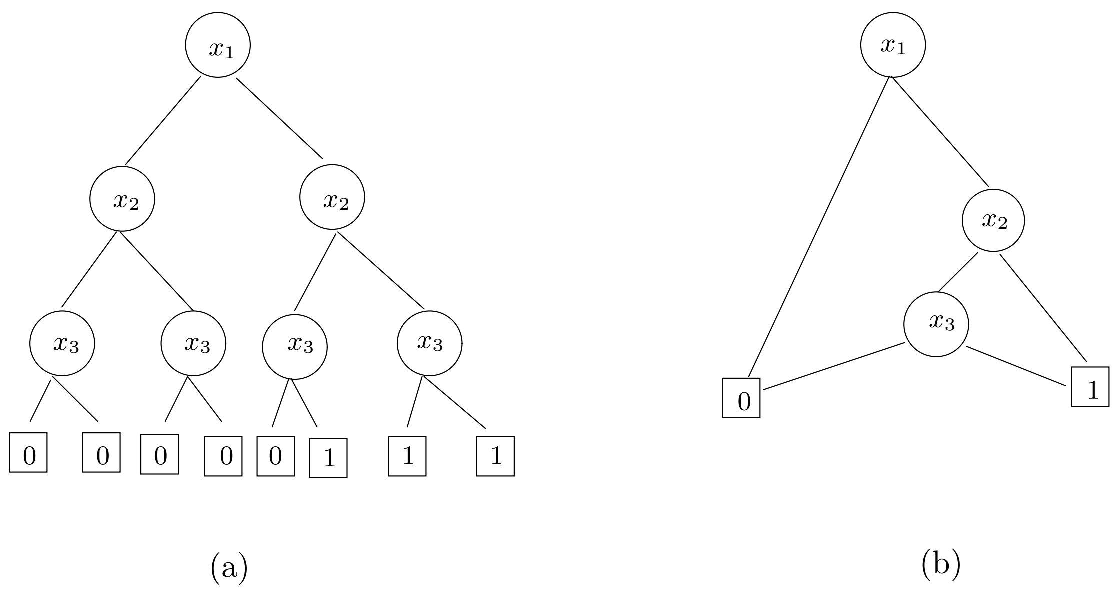

Figure 2.1: OBDD example for $f = x_1 \wedge (x_2 \vee x_3)$.

State	Boolean vector	Boolean formula
S_1	$(1,1)$	$x_1 \wedge x_2$
S_2	$(1,0)$	$x_1 \wedge \neg x_2$
S_3	$(0,1)$	$\neg x_1 \wedge x_2$

Table 2.1: Example of Boolean encoding.

It is shown in [Bry86] that, given a fixed ordering of the Boolean variables $x_1, \ldots, x_n$, the reduced graph of any Boolean function $f : \{0,1\}^n \rightarrow \{0,1\}$ is unique. Thus is, OBDDs are a *canonical* representation for Boolean functions.

Boolean operators can be applied to Boolean functions; for instance the disjunction operator $\vee$ can be applied to two Boolean functions f_1 and f_2 to obtain a third Boolean function $f_3 = f_1 \vee f_2$ allowing Boolean calculus to be applied to OBDDs.

The use of OBDDs to represent states and transitions has been proposed by [McM93]. The key idea here is to represent states (and sets of states) as Boolean formula which, in turn, can be encoded as OBDDs. As an example, consider the set of states $\mathcal{S} = \{\mathcal{S}_\infty, \mathcal{S}_\in, \mathcal{S}_\ni\}$ and the relation $\mathcal{R} = \{(\mathcal{S}_\infty, \mathcal{S}_\in), (\mathcal{S}_\in, \mathcal{S}_\ni), (\mathcal{S}_\ni, \mathcal{S}_\infty)\}$ (i.e., a simple loop). Let $N = \lceil log_2 |\mathcal{S}| \rceil$; in our example $N = 2$. Each element $S \in \mathcal{S}$ is associated with a vector of Boolean variables $\overline{x} = (x_1, \ldots, x_N)$; that is, each element of S is associated with a tuple of $\{0,1\}^N$. Each tuple $\overline{x} = (x_1, \ldots, x_N)$ is then identified with a Boolean formula, represented by a conjunction of literals, that is, a conjunction of variables or their negation.[1] It is assumed that the value 0 in a tuple corresponds to a negation. The encoding of the states in our example is given in Table 2.1.

[1]By slight abuse of notation, the same symbols $x_i (i \in \{1, \ldots, N\})$ are used to denote both Boolean variables in a vector and atomic propositions in logical formulae.

Sets of states are encoded by taking the disjunction of the Boolean formula encoding the single states. For instance, the set of states $\{S_1, S_3\}$ from the example in Table 2.1 is encoded by the Boolean formula $f = (x_1 \wedge x_2) \vee (\neg x_1 \wedge x_2)$.

A new set of "primed" variables $(x'_1, \ldots, x'_N)$ is introduced to encode the relation between two states $S, S' \in \mathcal{S}$. In particular, if $S\mathcal{R}S'$ holds, then S is encoded using the non-primed variables, S' is encoded using the primed variables, and the relation $S\mathcal{R}S'$ is expressed as a Boolean formula by taking the conjunction of the encoding for S and S'. The whole relation $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$ is encoded as a Boolean formula by taking the disjunction of all the transitions in R. In our example, the transition relation is encoded by the following Boolean formula f_R:

$$f_R(x_1, x_2, x'_1, x'_2) = ((x_1 \wedge x_2) \wedge (x'_1 \wedge \neg x'_2)) \vee$$
$$((x_1 \wedge \neg x_2) \wedge (\neg x'_1 \wedge x'_2)) \vee ((\neg x_1 \wedge x_2) \wedge (x'_1 \wedge x'_2))$$

2.4 Planners and PDDL

At the end of this Thesis we propose a planner-based validation technique as an alternative approach. The use of planner to solve satisfiability problems is not new to the literature [Ede08, ABM09], however as far as we know, planners where never used to validate context aware applications. In this thesis we benefit from similarities between the planning domain definition language and rule-driven computation which is driving the adaptation mechanism of most of the CAAAs.

From a general point of view, planners are designed to solve a *reachability* problem (i.e., how to reach some goal), and to report the trace leading to the goal from the initial state. Typically, planners implement heuristics and other efficient techniques for their operation, and in this sense they may be more suitable than model checkers for certain classes of problems.

The Planning Domain Definition Language (PDDL) is a LISP-like language for the definition of planning domains and problems, developed by the model-based planning community as a standard language for planning competitions [GL97]. A PDDL planning problem is a domain augmented with a set of goals and constraints that define one planning instance with respect to a given model. The following two listings report a classical planning example encoding a scenario in which a certain number of blocks need to be moved on a table to achieve a desired final configuration.

This code describes a domain called `blocks-worlds-domain`, which requires three features, listed after the `requirements` keyword: the possibility of using conditional effects and equalities, and the support for a STRIPS-like syntax [RN03]. The domain contains a constant `Table`, and three predicates: `(on ?x ?y)`, `(clear ?x)`, and `(block ?b)`. The notation `?x` denotes a placeholder for a certain object; the predicate `(on ?x ?y)` takes two arguments and its intuitive meaning is that object `?x` is on top of object `?y`. The predicate `(clear ?x)` is used to express the fact that nothing is on top of `?x`, and the predicate `(block ?b)` denotes that `?b` is an actual block. The `(:action` section contains the declaration of how the domain evolves. In this particular case, the only action is called `puton` and takes three parameters `?X, ?Y` and `?Z`, with the intuitive meaning of removing a block `?X` from `?Z` and placing it on top of `?Y`. The `precondition` section of the action lists a set of

Algorithm 1 Simple PDDL domain example.

```
; Domain definition
(define (domain blocks-world-domain)
    (:requirements :strips :equality :conditional-effects)
    (:constants Table)
    (:predicates (on ?x ?y) (clear ?x) (block ?b))
    (:action puton
    :parameters (?X ?Y ?Z)
    :precondition
         (and
             (on ?X ?Z)
             (clear ?X)
             (clear ?Y)
             (not (= ?Y ?Z))
             (not (= ?X ?Z))
             (not (= ?X ?Y))
             (not (= ?X Table))
        )
    :effect
        (and
            (on ?X ?Y)
            (not (on ?X ?Z))
            (when
                (not (= ?Z Table))
                (clear ?Z)
            )
            (when
                (not (= ?Y Table))
                (not (clear ?Y))
            )
        )
    )
)
```

constraints, expressed as a Boolean combination of predicates over the parameters, that must be true for the action to be "enabled" (notice the use of equality). Similarly, the effects of the action are listed in the `:effect` section as a Boolean combination of predicates (notice the use of the `when` keyword to express conditionals).

PDDL domains provide a general description of how a blocks-world looks like; concrete instances of this domain are defined in *problem files*, such as the one reported below:

Algorithm 2 Simple PDDL problem example.

```
; Goal definition
(define (problem tower-invert3)
        (:domain blocks-world-domain)
    (:objects A B C)
    (:init
        (block A)
        (block B)
        (block C)
        (block Table)
        (on A B)
        (on B C)
        (on C Table)
        (clear A)
        (clear Table))
    (:goal
        (and
            (on B C)
            (on C A)
        )
    )
)
```

Each problem has a name (`tower-invert3` in this case), and it must be an instance of a certain domain. In this particular example, the problem contains three objects A, B, and C, in addition to the constant Table. The keyword `:init` defines the *initial state* of the problem: the three objects and the Table are blocks, and initially A is on top of B, B is on top of C, C is on top of the Table, and A and the Table are clear. The final goal is to achieve an inverted situation: B should be on top of C, and C should be on top of A. The solution to this problem is a sequence of actions defined in the domain to achieve the desired goal, or an error message if the goal cannot be achieved. In the particular example above, a solution is simply a sequence of moves `puton` to achieve the desired configuration.

PDDL domains can include a number of other requirements, such as typing, durative actions, constraints, etc. For the purposes of our work we employ *fluents* [Thi05]. The fluent calculus extends the standard semantics of PDDL by providing the notion of states. The standard PDDL semantics defines a predicate for each single logical property which needs to be modelled; for instance, three predicates are needed to denote the fact that an object can be of three possible colours. This may result in quite a large predicate space. However, multiple predicates can be avoided by defining single logical atomic properties called fluents whose value can be set, compared and retrieved.

Various planners are available to solve PDDL planning problems with the extensions mentioned above; we refer to the proceedings of the International Conference on Automated Planning and Scheduling (ICAPS [ICA]) for further details. For the purposes of this work we use SGPlan [CwHW04] and MIPS-XXL [EJN06] because they implement all the PDDL3 features that we required.

2.5 About Locale

Among the CAAAs that we have examined during the progress of this thesis one captured our attention. In this commercial application the developers took some very particular design decisions in what we assume was an attempt of preventing certain context-based faults from happening. We decided to discuss this design decisions here to show how developers could make their application more robust.

Winner of the Android Developer Challenge, Locale is a context based phone profiler. The user, through a very intuitive configuration GUI, sets up a list of predicates based on various contextual conditions, and associates to each of those predicates an action. When a predicate is satisfied the action associated with the predicate is performed. Actions include changes in the device configuration, various notifications to the user and the possibility to start third-party applications.

Due to its similarity with our running example, Locale developers had to face failures similar to the ones described in this thesis. Since the adaptation logic is designed by the user through the GUI at runtime, the user can potentially apply a configuration affected by context-awareness faults.

To prevent such faulty configurations to be executed it would be possible to validate it at runtime, either on the device or by sending it to a remote Web Service, and to propose possible corrections. Both these solutions can be implemented by using the validation algorithms proposed by this thesis.

However Locale developers, instead of verifying the correctness of the user input, decided to prevent the user from applying a fault prone input by imposing a set of limitations on the input itself. It is interesting to observe how such limitation indeed prevent several faults from happening:

1. In Locale the user specifies predicates, and when one of those predicates is satisfied, the device adapts to a state in which that predicate is satisfied and the action related to that predicate is performed. However the user has also to order such predicates with a strong priority order, meaning that when a predicate with higher priority is applied the ones with lower priority are not evaluated. If a predicate with higher priority than the predicate of the state currently applied is satisfied, then Locale adapts to the new state. With this simple limitation Locale developers have protected their users from adapting nondeterministically and all the drawbacks discussed in Section 5.1.1.

2. In Locale users can assign actions which will be performed after an adaptation. To prevent such

actions from creating side effects, the Locale developers do not allow action to turn off sensors which could have been used to sense the context and to evaluate predicates. By introducing this limitation Locale developers have protected their users from those faults which in Section 5.2.4 we will call state invariant violations.

3. Locale users can specify predicates triggering certain behaviors. Those predicates are evaluated by their priority order. As long as a some predicate is satisfied Locale only allows adaptations to states identified by predicates with a higher priority level than the current one. If a predicate with a high level of priority is almost always satisfied it would prevent predicates with lower priority from being triggered causing rule/state liveness faults. To prevent these faults, the Locale developers forced users to use only conjunctions and negations in the predicate definition and to use each single context at most once. The more users aggregate conjunctions over different context, the more they are narrowing the space of values satisfying predicates. Moreover, since they can only use each context at most once, they cannot design tautologies such as $A \wedge \neg A$. In Appendix A we describe how predicates over the same context suffer also from hazards. Predicates without disjunctions are by design less vulnerable to hazards.

By examining these limitation that the developers have imposed on the definition of predicates in Locale we can easily assume that during various tests, users experienced some of the faults we are addressing and that developers have preferred to reduce the possibility of their occurrence rather than validating the set of rules as we do in this Thesis [2]. Note that since all these limitations apply to a single rule, faults can still occur as a result of multiple predicates. For instance it is possible to design two high priority predicates $P_1 = A$ and $P_2 = \neg A$ which will cause what in Section 5.2.2 we will call rule/state liveness fault in all the predicates/states with lower priority.

2.6 Terminology

Before continuing in next chapter, we should define some of the terms which, we will use in the rest of this Thesis.

System: with system we refer to the application, to all the devices or infrastructures are running together with their surrounding environment in which they interact. For instance for a mobile application, by system we indicate the application and the device including the sensors which are used and the surrounding environment. For a distributed application with the system we mean the whole composition of all its distributed components plus the network or the infrastructure that the application is using to communicate and the resources that are being used.

Application: with application we refer to all the software parts of a system including software components as well as resource and configuration files.

Middleware: with middleware we refer to a task specific layer of the application which is used as an interconnection between other layers or to create a certain level of abstraction. Middleware are normally independent software component which are embedded in the application to ease its development.

[2] We can only assume because the Locale developers did not reply to any of our emails.

Context-aware: any system or an application that is actively using sensor readings from its surrounding context as inputs to fulfill its purpose, we refer to as context-aware. With this definition a pacemaker can be considered as context aware because it use the frequency of the patient heart beat to administrate an electrical charge. At the same time an heart beat monitor is not context-aware because is simply monitoring the context and is not actively using it.

Fault: with fault we indicate an issue in the implementation or in the definition of a software component which can cause the system not to meet its specifications or not to fulfill its intended purpose.

Failure: with failure we indicate the variation from the expected behavior observed by the user or by a monitoring system as a result of a fault.

Error: with error we indicate the incorrect state in which the system is in, after a fault has occurred.

Chapter 3

A Model of the Architecture of CAAAs

This and the following chapters extend and complete a research work aiming to define a common high level architecture and a fault taxonomy for CAAAs [SRWE08b, SRWE10]. These previous works have been extended with an overview of CAAAs' architecture at different stages of their evolution which can help the readers foresee the future evolutions and with the addition of examples describing commercial applications which were suffering by context-awareness or adaptation faults.

Simultaneously, and sometimes independently from their users, CAAAs constantly monitor the context, elaborate read information and, if necessary, adapt. In other terms this context awareness means that the application stores and computes variables containing contextual information and that its behavior is based upon the value of certain of these variables. In Section 3.1.3 we name a set of variables representing the context as a *view* on the context. As it will be explained in this Chapter, CAAAs normally have at least three of such *view*s, which are computed hierarchically.

By isolating components handling such *view*s it is possible to identify a general common high level architecture for CAAAs. By means of this architectural model CAAAs can be cut horizontally in a stack of logical layers. Each layer represents a level of abstraction from the physical environment in which the application is being executed and provides to the above layers a more refined *view* on the context. In this Chapter we identify three of such layers of which, the top one is application dependent and represents the behaviors which a CAAA can assume; and the two lower ones represents how the application retrieves the context and how the context is computed in order to decide whether the application has to adapt or not.

This novel layered context-driven architecture supports developer and software testing engineers to understand and prevents common faults which can happen in this kind of applications. Traditional bug reporting tools classify faults in terms of how they can be reproduced by human interaction, which can hardly represent faults happening in proactive responses to context changes. In Section 4.2 we introduce a novel taxonomy dedicated to context aware faults. Faults are classified in terms of which context can trigger them and of which *view* they affect.

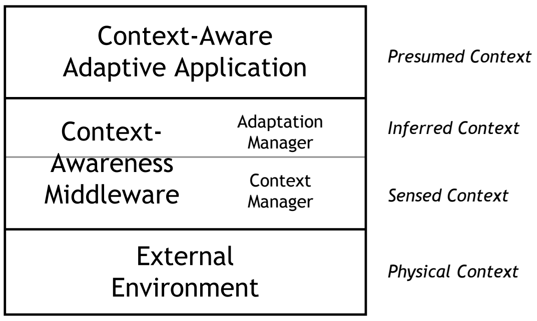

Figure 3.1: Canonical Architecture of Context-Aware Adaptive Applications.

3.1 A General Architecture

CAAAs have common characteristics which distinguish them from other applications types. Identifying such characteristics is the base for more specific analysis on CAAAs. Unlike other applications, CAAAs also monitor the context in parallel with their other application task and, when opportune, modify their behavior according to the new situation. Starting from this general point of view it is possible to define a first simple but effective architectural model.

The context-awareness execution flow starts by monitoring the *environment* often with parallel multiple threads and ends up by applying a different *application behavior* when necessary. With environment we represent both the surrounding physical environment, as well as any external configuration or parameters which the CAAA needs to use and to which it may have access. It is rather difficult to give a formal definition to what an *applicative behavior* is, as it depends on the nature of the application itself. In general applicative behaviors describe all the different modes in which an application can operate, meaning that to the same input, and with the same configuration, the application will produce a different output depending on the current operational mode. Note that such application is still deterministic because its response can be predicted by knowing the mode in which it is operating.

As depicted in Figure 3.1, we can model a layered architecture in which, on top of the surrounding environment an event-driven *Context Manager* collects and maintains low-level context information, and an *Adaptation Manager* queries and processes the current context values on behalf of the CAAA and automatically triggers adaptive behavior by the CAAA.

Context Manager and *Adaptation Manager* are the two most important macro-components of which the pervasive part of a CAAA is composed. Their implementation changed through the time according to the CAAAs evolutionary process which was explained in Section 1.1. Figure 3.2 depicts in detail how *Context Manager* and *Adaptation Manager* evolved at every step of the evolutionary chain of CAAAs. These architectures are the ones that we considered being the most interesting but variations are also possible. For instance developers with specific requirements may re-implement/customize existing layers to their needs.

Figure 3.2 also underlines which layers of the CAAA are given, as part of a framework/middleware and which ones need to be implemented by application's developers.

For embedded or industrial applications in which the hardware is custom or not-standard, develop-

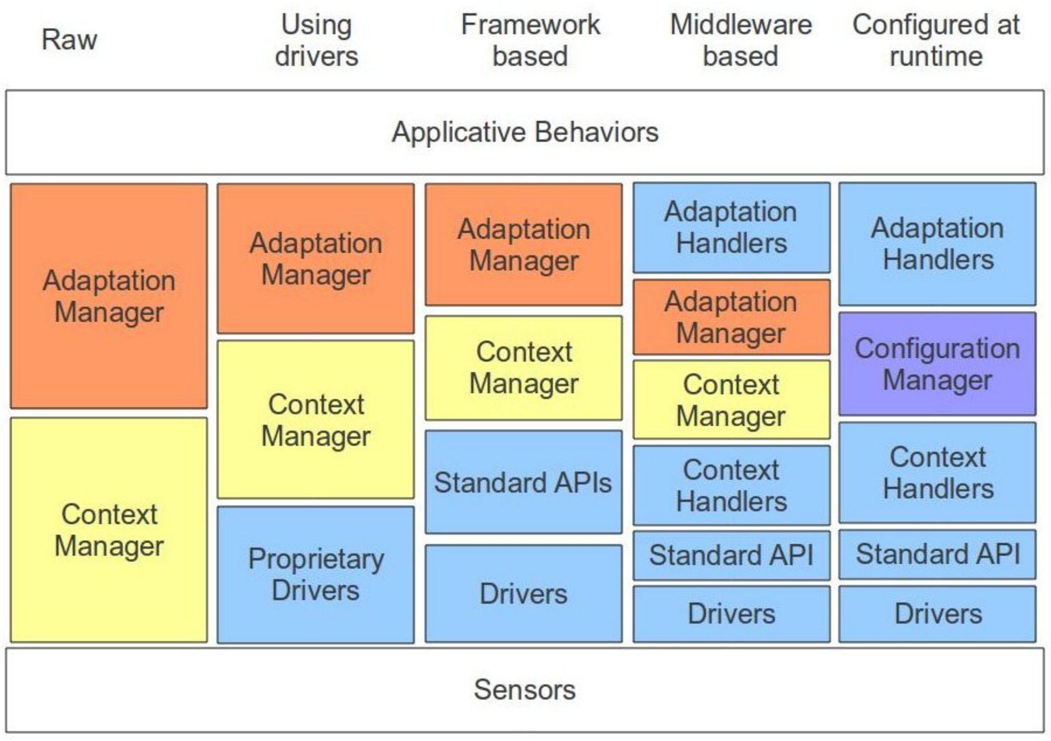

Figure 3.2: Evolution in the architecture of CAAAs

ers are responsible for the whole implementation. Applications built with this architecture are, generally, tightly coupled to a specific hardware and their users have very few (or even none) configuration options. In this first architecture applications have no support from external/existing components. The *ContextManager* reads raw values directly from the hardware and has to filter them before passing them to above layers. Similarly also the *Adaptation Manager* has no adaptation support and, in addition to the adaptation logic, it also directly switches application's behaviors. In CAAAs following this architecture the adaptation logic is normally hard coded, and tightly coupled with other application components.

In the presence of standard hardware the *ContextManager*'s implementation is greatly simplified, as provided drivers take care of the data readings, and noise filtering. Still the *ContextManager* communicates directly with the drivers strictly limiting the portability of the application. Deploying on devices with different sensors or even with a different version of the driver will require a strong maintenance.

With the advent of standard frameworks such as the Java ME platform [Sun09] the portability issue is solved with the introduction of an additional layer between the *ContextManager* and the drivers. The introduction of such layer has been possible with the definition of open standards respected by associations of device manufacturer and with the implementation of a set of open source APIs.

Frameworks make it possible to deploy a CAAA on multiple devices but still require the *ContextManager* to interact with an heterogeneous set of APIs and the *Adaptation Manager* to apply adaptations directly.

Context-awareness middlewares have been introduced to address exactly these two limitations. An additional layer on top of the framework's API simplifies the acquisition of contextual variables making often transparent to developers how a certain information was obtained. An extra layer has also been introduced between the *Adaptation Manager* and the application supporting adaptations. With the addition of these two extra layers the *ContextManager* focuses on *when* and *which* contextual information are read without considering *how*, similarly the *Adaptation Manager* focuses on *when* and *which* adaptation to apply without specifying *how* to apply them.

By being able to focus on the adaptation logic, developers have been able to increase its complexity often with the support of various additional technologies. Although typically some form of rule processing is employed, leading to rule-based adaptations [SRWE08b, SRWE08a] in which implemented rules define, at least in part, the application behavior. These rules are typically specified in terms of logical predicates over variables representing context readings [CEM03a, SRa]. Alternatively, in some cases the presence of context variables can be represented better through probability-based predicates, in which case stochastic models are used [ZLS^{+}08, KKKM06].

The next evolutionary step, to which we are assisting now, is the introduction of a *ConfigurationManager*. At runtime the CAAA is fed with a configuration file, which could be standardized and therefore completely application independent. This configuration file is used by the *ConfigurationManager* to generate at runtime both *ContextManager* and *Adaptation Manager*. Such *ConfigurationManager* could be either a component of a certain CAAA or part of a context-awareness middleware. CAAAs developer provides GUI and tools to help the end user in creating its own configuration file. At runtime the architecture of such novel CAAAs is the same as the one depicted in Figure 3.1, however their implementation is quite different and it does not provide a static implementation of *ContextManager* and *Adaptation Manager*.

3.1.1 Views on the Context

As depicted in Figure 3.1, the layered architecture of CAAAs gives rise to four different views of the context—the *physical context*, the *sensed context*, the *inferred context* and the *presumed context*. At any given point during the execution of a CAAA, all four of these views may differ from each other.

Starting from the external environment, which embodies the *physical context*, the Context Manager loads environmental information by using available sensors. Each sensor works independently from the others, and they sense context asynchronously with respect to each other, possibly via multiple threads. In this way the Context Manager generates the *sensed context*, which is a discretization of the physical context created by multiple sensors at different times.

Asynchronously with respect to the sensing thread(s), the Adaptation Manager uses the sensed context to determine when to perform an adaptation. Such decisions are made based on the evaluation of a set of *adaptation predicates*. Generally, these are Boolean predicates whose evaluation produces the *inferred context*, representing a set of higher-level concepts inferred about the external environment. Once any relevant predicates are satisfied, an adaptation is triggered, and the application behavior changes to match the *presumed context*, which represents a high level, application-oriented view of the external environment. In some applications the presumed context simply corresponds to an implicit, conceptual view represented by a particular configuration or composition of components. In others it is realized explicitly through particular state variables or parameters.

Ideally, the physical context and presumed context always should be consistent with each other and represent the same situation. However, this may not happen for several different reasons, depending on the sequences of assignments and accesses to context variables between the different layers. An inconsistency between the physical context and presumed context may be caused by a *fault*, which itself may lead to a *failure* of the CAAA. The kinds of faults that may arise, and thus the kinds of failures that may occur, are determined in part by the choice of adaptation technology, as we show in this thesis.

3.1.2 An Example of Multiple Views of the Context

Before going into more detail and examining variables in each layer of the architecture, we clarify the idea of multiple views of the context with an example.

Consider a mobile CAAA that redirects incoming calls from a mobile phone to a Bluetooth hands-free system when the user is driving. Assume the physical context to be

The user is in a car equipped with a Bluetooth hands-free system.

The sensed context, assuming that the CAAA would be using Bluetooth, would be something like

Bluetooth device 00:01:E4:AC:34:71 was detected.

This information is then evaluated and will produce the following inferred context:

It is true that the car's hands-free system is in range.

Using this inferred context, an adaptation will be performed, the application will connect to the hands-free system using a pre-established key, and then the presumed context will become

The user is driving, and incoming calls are diverted to the hands-free.

This adaptation has been successful, and the user, without any interaction, is now able to use the phone hands-free.

Even for a simple scenario like this, CAAA adaptation can become complex. For example, as described, the sensed context does not explicitly specify if the user is the driver. That information is presumed by the fact that the hands-free system was detected. To avoid misinterpretation of the context and to constrain the adaptation, the application would require additional sensors capable of identifying whether the user is driving, or additional logic to detect the presence of multiple people in the car.

3.1.3 Propagation of a Context Change through the Layers

The layered architecture has an impact on how variables representing contextual information are processed. In particular, to generate the different views of the context, each layer of the architecture handles a set of variables and converts context values to a form appropriate for its needs. Figure 3.3 depicts the life-cycle of a context variable as it is converted to different forms in the layers.[1] Each triangle at the bottom of the pyramid typically represents a context variable, continuously refreshed. Those variables

[1]Note that the three levels of Figure 3.3 correspond to the top three layers of Figure 3.1. The physical context has no representation in terms of variables because it represents the real environment, external to the system.

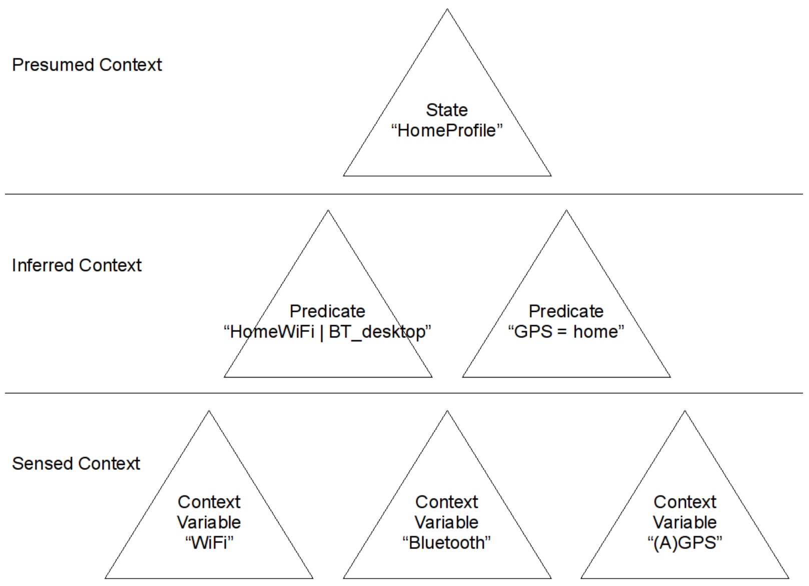

Figure 3.3: Propagation of a Change in the Context.

are combined into context predicates, which is represented by the second level of the pyramid. Context predicates are evaluated to adapt in a configuration, depicted in the top of the pyramid, which is suitable for the current presumed context.

Refreshing the sensed context. At the bottom of the propagation pyramid are context variables read directly from sensors. These variables can have multiple formats, and their structure depends on how complex the context is that they represent. For instance, time can be used to order events sequentially, in which case it can be represented simply by an integer. But if time is used to identify a specific date and time, then a data structure such as a *GregorianCalendar* object is needed.

Regardless of their type, sensed variables must be *refreshed*. Middleware support for context refreshing comes in the form of two common *context handler* interfaces: an asynchronous *call-back* [Pro06a, Ope08a, Jav00] and a synchronous *get* [Ope08b]. In both cases, the middleware may require negotiating the precision of the retrieved value or the refresh rate. For instance, the Location API of J2ME [Pro06a] requires negotiating the precision of the retrieved location as well as the frequency of updates and will throw an exception if the required precision is too high. Note that this sensing activity is not synchronized and that refresh rates can vary dramatically. For instance, in the Java Platform Micro Edition [Sun09], the system clock has a refresh rate of 1 millisecond, the GPS sensor refreshes every few seconds [Pro06a], and the Bluetooth sensor refreshes almost once a minute [Jav00].

Each context variable is refreshed by CAAAs also monitor the context in parallel with their other applicative task and, when a single context handler. Those context handlers work independently of each other and asynchronously suspend, update and loop. This implies that different context variables are refreshed independently even if they are sensing the same phenomenon in the environment, and thus they may be inconsistent with each other.

Computing the inferred context. The second layer of the pyramid contains the context predicates used to compute the inferred context. Each predicate can be evaluated over multiple context variables and is stored in Boolean variables representing the inferred context. There may be many such predicates and Boolean variables used to compute the inferred context, and so it is not necessarily the case that these Boolean variables are fewer in number than the context variables over which they are evaluated. As previously discussed, the implementation of this layer varies across CAAAs and middleware (e.g., rule-based [CEM03a, AMNT08], stochastic [ZLS+08, KKKM06], hybrid [LCT08, WER07c]). Irrespective of the implementation, some representation of the inferred context is needed ultimately to support decision making by the Adaptation Manager.

Given that the predicates in this layer may include multiple context variables, the different refresh rates and choice of interfaces with which those contexts are sensed can lead to the incorrect evaluation of a predicate. This in turn will result in the incorrect inferred context and perhaps an inappropriate adaption. To address this issue, adaptation managers often delay a full predicate evaluation until other conditions are satisfied (e.g., other relevant variables are refreshed within a threshold, sensed variables lead to consistent state), sacrificing adaptation performance for stability [SRWE08a, LCT08].

Adapting and setting the presumed context. Once a predicate or a set of predicates is evaluated and triggers an adaptation, the CAAA adapts to match the environment by generating a presumed context (as shown in top level of the pyramid). The form of these top-level contextual variables again depends on the implementation. They can be state variables or more complex configuration of components aiming to achieve a certain task (e.g., to connect a mobile device to the car's hands-free system through Bluetooth).

3.2 Related Work

Several works in the literature propose context-awareness middlewares to support and simplify the implementation of the internal logic. Mainly in those middlewares the internal logic is rule-based or state-based, and it is dynamically loaded from a configuration file or from a database. Among those, Carisma [CEM03b] and ContextNotifier [SR07] loads trees of logic gates which are used to filter the context and activate a trigger.

In the attempt of solving the problem of inconsistent data reading certain middlewares provides additional contextual information acquiring them from external sources which can be both local or distributed. Sensay [SSF+03], connects to the application an external programmable box with multiple sensors which performs real time computation on the received data.

Similarly certain middlewares apply a sensor-server-client architectural model, which is known as *middleware centric context aware model.* This model has been broadly adopted for industrial monitoring in which clients receives on demand only a small amount of the computed data while the server is monitoring the whole factory. The benefit of this architecture is that the server will continuously compute the context independently from the clients allowing them to go in standby to spare battery. The drawback is that clients disconnected from the server do not receive updates from the context. Nokia research applied the idea of a centralized context-aware middleware and developed its own middleware (MUPE) [RS04] and a script language for context specification (CSP) [Lak03]. CEP is an XML-based language trying to formalize context exchange between a client and a server. Base element for CEP are logical connectors, atomic values and script actions.

Certain middlewares implements a bidirectional communication allowing the server to receive part of the context directly from its multiple clients also allowing their interaction. ContextPhone [ROPT05] uses an online jabber server to store users' status, and a Symbian OS implementation for smartphone as stand-alone client. Users are instantaneously notified by their buddies status, location and other contextual information. Once again clients are completely dependent on the server, strictly requiring a constant communication which drains the battery.

Chapter 4

Taxonomy of Faults in CAAAs

During the course of this research work we came across several failures in the CAAAs that we were examining. In addition to traditional failures CAAAs suffer from peculiar failures caused by faults in their adaptive and context-aware components. These peculiar failures are hard to replicate and it is difficult to describe why they can happen. The existing bug-tracking tools are often almost worthless in classifying them, reducing error reports to mere verbal descriptions. These descriptions vary from simple descriptions of the effect, such as "this is supposed to happen but sometimes it does not", to more detailed instructions of how to reproduce the failure such us "run the application, activate the GPS, turn the application off and on again, then re-activate the GPS".

In Chapter 3 we defined an architecture capable of describing CAAAs' adaptive mechanisms. We observed that describing CAAAs' failures in terms of layers of such an architecture was helping us isolate and fix the causing fault. Starting from this intuition we felt that it was possible to provide quality assurances with instruments to formally classify faults and failures.

In this chapter, in Section 4.1 we describe seven CAAAs which we have examined during our research and we present eleven unique context-aware and adaptive faults affecting them. In Section 4.2 we define a taxonomy capable of classifying such faults over multiple dimensions. We conclude the chapter by classifying the faults we presented with our taxonomy and by showing how such classification, not only can help in identifying the faults but also may underline weakness or vulnerability in the whole application.

4.1 Faults in the CAAAs We Studied

In this section we explore, in addition to PhoneAdapter which was introduced in Section 4.1.1, six other CAAAs utilizing different middleware and a wide range of context information, and we describe some of the faults and failures we have observed associated with their architectural layering and the incorrect propagation of context information. We then conclude the chapter by presenting a taxonomy of CAAA faults and failures that represents a preliminary attempt to synthesize our experience with these four applications, and we apply the taxonomy to the faults and failures we observed. Table 4.1 presents a summary of the CAAAs we have studied, including a description of the context and underlying middleware used.

These artifacts have been selected in order to give the reader a complete view of which faults may happen and how they are perceived by the end user. PhoneAdapter and GPS-Recorder have been implemented during the course of this research as running examples and their faults will be examined in details the following chapters. TourApp is well known application used as a benchmark for context-awareness testing. TourApp has been included to show how known faults detected by other researches fits in our taxonomy. The conveyor belt example has been included because it is the running example of closely related work and also because it is a good example of a CAAA not build upon a context-awareness middleware [LCT08]. PowerManager and Timeriffic have been included as examples of commercial and open-source applications. These represent the state of the art for this kind of applications. SeNIE is an example of a CAAA implemented with a neural network instead of a rule-based logic. SeNIE ha been included to that the taxonomy can be applied independently of the technology used to implement the adaptation logic.

Name	Description	Middleware	Context Data
PhoneAdapter [SRWE08a]	An application for adjusting a mobile phone profile according to the user's activity	ContextNotifier [SRa]	Bluetooth, location, speed, calendar
TourApp [WER07c]	An application for adjusting PDAs information as a user tours a facility	Context Toolkit [SDA99]	Location using beacons, battery
SeNIE [SPF$^+$06]	An application for identifying hand gestures from a glove used with a body-area sensor network	SeNIE	Bend sensors and a tri-axial accelerometer
Conveyor Belt [LCT08]	An application for locating packages moving through a conveyor belt	Not specified	Four ordered RFIDs
PowerManager [pow]	An Android application and service proactively adapting the system settings to spare battery	Android APIs [All09]	All the sensors available to Android
Timeriffic [tim]	An simple Android application muting and unmuting the device	Android APIs	The internal system clock
GPS-Recorder (See Section 4.1.7)	An application to record trekking traces	Not implemented	Location and WiFi beacons

Table 4.1: Examined Artifacts

4.1.1 PhoneAdapter

Case 1. Consider two of the profiles defined by *PhoneAdapter*, *Home* and *Office*. The former is meant to be applied automatically whenever the application detects that the user is at home, while the latter is meant to be applied automatically whenever the application detects that the user is in his office. The application uses GPS to infer when the user is at home, and it uses Bluetooth to discover the user's office PC, from which it infers that the user is at work. The necessary *physical context* is first sensed by GPS and Bluetooth sensors on the phone and fed to the middleware's Context Manager, which updates context variables used by the CAAA's adaptation rules specified for the *Home* and *Office* profiles. This *sensed context* stored in the context variables is used by the Adaptation Manager to evaluate the predicates of the adaptation rules. The evaluation of predicates results in *inferred contexts* that determine which rule or rules to trigger. The triggering of a rule initiates a chain of adaptive behaviors in the CAAA, causing the CAAA to operate in a new *presumed context*, which is revealed by the selection of a new profile.

In the *PhoneAdapter*, we discovered an unanticipated fault related to these profiles that arises because of an unforeseen form of mobility, leading to an inconsistency between the physical context, the sensed context and the inferred context. In particular, if the user's work PC is a laptop, then it is very convenient for the user to bring his laptop home. In such situations, the true physical context of the user is his home location, but the Context Manager senses both the home location and the work PC. This leads the Adaptation Manager to infer simultaneously the existence of two different contexts, leading to the simultaneous triggering of the rules for both contexts and thus to a nondeterministic setting of a new profile.

Case 2. Consider two other profiles defined by *PhoneAdapter*, *Jogging* and *Driving*. The former is meant to be applied automatically whenever the application detects that the user is jogging, while the latter is meant to be applied automatically whenever the application detects that the user is driving in a car. The application uses GPS to infer user location and user mobility, and it uses Bluetooth to detect the hands-free system in the user's car. In our implementation of *PhoneAdapter*, we discovered another unanticipated fault related to these profiles that arises because of differences in the refresh rates of the GPS and Bluetooth sensors. Suppose the user leaves his home, enters his car and starts driving. The physical context of the user is thus the car in which he is driving. But due to the differences in refresh rates, the Context Manager senses that the user has left his home location and is moving rapidly. This causes the Adaptation Manager to infer that the user is jogging, thereby causing the application to apply the *Jogging* profile as the new presumed context. Eventually the Context Manager detects the car's hands-free system via Bluetooth, but by then it is too late to apply the *Driving* profile, because the adaptation rules are defined in such a way that the *Driving* rule cannot trigger when the *Jogging* profile is active.

Case 3. *PhoneAdapter* has a second driving profile, *Fast Driving*, which is applied when the application detects, again via the GPS sensor, that the user is driving at high speed. We observed that *PhoneAdapter* failed to adapt to this profile when the user started driving and accelerated quickly. The problem was that the rapid acceleration prevented the middleware from propagating the information necessary to apply the *Driving* profile first, which is a necessary condition for *Fast Driving* to be applied. As a result, *PhoneAdapter* allows the phone to accept calls while driving at high speeds.

Case 4. Consider one more profile, *Meeting*. When the *Office* profile is active, the *Meeting* profile is meant to be applied whenever the application infers that the user is in a meeting, based on the user's calendar and the phone's clock indicating the presence of a scheduled meeting time, and on Bluetooth discovering another person for a meeting. Our initial implementation of *PhoneAdapter*, however, fell into an adaptation cycle between the *Office* and the *Meeting* profiles since both of their conditions are triggered whenever a meeting is held in the office, leading to inconsistencies between the sensed and inferred contexts.

Architectural considerations. Faults such as the one described in Case 1 happen when multiple adaptations interfere with each other. If identified, these faults can be fixed by redefining the interfering predicates. Detecting all of them can be difficult and time-consuming. Rather than spotting and fixing single interferences, a design which does not allow them should have been used.

Nondeterministic adaptations happen in *PhoneAdapter* because, unlike other applications such as Locale [Two09], it supports adaptation rules with the same priority. Such rules can interfere with each other, creating multiple inconsistent inferred contexts. A strong ordering of rules helps prevent nondeterministic adaptations but makes them less flexible and may just move the problem from the inferred context to the presumed context. In Case 1, for instance, giving the adaptation to *Office* higher priority will cause the *Office* profile to be applied erroneously when the user is at home. Lowering its priority will produce the opposite scenario when the user brings his personal laptop to his office. The fault in Case 1 can be solved by giving more importance to those contextual information which are certain and less to those which are assumed. For instance, when the application is trying to identify the location, GPS readings could have a priority higher than non-GPS predicates that use other context variables to infer the location (such as Bluetooth discovery of location-specific devices).

PhoneAdapter reacts to a context change immediately upon a change to some context variable, which can be independent of some situation of interest to the application. The time it takes to detect the change can vary depending on the internal refresh rate of the underlying middleware, making the reaction itself too impulsive in some instances and too slow in others. Occurrences of faults as the ones described in Cases 2 and 3 could have been, at least partially, prevented by maximizing the sensors refresh rate and by reacting instantaneously only when the applications logic demands it. For instance the profile Driving Fast, which in the applicative domain requires a quick reaction, should be applied (and un-applied) as soon as satisfied. Jogging which instead describes a durative action, should be applied only if the trigger condition remains satisfied for a certain amount of time.

4.1.2 TourApp

TourApp is a context-aware application that runs on mobile devices of visitors attending an exhibition to notify them of presentations of interests. *TourApp* was distributed originally with the Context Toolkit [SDA99] and has become a common demonstration application for context-awareness middleware.

Context sensing is supported by different *Widgets*, each running on a router in some room of the exhibition. As the user walks toward a registration desk, the application is supposed to sense a *Registration Widget*, which stores the user's preferences and determines which information the user will receive. As the user enters different presentation rooms, the application will receive information (e.g., presentation title, duration, speaker bio) from a set of *Presentation Widget*s. As the battery becomes critically low during the visit, the application will enter a power saving mode, reducing the quantity of displayed information.

Case 5. Consider a user launching the application at the registration desk, but within range of two different routers—the one in the registration room and the one in the first presentation room. As the device has no control over which Widget will be sensed first, the *Presentation 1 Widget* running on the latter router may be sensed before the *Registration Widget* running on the former router. In that case, the inferred context will become *"It is true that the user is in presentation room 1,"* and the application will assume that the user decided to skip the configuration phase at registration and will adapt to *tour mode*, loading the information about a presentation the user did not intend to visit. What is worse, registration becomes unavailable once the application is in *tour mode*.

Architectural considerations. *TourApp* infers the user's activity from the user's location, which if sensed incorrectly can propagate a fault to higher layers. However, the expected sequence of user actions is known from the application domain. *TourApp* could use this knowledge to prevent faults by applying a more restrictive predicate for adaptations that deviate from the expected route. For instance, requiring each user to configure his device in the registration room and preventing adaptations to *tour mode* for un-configured devices could help to prevent the fault described in Case 5.

4.1.3 SeNIE

SeNIE [SPF+06] is a toolkit for reading and computing streams of data from a connected body area sensor network [HPB+09]. A user, wearing several sensor nodes, transmits data about his movements to *SeNIE*, which computes them on the fly and notifies external components. One of *SeNIE*'s key features is provided by a gesture recognition component. Possible gestures include a closed fist, an open palm, and some intermediate gestures with a combination of closed and open fingers such as “thumbs up”, “on the phone” (thumb and little finger) and “music” (thumb, index finger and little finger).

Case 6. When a user is opening and closing his hand, passing from the closed fist to the open palm, the sensed context contains a sequence composed of the initial gesture, one or more transitional gestures, and the final gesture. The training of the gesture classifier did not consider such transitions. As a result, when a transitional gesture matched one of the predefined gestures in the library (e.g,, thumbs are up in the transition from closed to open palm), an intermediate configuration may be incorrectly applied and an incorrect context presumed.

Architectural considerations. The inferred and presumed contexts of *SeNIE* do not model dynamic conditions. The lack of inferred contexts defining the transitions between two configurations, such as “closing palm”, or even the more general “changing gesture”, cause the output produced by the gesture recognition to be incorrect. Improving the granularity of the situations modeled by the inferred context would mitigate the fault described in Case 6.

4.1.4 Conveyor Belt

This application was conceived by Lu et al. and used to illustrate the effectiveness and efficiency of their *Context Inconsistency Resolution (CIR)* techniques [LCT08]. The scenario consists of a sequence of RFIDs placed along a conveyor belt to identify the position of a moving package in order to manipulate it precisely. When we map this application to our architectural model, the context management layer reads all the RFIDs, and the adaptation logic simply uses their *estimate_position* function to infer the position of the package, which it does by reading the strength of the signal received by each RFID. Once the package's position on the conveyor belt has been determined, the program unit relevant to that location is started.

Case 7. During the computation, a package should move along the conveyor past a number of position readers. The application reads the signal strength of all the readers, generating a sensed context such as *“[strength0, strength1, strength2, strength3]”*, which may be computed as an inferred context *"position == reader1 is true"*, and which in turn may trigger an adaptation to a presumed context *"executing program unit1"*. Thus, the application can adapt quickly to a new presumed context as soon as it has been computed by the adaptation logic, but this makes the application highly reactive but also vulnerable to noise. In particular, a valid sequence of inferred context would correspond to a package making forward progress along the conveyor. However, during the execution it may happen that, due to noise in the sensed context, the Adaptation Manager computes a sequence of inferred contexts indicating that a package has reversed course and is traversing backwards, which is impossible. In such a case, the application may erroneously adapt to a presumed context such as *"executing program unit0"*, causing operation on an empty position of the conveyor. Similarly, it may happen that the Adaptation manager computes a sequence of inferred contexts indicating that a package has skipped some stages and then has reappeared at a later position.

Architectural considerations. Similarly to *TourApp*, *Conveyor Belt* identifies package position from sensor readings and has an expected sequence of positions through which the package should move. Real world constraints, such as the direction in which the belt is moving, the last correct known position and the maximum belt speed can be used to correct or discard faulty readings, preventing them from propagating beyond the sensed context [LCT08].

4.1.5 Power Manager

Power Manager is a commercial Android application available from the Android Marketplace. Out of the box, *Power Manager* maximizes the battery duration by turning on and off GPS, Bluetooth and Wifi, and by regulating the screen brightness according to the current battery level [pow]. The basic idea is to turn off each sensor when the battery is running low and to turn them on again when the battery is full or while the phone is recharging using the AC power or USB.

Like most CAAAs, *Power Manager* allows users to configure its adaptation logic as they please. The one useful for our studies included five profiles:

1. *Initial*: the initial profile when the application starts,
2. *ChargingAC*: the phone is charging using an AC charger,
3. *ChargingUSB*: the phone is charging using a USB cable,
4. *OnBattery*: the phone is running on battery and the battery charge is above 30%,
5. *BatteryLow*: the phone is running on battery and the battery charge is under 30%,

In this custom configuration, adaptations to *BatteryLow* turn off Bluetooth, GPS and WiFi to extend the battery life. Adaptations to *ChargingUSB* and *ChargingAC*, instead, are defined with the associated actions of turning on all the sensors, since the phone can use them without exhausting the battery, which is on charge. Adaptations to *OnBattery* do not perform any action since the battery is full and the phone can be used as it was configured.

Case 8. While we were running it on an Android G1 phone with a custom configuration we found that the battery duration was reduced to one half of the normal duration. As we tried to discover what was causing such a malfunction we observed that recharging the phone also activates all the sensors, which are then left active while the device switches to *OnBattery*. Indeed the *OnBattery* configuration does not perform any corrective action when applied. As a consequence 70% battery was quickly drained forcing the device to switch to *BatteryLow* after a couple of hours.

Architectural considerations. Indeed the fault was caused by the configuration we applied. However *Power Manager* asks the user to specify for each profile, a corrective action to be performed when a profile is applied. If we assume that the current configuration would be unknown, as it could be after several adaptations, the definition of such corrective action is fault prone. Defining the configuration of each profile independently from the others would have prevented this fault from happening.

4.1.6 Timeriffic

Timeriffic is a simple application aimed at muting the cellphone at night and un-muting it during the day [tim].

Case 9. In the initial stages of its development, *Timeriffic* was affected by a fault causing the phone to be muted during the day instead of at night. The fault was caused by a bug in the code in which a developer inverted the actions of the adaptation rules of day and night mode. The bug was introduced in revision 7 and fixed over two months later in revision 86. The faulty logic was hard-coded and the bug was hard to be spotted. In the corrected version the developers removed completely the hard-coded logic and introduced an XML configuration file.

Architectural considerations. Created originally to be a very simple yet useful background service, *Timeriffic* was initially implemented as an hard-coded rule-based application without taking advantage of any existing rule-based middleware. Underestimating the complexity of the two adaptation rules has introduced a very resilient fault jeopardizing the whole adaptation logic. In this sense *Timeriffic* is living proof that even a simple set of rules can be error prone. After moving the whole logic to a configuration file, developers have been able to make *Timeriffic* much more configurable and user-friendly.

4.1.7 GPS-Recording

GPS-Recording is a very simple application that is developed to evaluate our PDDL-based approach. Essentially, this is a simple GPS-based trekking tour recording application. Tourists can rent from a base camp a GPS-enabled device on which the recording application is running. When the application starts, it enables the GPS and starts reading the current location. As long as the user remains in the base camp *GPS-Recording* waits for the tour to start by entering in a stand-by low-consumption mode. As soon as the users leaves the base camp, *GPS-Recording* starts collecting GPS information, showing their location on a map, and recording statistics about the route, including position, altitudes and speed. At the end of the tour results can be uploaded and stored. A distracted user may start the application when he is already on route. In that case *GPS-Recording* will start recording immediately.

Case 10. Imagine a user starting *GPS-Recording* from the base camp. The application starts, turns on the GPS, and immediately starts collecting data as if the user has already left the base camp. The implementation states that if ”GPS is enabled and if location is base camp” the application should wait in standby mode. Instead, due to the asynchronous nature of the updates, the standby mode is skipped and the recording starts. The fault can be explained as follows. When the GPS is turned on there is a gap of time in which the sensor is on but in which the location is not available. If the GPS is turned off, *GPS-Recording* enables it on start-up right before evaluating the context. When that happens, even if the user is physically located in the base camp, the standby mode is not applied because the location is ”unavailable” and not “base camp”.

Case 11. Similarly to *case 10*, if the user is in the base camp, with the application correctly in stand by mode, sporadically it may happen that the application starts recording while the user is still in the camp. Once again, if the location becomes “unavailable”, such as when the user goes inside a thick building in the base camp, the condition “location == base camp” is not satisfied any more and the application assumes that the trekking trip has started.

Architectural considerations. In most of the frameworks GPS readings have a non-trivial data structure providing several interconnected information, including geo-location, altitude, speed, movement direction, time, precision of the reading, number of available satellites and their signal strength. All this information makes GPS a very powerful data source. However, certain GPS readings are not always available or correct. Applications heavily relying on GPS readings should be aware of such weaknesses and rely on some high-level error correction/prevention.

4.2 Taxonomy

To gain a better understanding of the types of faults occurring in CAAAs, their frequency, and their impact, we have studied several failures and the faults they induce. Based on this experience, we are able to delineate a taxonomy of faults and their corresponding failures that arise due to the peculiar nature of CAAAs. This taxonomy is open to future improvements as the number of CAAAs grow and more data become available.

Our previous work focused on rule-based adaptation in CAAAs and the faults that arise through the use of rule-based context-awareness middleware. We distinguished between two broad classes of faults, behavioral faults and hazards. The former are faults in the internal logic of the set of rules used to control adaptation, while the latter are faults due to the asynchronous updating of context variables, and the different rates at which those updates are performed as readings are taken from context sensors. The taxonomy we present here is meant to subsume our previous categorization.

The taxonomy classifies faults and failures in CAAAs along seven dimensions. The dimensions characterize those aspects of faults and failures that appear to be most important for their detection and elimination by engineers. In fact, the taxonomy could be be integrated into error-tracking systems such as Bugzilla [Ind09], allowing developers to describe faults in an application systematically. As the description of a fault is fleshed out, it can be compared to similar previous faults in order to help further with the localization and elimination of the fault. In addition, as the number of discovered failures and fixed faults increases over time, a statistical profile of the application and its use of context layers can be developed, allowing engineers to focus their attention on the most fault-prone layers of the application architecture. Table 4.2 summarizes the dimensions of the taxonomy and provides a labeling scheme that we use below to classify the faults and failures described previously in this Chapter.

In the first dimension, different faults arise in different architectural layers, and their effects typically are realized as failures in higher layers (row I in Table 4.2). The physical context is taken to be correct

Dimension Number	Dimension Name	Observed Values	
I	Source Layer	S:	Sensed
		I:	Inferred
		P:	Presumed
II	Observation Layer	S:	Sensed
		I:	Inferred
		P:	Presumed
III	Severity	A:	Annoyance
		N:	No Output
		O:	Incorrect Output
		T:	Termination
		U:	Unsafe Operation
IV	Recoverability	R:	Implicit from Refresh
		D:	Direct
		P:	Possible/Eventual
		I:	Impossible
V	Origin	H:	Human
		W:	Sensor Wear
		S:	Sensor Deployment
		D:	Design
		L:	Sensing Libraries
VI	Method of Reproduction	I:	Input
		T:	Timing
		E:	Execution Path
		P:	Physical
VII	Root Cause	[*case specific*]	

Table 4.2: Preliminary Taxonomy of Faults and Failures in CAAAs

since it represents the real environment. Identifying the source layer of a fault is a necessary step in pointing a developer to the faulty component.

Second, and related to the first, different faults have different degrees of observability in different layers, with some faults having high observability (leading directly to clearly obvious failures) and others having low observability (leading indirectly to more subtle failures). Furthermore, faults tend to be detected in layers other than the ones in which they occur (row II). Understanding in which layers a fault manifests itself as observable misbehavior helps developers in pinpointing the source layer and helps testers to understand if a fault has been fixed or not.

Third, the failures resulting from faults have different severities, from a mild annoyance to the user, to an incorrect output or no output at all, to full termination and even unsafe operation (row III). Furthermore, while propagating from the layer in which it occurs to a higher layer, a fault can increase its severity, or can be mitigated or even fixed if the system incorporates some form of error prevention/recovery.

Fourth, and related to the third, different faults entail different degrees of difficulty in recovering from the failures they induce (row IV), ranging from implicit recovery via refresh of underlying sensors to direct recovery via error recovery mechanisms. Some cases lie between these extremes when certain sequences of events allow for recovery, and still other cases are not recoverable at all. Classifying both the severity and recoverability of a fault helps prioritize the work in eliminating faults.

Fifth, different faults have different origins, including the "wear and tear" of physical sensors, the manner of sensor deployment, inconsistencies between application libraries used within a middleware or CAAA (with each library itself behaving in a consistent manner in isolation), or misunderstandings or unforeseen circumstances on the part of the application developer (row V). Identifying the origin of a fault helps developers to fix the fault. Furthermore, keeping track of the origin of several faults helps to reveal systemic weaknesses in the CAAA.

Sixth, different failures can be reproduced with varying degrees of effort, depending on the faults that induce them (row VI). Reproducing a failure may require certain inputs, or the occurrence of certain timing conditions, thread schedules, or physical conditions in the environment, or some combination of these. As in any system exhibiting asynchrony and nondeterminism, faults (particularly hazards) that arise in the presence of such features are particularly difficult to reproduce and typically require the use of runtime instrumentation or simulation. Understanding the method of reproducing a failure associated with a fault helps testers improve their testing to report new faults and to mark existing ones as fixed. The origin and method of reproduction of a fault are coupled with the source and observation layers, respectively. The former focus on the misbehavior itself, while the latter focus on the context associated with the misbehavior. These combinations are synergistic, for instance allowing a developer to classify a fault as being a design issue with the inferred context, whose associated failure is reproducible with a certain input value and observable in the presumed context.

Seventh, we can identify more precisely the root causes of faults in terms of the particular design decisions and/or technology choices involved in producing them (row VII). The root cause thus specifies application-specific information that further helps developers in locating and addressing the fault.

Table 4.3 classifies the eleven failure cases described in previous sections according to the dimen-

	Taxonomy Dimensions						
Case	I	II	III	IV	V	VI	VII
1	I	P	O	P	H	IT	Incomplete Rule Logic
2	S	P	U	P	L	ITP	Inconsistent Sensing Rates
3	S	IP	U	P	L	ITP	Slow Sensing
4	I	P	A	I	H	IT	Problematic Rule Logic
5	S	IP	O	I	S	IE	Overlapping Sensor Fields
6	I	IP	O	R	D	IT	Granularity Mismatch
7	S	IP	O	D	SW	IT	Sensor Noise
8	P	P	A	P	D	PE	Faulty Rule Logic
9	I	IP	O	P	D	IT	Faulty Rule Logic
10	S	IP	O	I	LWD	IT	Sensor Activation
11	S	IP	O	I	S	IT	Sensor Noise

Table 4.3: Classification of the Failures Observed in the Studied Applications

sions outlined above. As shown in columns I and II of Table 4.3, only three of the eleven failures, Case 6, Case 8 and Case 9 were detected in the same layer as the faults that caused them. Of the remaining eight, seven were detected in the layer immediately above, possibly because that layer produced an incorrect computed output. Only in Case 2 did the fault propagate through multiple layers, from the sensed context to the presumed context. Of the six faults arising in the sensed context, none could be observed directly in the sensed context. This simply could be due to the examples we considered, but it also seems that faults are more easily detected in contexts at a higher level of abstraction, because the sensing libraries do not expose enough information to detect them directly.

As shown in columns II and III, there appears to be no correlation between the severity of a fault and its observability. For instance, severe faults exposing the user to dangers such as Cases 2 and 3 were not observed in the layers in which they arose.

If we compare columns V and VII, we can see that problems with sensors and sensing libraries have a direct impact within the sensed context. Developers should be aware of this and should find ways of preventing faults in the sensed context from propagating to higher layers. Faults in generating the inferred or the presumed context seem to be related to weak design or increased complexity.

Finally, as shown in column VI, all eleven cases require significant effort to reproduce, requiring not only a certain sequence of inputs but also certain timing conditions, execution flows or even the presence of certain physical conditions, in order to "fool" the sensing libraries into reproducing a failure.

As the discussion above shows, we can see that the taxonomy provides developers with a way to identify the architectural layers involved in an observed failure. The taxonomy also highlights several opportunities to better support the development of more reliable CAAAs. This support can be provided at different layers of the architecture as well, ranging from support for sensor builders to library developers to rule designers. Support for checking the space of behaviors defined by a CAAA's governing rules and for verifying the sensed and presumed contexts for consistency could address some of the root causes we have observed.

As we expect that context-awareness middlewares will handle both the context reading and the adaptation mechanisms, we believe that techniques for testing CAAAs should focus in the adaptation logic or in the configuration file provided by the user. Accordingly, the next chapters describe the classes of techniques that can be used to detect faults in the application logic or in its configuration and to mitigate their impact.

4.2.1 Benefits of Using a Context-Aware Taxonomy

Our taxonomy has been defined to help developers to improve the quality of their CAAAs. It is important to underline that users will probably not be able to report failures by filling all the dimensions of this taxonomy and that in most of the cases the developers will have to fill the missing fields themselves. However by investing time in classifying all the reported failures developers will have two benefits: (1) this taxonomy will help them in speed up the correction by pointing them in the right direction and (2) by providing a series of well documented statistical information on which component of the application is more fault prone and how it fails.

Imagine a software house with various developers each one of them responsible for a different component. What normally happens with a bug tracking tool is that a random developer starts addressing a reported failure and once the existence of a fault has been verified the correction is delegated to the developer responsible for the component in which the fault was generated. The delegation from a developer to the next one normally includes a textual description of the progress made by the developer currently working on the fault. Such description includes information such as how to replicate the failure and where the fault is located. Unfortunately such descriptions are often incomplete or even missing slowing down the correction process. Our taxonomy provides a standard way of reporting useful information which assist the developers by telling them exactly how to classify each reported failures.

Moreover, after a certain number of faults have been classified with our taxonomy, it is possible to statistically analyze the reported data in order to find which component are most fault prone and which ones are the most common causes for a failure. Such statistical reports can be used by a software house to prioritize the future development phases of their application.

4.3 Related Work

The literature contains several studies on how to classify and isolate faults in CAAAs. Each one of such works focuses on a different aspect of the context-aware and adaptive mechanisms.

During each adaptation the application evolves. This evolution is not immediate, since there may be a transitory in which the application is waiting for certain components to load. While performing this transition the application may fail to react consistently to further inputs. Predicting and validating the application's behavior during this transitory is not trivial. A proposed solution solution is to freeze the computation during the whole adaptation [ZC06]. Unfortunately such a design model, which is based on Petri nets, does not fit with systems which cannot afford to suspend their services. Another approach proposes to use a rule based system as a consumer for context aware events and using a sequential producer/consumer mechanism to buffer requests during the adaptation [CFL03]. This solution could be successfully applied to our architectural model by introducing a queue between the context manager and the application logic.

Concurrency is one of the main sources of faults in context-aware applications. These faults are generated by interference between threads reading sensors, adapting and performing high/level tasks. One of the most relevant works in the field of testing concurrency is the one by Long et al. [LHS03] that uses a formal language to force the execution of each possible combination of threads within a monitor. This approach is effective in discovering deadlocks, starvation and interferences because for each monitor makes it possible to test different predetermined situations. This approach is not effective to validate concurrent faults in CAAAs because faults are not related to monitors. However the same idea could be reused and applied to context aware program points and to adaptive program points [WER07b].

For application domains, in which applications adopt a certain architecture, domain-specific fault taxonomies have been defined. Bruning et al. have designed a fault taxonomy for Service-Oriented Architecture (SOA) [BWM07]. The approach they used is similar to ours. We classified faults with seven independent parameters including in which component the fault occurs and its severity. Bruning instead organizes faults in a tree in which to classify a fault engineers have to navigate from the root to a leaf. All the root children represent the different stage of SOA services (publishing, discovery, composition, binding, execution) which is similar to the source layer in our taxonomy. All nodes at depth two or more represent the fault origin. All the faults described by Bruning are blocking, therefore severity, recoverability and observation layer are not mentioned. Bruning does not consider those faults in which the composition and the execution terminate successfully but in which, due to a non blocking error, the produced result is faulty. Chan et al. have designed a fault taxonomy for web service composition [CBS+09]. Similarly to ours they isolate all the observed effects and six elementary fault classes identifying the fault origin. By combining observed effects and fault origin they classify faults in thirteen classes of faults divided in three macro groups. Chan does not classify faults in terms of severity and recoverability. It is therefore not possible to specify if an observed fault aborts the computation or if the system will perform some error recovery policy.

Chapter 5

The A-FSM Model

In Chapter 3 we have shown that CAAAs typically are built using adaptation rules triggered by changes in context variables values. Because of the complexity of the rule logic and the asynchronous updating of context variables, it is possible for these rules to embody faults.

An exhaustive testing of such rules would require to run several combinations of execution paths and to validate them against a set of expected results. Enumerating all the possible combinations and knowing their expected values requires a deep knowledge of the application's adaptation logic. Given that most of the implementations are rule-based, such knowledge can be effectively provided by a model representing those rules. Moreover, as also stated in Chapter 3, context-awareness middlewares are more and more taking care of the implementation of standard CAAA components, moving developers' focus to the adaptation logic.

In this chapter, in Section 5.1 we exploit the rule-based design of CAAA adaptation logic to define the *Adaptation Finite-State Machine* or A-FSM, a new finite-state model that supports the analysis of potential faults in the adaptation rules of CAAAs. The A-FSM model also helps testing engineers in validating CAAAs. In Section 5.2 we use the A-FSM to define properties whose violation leads to faults. To conclude, in Section 5.3 we present models extracted from some of the CAAAs that we have used in the research. In Chapter 6 and in the rest of this thesis we present verification and fault detection techniques based on the A-FSM model.

5.1 Formal Definition of A-FSMs

Context-awareness middlewares lend themselves naturally to the derivation of finite-state models from the adaptation rules they support. In this section we present formal definitions for the kind of rules that we described informally in Section 4.1, and then we formally define the A-FSM induced by such rules.

Typically, CAAAs "live" in an environment that can be sensed by means of a set of sensors, such as GPS and Bluetooth receivers, (video)cameras, etc. We denote with $\mathcal{C}$ the set of all the *sensed context variables* that can be sensed by a CAAA. For instance, $\mathcal{C} = \{c_{\text{time}}, c_{\text{gps}}, \dots\}$, where each c_i is a complex structure that includes, for instance, the date, time timezone of the clock and the actual GPS position, the signal strength, etc.

Based on $\mathcal{C}$ we define a set $\mathcal{V}$ of *propositional context variables* (PCVs for short); elements of $\mathcal{V}$ are

Boolean variables $v_1, v_2, \ldots$ used to reason about context variables in $\mathcal{C}$. For instance, a variable v_i can encode the fact that "the Bluetooth device is paired with HomePC", or "the speed obtained from the GPS device is greater than 5 km/h". PCVs may not be independent: for instance if a variable v_i encodes the fact that the speed is greater than 40 km/h, and v_j encodes the fact that the speed is greater than 5 km/h, then v_i implies v_j. We capture these dependencies by means of *Global Constraints*: a Global Constraint is a Boolean formula where such dependencies are made explicit, and we denote with $\mathcal{G}$ the set of all such constraints. A formal definition of these constraints can be found in Section 5.1.2.

The propositional context variables are used to define the *rules* that govern transitions between *states* of a CAAA. We denote with $\mathcal{S}$ the set of the possible states of a CAAA; as an example, possible states of a CAAA are "Silent", "Driving", "Meeting", etc., characterizing, in this case, different *profiles* of a mobile phone application. We denote with $\mathcal{R}$ the set of rules that govern transitions between states. Each rule $R \in \mathcal{R}$ is a tuple $(\texttt{name}, P, S, \texttt{ActionList}, i)$, where `name` is a string identifier, P is is a Boolean predicate over $\mathcal{V}$, $S \in \mathcal{S}$ is the *destination* state, `ActionList` is a list of actions to be performed upon entering S, and i is an integer number denoting the *priority* of a rule (with lower values meaning higher priority). Actions are expressed as a Boolean combination of $v \in \mathcal{V}$ and encode the modifications to the context variables caused by the transition (For instance, a transition may switch off the Bluetooth device, and this is encoded as an action by negating all the $v \in \mathcal{V}$ referring to the Bluetooth; these negations are added to `ActionList`). The predicate P of an action is the "trigger" for the rule, i.e., the transition caused by a rule R can only be performed when P is true. Rules are associated with starting states by means of a function $\mathcal{T} : \mathcal{R} \to 2^{\mathcal{S}}$. If a state S is associated with a rule R by means of $\mathcal{T}$ and the corresponding predicate P is true, then we say that R is *active* in S. If more than one rule is active in a given state, only the one with the lowest priority value is triggered.

For each state in the CAAA we define a (optional) set $Inv(S)$ of *invariants* that must hold true if the CAAA is in state S, expressed as Boolean predicates over PCV. If one of the predicates is not true, then we require a rule to be active in S to enable a transition to a new state.

The description of a CAAA is completed with the definition of an *initial* state $I \in \mathcal{S}$. The tuple $(\mathcal{C}, \mathcal{V}, \mathcal{G}, \mathcal{S}, \mathcal{R}, \mathcal{T}, Inv, I)$ defines an extended finite state machine that we call the Adaptation Finite State Machine (A-FSM). Notice that this is not a pure rule-based formalism due to the presence of the (optional) elements to reason about states (i.e., the sets $\mathcal{G}$ and Inv). However, we introduced these elements because they capture naturally various properties that stakeholders and developers want to make true in a CAAA. Notice that A-FSM models can be obtained automatically by parsing the configuration file used by the CAAA itself or by instrumenting its source code. Since most of the modern CAAAs also provide their users a configuration GUI to specify the desired adaptive behavior at runtime, a model could also be obtained directly from the user configuration.

5.1.1 Priority and Rule Ordering

Multiple rules are often active simultaneously in the same state. If the predicates of two or more of them are satisfied simultaneously then the CAAA will adapt by nondeterministically triggering one of the rules. As we will describe more in detail in Section 5.2.1, if that happens then the adaptation logic is nondeterministic. Rule designers can prevent this from happening by fixing an order in which rules should be applied.

Given two rules A and B, and their trigger predicates P_A and P_B we can force rule B to be satisfied only when rule A is not by assigning to rule B a new trigger predicate P'_B defined as: $P\prime_B = \neg P_A \wedge P_B$. By means of this conversion it would be possible to enforce any possible ordering without adding extra notation to the A-FSM definition. The obtained predicates can be computed very effectively, as the complexity is only introduced in the predicate itself. If needed, the predicates could be further simplified by applying various Boolean techniques (e.g. removing double negations, applying De Morgan's law, removing contradictions and tautologies).

Unfortunately, in the presence of several rules in the same state, even if the computational overhead remains minimal, the obtained predicates may become not human understandable and error prone.

To support rule designers we introduced the concept of *priority* which can be considered as syntactic sugar. In the A-FSM model we associate a *priority* level for each rule which defines the order in which it will be computed by the *Adaptation Logic*.

If each priority level contains at most one rule, rules are ordered with a *strong priority order*, otherwise if multiple rules have the same priority the ordering is called a *weak priority order*. The *A-FSM* model supports both the priority orders transparently and the decision of which one to apply is left to rule designers. None of these two ordering is superior to the other and as we will discuss in Section 5.2, they suffer from different faults.

In our implementation we accept 9 priority levels from 1, the highest, to 9, the lowest, of which 5 is the default one.

5.1.2 Global Constraints

Assignments of values to propositional context variables represent a suitable input for an A-FSM and therefore for its associated CAAA. PCVs derived from different relational expressions over the same sensed context variable are not necessarily independent. For instance, if variable c_1 encodes the fact that the speed is greater than 5 km/h and c_2 that the speed is greater than 50 km/h, then it is not possible for c_2 to be true and c_1 to be false simultaneously.

The A-FSM model allows testing engineers to define additional *global constraints* to eliminate such inconsistent truth assignments; the analysis algorithms then treat such global constraints as additional conjuncts of all rule predicates, effectively reducing the state space that needs to be explored.

In our implementation we support constraints in the form of implications because that was convenient to model the constraints in our case studies. Particularly we found it convenient to impose that, certain PCVs cannot be satisfied (or negated) depending the assignment of other PCVs. For instance if a PCV representing the predicate $gps_{location} ='' home''$ is satisfied, any other PCVs representing the GPS location cannot be satisfied.

Developers may need to extend our implementation with other functions representing more sophisticated or more specific forms of constraints depending on the physical properties affecting the context variables.

5.2 Properties and Fault Patterns

Rule designers or even end users define or add adaptation rules following specific needs or behaviours which they want the CAAA to follow. Even if correctly designed, when introduced in a complex rule space, rules tend to interfere with each other, causing annoyances or failures.

To support rule designers and testers in validating instances of the A-FSM model against common faults, we have defined a set of properties which the model must satisfy. When a model or a set of rules violates one of such properties the implementation, represented by the model, may contain a fault. We categorize faults violating of one of such properties with a unique pattern of fault.

5.2.1 Determinism

CAAAs are by design adaptive, meaning that given a certain input they react differently according to the current state which defines the current behavior and their current configuration. However, even if CAAAs are adaptive their behavior must be deterministic. Given a CAAA and knowing its internal state and its configuration, its reaction to a certain input should be predictable. In other words given a CAAA starting from a known initial state with predetermined initial configuration and given a fixed sequence of input, it should be possible to determine the new state and the new configuration to which the CAAA would adapt as a result of the given input. In applications violating the Determinism property the sequence of adaptations depends on hidden implementation aspects or on uncontrollable physical factors making the final behavior unpredictable. We define the determinism property as follows:

Determinism property: *For each state in the A-FSM and each possible assignment of values to propositional context variables in that state, the Determinism property is respected if there is at most one rule that can be triggered. If the rules of a CAAA violate the Determinism property, we say that the rules contain a* Nondeterministic Activation fault*, a pattern of faults characterized by the presence of multiple active rules in the same state and with the same priority whose predicates can be satisfied by the same set of context updates. This kind of fault happens because unrelated predicates might not be mutually exclusive.*

Nondeterministic Activation fault can be eliminated in three ways: (1) by reformulating the predicates of the affected rules in such a way that at most one is satisfied by the bit string; (2) by splitting the affected state into multiple states, with the affected rules associated with different states; and (3) by assigning different priorities to the affected rules.

As described in Section 5.1.1, a strong priority order forces rules of a single state to be mutually exclusive by design. Therefore it guarantees that the Determinism property is respected.

5.2.2 Rule and State Liveness

In the A-FSM model a rule describes an adaptation from the current state to a more suitable one. Once a rule has be designed there should be at least one assignment of PCVs satisfying its trigger predicate. If none of the possible assignment of PCVs satisfies the rule then its adaptation will never be performed and its action never applied. This is similar to dead or unreachable code in a programming language. Similarly states, with the exception of final states, are supposed to be sustained only until the next adaptation, meaning that in each state there should be at least a satisfiable active rule. We define the liveness property of rules and states as follows:

Rule Liveness property: *For each state in the A-FSM and each of its active rules, there is at least one assignment of values to propositional context variables that satisfies the predicate of the rule. If the rules of a CAAA violate the Rule Liveness property, then we say that the rules contain a* Dead Predicate fault*, a pattern of faults characterized by the presence of an unsatisfiable predicate in the set of active rules of some state.*

State Liveness property: *For each state in the A-FSM, if the state contains any active rules (and thus is not a final state), then at least one of the active rules has a satisfiable predicate. If the rules of a CAAA violate the State Liveness property, then* all *the active rules of the state contain a Dead Predicate fault, and we say that the set of rules contains a* Dead State fault*, a pattern of faults characterized by a deadlock state.*

A state can violate the State Liveness property when it has no active rules or when all of its rules violate the Rule Liveness property. A rule violates the Rule Liveness property if its trigger predicate is not satisfiable in the current state because of the current assignment or because other rules with higher priority are satisfied first. A-FSMs implementing a weak priority order as described in Section 5.1.1 are less prone to dead predicate faults because multiple rules can coexist at the same priority level.

5.2.3 Reachability

In the A-FSM model states represent the different behaviors that the modeled CAAA can assume. Each one of such behaviors are meant to be applied as a response to certain circumstances. However, due to issues in the rule design, one or more states can become unreachable, preventing the application from applying the correct behavior. We describe state reachability as follows:

Reachability property: *For every state, it is possible to reach the state from the initial state via some sequence of adaptations. If a state of the CAAA is not reachable (through any sequence of adaptations), we say that the rules contain an* Unreachable State fault*.*

The Unreachable State fault represents a complementary concept of the Dead State fault. The former underlines those states unreachable by the automata, the latter those in which the automata remains blocked. Besides the obvious circumstance in which there are no incoming rules to a certain state, we have identified two causes which lead to violation of the reachability property: (1) a too strong priority order may cause high priority rules to mask other active rules; if the masked rules are the only ones reaching their destination state, that, or those, states become unreachable, violating both the liveness and the reachability property, (2) a particular assignment of PCVs either selected by the user or resulting of previous adaptations, which prevent rules from being applied (e.g. if the GPS has been disabled to spare battery certain location based rules may not be applied).

5.2.4 State Invariants

At runtime PCVs are sequentially reassigned by changes in the environment, as effect of adaptations and by explicit user's intervention. After an adaptation some PCVs hold assignments they had due to previous events. Such assignments are unpredictable and on certain states lead to faults, causing the

CAAA to expose a behavior not suitable for the current situation. The nature of those side effects vary between a mild annoyance to more serious faults. Imagine for instance an application skipping a speed security check because the GPS has been turned off. To detect faults or side effects resulting from misconfiguration we have enriched the A-FSM with the notion of *invariants*. Similarly to what it happens in contract based-testing [Aic03] an invariant allows testing engineers to specify conditions which should be respected while a certain behavior is applied. We define the State Invariant property as follows:

State Invariant property. *After a context change the current state $S \in \mathcal{S}$, and the current PCVs assignment to $\mathcal{V}$, are violating the State Invariant property if the assignment is stable in S and if it violates at least one invariant in $Inv(S)$*. We call State Invariant violations those faults which violates the State Invariant property.

There are two possible ways in which a violation can occur: (1) upon *entering* a state S by means of a rule R, the actions associated with R make assignments to PCV that violate the invariant; (2) a change in the context occurs while the A-FSM is in state S (which is reflected in a change to the PCVs), such that the new PCVs assignment violates at least one invariant in $Inv(S)$, and there is no rule R *exiting* from S which is activated by the new PCVs assignment. Note that faults violating the stability property below may also hide state invariant violations along their paths. Similarly that state invariants violations may also happen simultaneously with violations of the Rule Liveness property. When this happens an unanticipated PCVs assignment is preventing the CAAA from adapting and is imposing faulty behavior.

5.2.5 Stability

CAAAs suffer from stability and metastability issues which manifest themselves with an (un)bounded sequence of consecutive adaptations. Such sequences have a different impact on the expected application behavior, varying from simple annoyances to the user to continuous adaptation loops. The cause of such failures is a subtle dependence between consecutively triggered predicates and the sensed duration of assignments of certain PCVs.

Adaptations should not implicitly depend on the duration in which a propositional context variable holds a certain value. If they do so the obtained behavior would be nondeterministically based on how promptly the application is sensing such values. While it is legitimate for an application to use durations as PCVs, (e.g., logging and tracing applications may need to monitor or record certain events or to trigger a certain behavior after a certain time-based threshold has been reached), but not to base sequences of adaptations on the duration of such PCV assignments.

Imagine an adaptive application which, with a given PCVs assignment could adapt from a state A to a state B and from B to C. Assume also that, for various reason, each adaptation would take an amount of time considerably greater that the refresh rate of the constant. This would be the case of adaptation involving the execution of additional components or the connection to remote services. Once the first adaptation has been triggered the final destination state is nondeterministic and it could be both B or C depending on how long the PCVs assignment lasted and how slow was the adaptation. Also suppose that the same assignment from C would trigger an adaptation to A. If that happen the application becomes metastable and keeps adapting until the context changes. Once again when the context changes and the application returns stable its state could be any of those involved in the loop.

Consider an ordered set $SC = \{S_0, S_1, S_2, ..., S_n\}$ of states connected by an ordered sequence $AR = \{R_0, R_1, R_2, ..., R_n\}$ of adaptation rules. If the first and the last states S_0, S_n of SC coincide we say that the set SC is a *loop*. Assume that it exist an ordered sequence of PCVs assignment $V = \{V_0, V_1, V_2, ..., V_n\} \subset \mathcal{V}$, of which V_0 satisfies R_0 in S_0, V_j satisfies R_j in S_j with $j \in \{1...n\}$ and $V_j = V_{j-1} \wedge k_{j-1}$ where k_{j-1} is the action predicate associated to the rule R_{j-1}. If V exists then the CAAA is metastable and its adaptations are implicitly depending on the sensed duration of the sensed context. If $V_0 = V_n$ then the CAAA is unstable as long as the sensed context holds.

If V_0 is sensed in S_0 the application adapts in S_1 and applies k_0 to V_0. Once in S_1, if V_1 still holds, then the application will adapt to S_2 and it will continue to adapt until the last state S_n is reached, after which it will restart from S_0 if SC is a loop. The final state is nondeterministic and it could be any state in SC depending on how long the sensed context holds. During this sequence of adaptations the application has been unstable and the number of adaptations that have been performed is also unpredictable. If the sensed context changes it interrupts the sequence of adaptations and leaves the CAAA nondeterministically in one of the intermediate states. To unmask faults such as these we define the Stability property as follows:

Stability property: *A CAAA is stable if there is no PCVs assignment which can produce a sequence of adaptations such that the choice of which state ends the sequence depends on the duration with which the assignment itself holds its value. More specifically, given a state S_i and a PCVs assignment V triggering an adaptation from S_i to S_j with action k, we say that a CAAA contains an* Adaptation Race fault *if the assignment obtained by applying k to $\mathcal{V}$ is triggering a sequential adaptation from S_j. If the sequence of adaptations triggered by the the propagation of the initial assignment forms a loop, then we say that the rules contain an* Adaptation Cycle fault.

Often these patterns of behavior may produce multiple adaptations that merely annoy the user with repeated updates. Nevertheless, races can be dangerous, particularly in situation when the CAAA is operating some kind of electro-mechanic device (e.g. a robot) in which repeated updates may damage the circuits or endanger the operator.

5.3 Extracting the A-FSM From Our Case Studies

5.3.1 PhoneAdapter's Model

PhoneAdapter is a phone profiling application that we have introduced in Section 4.1.1. Table 5.1 presents the set of adaptation rules according to which *PhoneAdapter* modify its behavior. For convenience, the table represents rule predicates both in their fully expanded form and as a composition of PCVs. The conversion between a PCV and its underlying predicate is contained in Table 5.3. Table 5.2 shows the actions applied by each rule. In this example rules only affect the ring-tone volume's, the vibration, the call diversion and the synchronization. This table has only been included for completeness and, since none of the action changes the value of any of the PCVs they only modify the phone behavior

Table 5.1: Adaptation Rules of PhoneAdapter

Rule Name	**Current States**	**New State**	**Full Predicate**	**Simple Predicate**	**Priority**
ActivateOutdoor	General	Outdoor	GPS.isValid() $\wedge$ $\neg$ GPS.location()=home $\wedge$ $\neg$GPS.location()=office	$A_{gps} \wedge \neg B_{gps} \wedge \neg C_{gps}$	5
DeactivateOutdoor	Outdoor	General	$\neg$ActivateOutdoor	$\neg$ $(A_{gps} \wedge \neg B_{gps} \wedge \neg C_{gps})$	5
ActivateJogging	Outdoor	Jogging	GPS.isValid() $\wedge$ GPS.speed()$>$ 5	$A_{gps} \wedge D_{gps}$	5
DeactivateJogging	Jogging	Outdoor	$\neg$ActivateJogging	$\neg(A_{gps} \wedge D_{gps})$	5
ActivateDriving	General, Home, Office, Outdoor	Driving	BT=car_handsfree	A_{bt}	1
DeactivateDriving	Driving	General	$\neg$ActivateDriving	$\neg$ A_{bt}	1
ActivateDrivingFast	Driving	DrivingFast	GPS.isValid() $\wedge$ GPS.speed()$>$ 70	$A_{gps} \wedge E_{gps}$	0
DeactivateDrivingFast	DrivingFast	Driving	$\neg$ActivateDrivingFast	$\neg(A_{gps} \wedge E_{gps})$	0
ActivateHome	General	Home	BT=home_pc $\vee$ (GPS.isValid() $\wedge$ GPS.location()=home)	$B_{bt} \vee (A_{gps} \wedge B_{gps})$	5
DeactivateHome	Home	General	$\neg$ActivateHome	$\neg(B_{bt} \vee (A_{gps} \wedge B_{gps}))$	5
ActivateOffice	General	Office	BT=office_pc $\vee$ BT=office_pc_* $\vee$ (GPS.isValid() $\wedge$ GPS.location()=office)	$C_{bt} \vee D_{bt} \vee (A_{gps} \wedge C_{gps})$	5
DeactivateOffice	Office	General	$\neg$ActivateOffice	$\neg(C_{bt} \vee D_{bt} \vee (A_{gps} \wedge C_{gps}))$	5
ActivateMeeting	Office	Meeting	Time$>=$meeting_start $\wedge$ BT.count()$>=$ 3	$A_t \wedge E_{bt}$	4
DeactivateMeeting	Meeting	Office	Time$>=$meeting_end	B_t	4
ActivateSync	General	Sync	BT=home_pc $\vee$ BT=office_pc	$B_{bt} \vee C_{bt}$	9
DeactivateSync	Sync	General	$\neg$ActivateSync	$\neg(B_{bt} \vee C_{bt})$	9

and not how the phone behavior adapts.

As shown in the table, *PhoneAdapter* adapts between nine different states according to 16 different rules expressed over three different sensed context variables, namely BT (Bluetooth), GPS and time, which are monitored via 12 propositional context variables representing the 12 different relational expressions in which the sensed context variables are used. For example, one such relational expression is *GPS.location()=home*, which tests whether the location sensed by the phone's GPS sensor corresponds to the user's home location (stored in configuration variable *home*). This relational expression is represented throughout the rules by the propositional context variable B_{gps}.

We define the following global constraints for *PhoneAdapter*, which account for the facts that (1) checking context via GPS first requires GPS to be on, (2) locations are mutually exclusive, (3) disequations on the same context variable must be consistent with each other, and (4) the end time of a meeting is later than its start time:

$$\neg A_{gps} \Rightarrow (\neg B_{gps} \wedge \neg C_{gps} \wedge \neg D_{gps} \wedge \neg E_{gps})$$

$$(B_{gps} \Rightarrow \neg C_{gps}) \wedge (C_{gps} \Rightarrow \neg B_{gps})$$

$$E_{gps} \Rightarrow D_{gps}$$

Table 5.2: Adaptation Rules' action of PhoneAdapter

Rule Name	Action
ActivateOutdoor	volume=high, vibration=on
DeactivateOutdoor	volume=medium, vibration=on
ActivateJogging	BT=off, volume=high, vibration=off
DeactivateJogging	BT=on, volume=high, vibration=on
ActivateDriving	volume=high, vibration=off
DeactivateDriving	volume=medium, vibration=on
ActivateDrivingFast	vibration=off, volume=off, divert_call=on
DeactivateDrivingFast	vibration=off, volume=high, divert_call=off
ActivateHome	volume=medium, vibration=off
DeactivateHome	volume=medium, vibration=on
ActivateOffice	volume=low, vibration=on
DeactivateOffice	volume=medium, vibration=on
ActivateMeeting	volume=off, vibration=on
DeactivateMeeting	volume=low, vibration=on
ActivateSync	synchronization=on
DeactivateSync	synchronization=off

Table 5.3: PCVs in PhoneAdapter's A-FSM model

Context	Predicate	PCV
GPS	GPS.isValid()	A_{gps}
GPS	GPS.location()=home	B_{gps}
GPS	GPS.location()=office	C_{gps}
GPS	GPS.speed()$>$ 5	D_{gps}
GPS	GPS.speed()$>$ 70	E_{gps}
Bluetooth	BT=car_handsfree	A_{bt}
Bluetooth	BT=home_pc	B_{bt}
Bluetooth	BT=office_pc	C_{bt}
Bluetooth	BT=office_pc_*	D_{bt}
Bluetooth	BT=office_pc_*	D_{bt}
Bluetooth	BT=office_pc_*	D_{bt}
Bluetooth	BT.count()$>=$ 3	E_{bt}
Time	Time$>=$meeting_start	A_t
Time	Time$>=$meeting_end	B_t

$$B_t \Rightarrow A_t$$

Figure 5.1 depicts the A-FSM we derive from the adaptation rules of *PhoneAdapter*, with state *General* being its initial state. For completeness Table 5.4 shows all the state invariants defined for PhoneAdapter.

At the early stage of this research work we tested PhoneAdapter by running it within TestingEmulator [SRb] and by manually triggering variations in the context and we exposed several faults as anticipated in Section 4.1.1. By means of the A-FSM model we can now examine such failures and we can explain them as violations of properties which the A-FSM model should respect.

In state *General* 4 of the 5 outgoing rules try to infer the location both from Bluetooth and GPS readings. Such context readings are independent from each others and they may satisfy multiple trigger predicates simultaneously causing nondeterministic faults to happen. The fifth of those 5 rules, *ActivateSync* has low priority and its predicate is completely masked by the other rules causing a Dead Rule

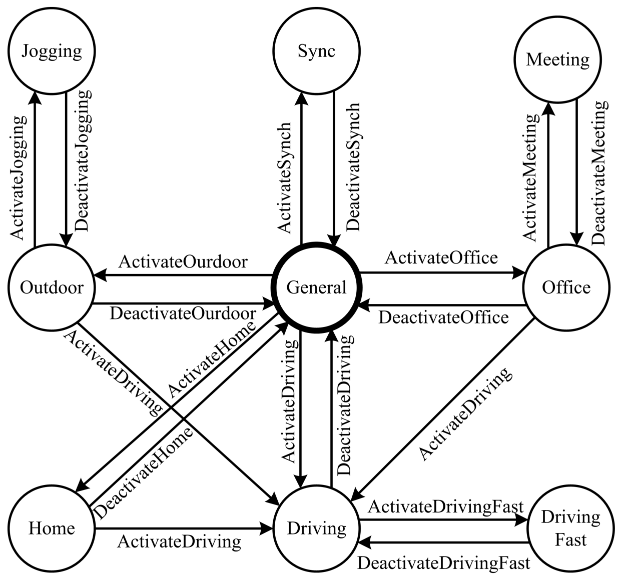

Figure 5.1: A-FSM of PhoneAdapter.

Table 5.4: State invariants in PhoneAdapter's A-FSM model

State	State invariant
General	
Outdoor	
Jogging	GPS.speed() $>$ 5
Driving	BT=car_handsfree
DrivingFast	BT=car_handsfree
Home	
Office	
Meeting	
Sync	BT=home_pc $\vee$ BT=office_pc

fault. Unfortunately *ActivateSync* is also the only incoming rule to state *Sync*, therefore the Dead Rule fault to which it is affected propagates and causes also a State Reachability fault preventing state *Sync* from being reachable. The Stability Property is also not satisfied in several states causing the model to be metastable if a certain environmental changes happen. Particularly, the rules *ActivateMeeting* and *DeactivateMeeting* creates a loop causing the model to become unstable if the application enters in state *Office* or *Meeting* after a meeting.

5.3.2 GPS-Recording's Model

As above *GPS-Recording* is an application that we crafted to evaluate and compare our techniques. Its implementation was derived directly from the model (and not the other way around). The A-FSM we used is depicted in Figure 5.2 of which each adaptation is described in Table 5.5. The A-FSM model also includes the state invariants listed in Table 5.6.

As introduced in Section 4.1.7 the application enables GPS-based trekking tour recording. Tourists can rent from a base camp a GPS-enabled device on which the recording application is running. When the application starts, it enables the GPS and starts reading the current location. As soon as the users

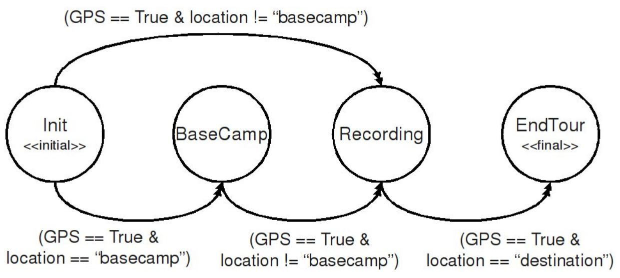
(GPS == True & location != "basecamp")
Init
<<initial>>
BaseCamp
Recording
EndTour
<<final>>
(GPS == True & location == "basecamp")
(GPS == True & location != "basecamp")
(GPS == True & location == "destination")

Figure 5.2: Crafted GPS recording application.

Table 5.5: Adaptation Rules of GPS-Recording

Rule Name	**Current States**	**New State**	**Full Predicate**	**Priority**	**Action**
StartConfiguration	Init	BaseCamp	GPS.isValid() ∧ GPS.location()= "base-camp"	5	waitToStart()
StartRecording	Init, BaseCamp	Recording	GPS.isValid() ∧ ¬GPS.location()= "base-camp"	5	enableRecordingMode()
SaveResults	Recording	EndTour	GPS.isValid() ∧ GPS.location()= "desti-nation"	5	saveTourStatistics()

leave the base camp, the application starts collecting GPS information showing them on a map, and recording statistics about the route, including position, altitudes and speed. At the end of the tour results can be uploaded and stored.

We model the application with four states:

1. *Init*: the initial state;
2. *BaseCamp*: the user is still in the starting point of the route and the application is not recording;
3. *Recording*: the user has left the base-camp and the application is recording times and locations; and
4. *EndTour*: the user has reached the destination, the recording is blocked and the statistics are uploaded.

We introduce a number of rules: a rule *ActivateBaseCamp* is triggered from *Init* if $GPS = True$ and *location* = *"base-camp"*. A rule *StartRecording* is triggered from *Init* or from *BaseCamp* if GPS is providing a valid reading and if *location* ≠ *"base-camp"*. The tour ends when from *Recording* the application triggers *ApplyEndTour*, which is satisfied if *GPS* = *True and location* = *"destination-camp"*. When triggered each one of these rules execute one of the procedures listed in the "action" column of Table 5.5 which activate a certain behavior in the application. To respect the law of physics affecting the

Table 5.6: State invariants in GPS-Recording's A-FSM model

State	State invariant
Init	
Basecamp	GPS.location()=basecamp
Recording	GPS.location()¬destination
EndTour	

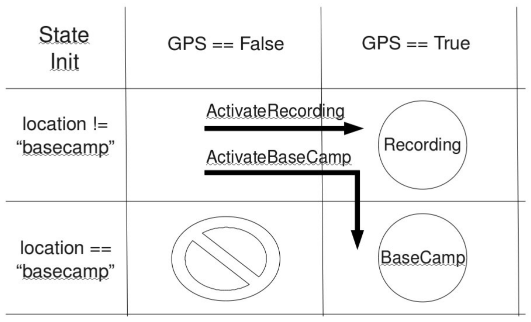

Figure 5.3: State Matrix of state *Init*.

PCVs we applied the following constraints:

$$\neg GPS.isValid() \Rightarrow \neg(GPS.location() = \text{“}base - camp'' \vee GPS.location() = \text{“}destination'')$$

$$GPS.location() = \text{“}base - camp'' \Rightarrow \neg GPS.location() = \text{“}destination''$$

$$GPS.location() = \text{“}destination'' \Rightarrow \neg GPS.location() = \text{“}base - camp''$$

.

The expected behavior of this application is a first adaptation from *Init* to *BaseCamp*, a second adaptation to *Recording* and a third adaptation to *EndTour*. However it should also be possible to adapt from *Init* directly to *Recording* if a user turns the application on when he has left the base camp already (see Figure 8.1). Instead, starting from the initial state it is not possible to find any valid path capable of triggering *ActivateBaseCamp* and therefore capable of reaching the state *BaseCamp*.

Figure 5.3 shows the Karnaugh map [Kar53] for the activation of state *Init* trigger predicate. By means of this Karnaugh map it is possible to follow the execution paths in the space of possible assignments of state *Init* and it helps to detect faults. A constraint imposes that when the GPS reading is not valid then all the predicates on locations are False, therefore the assignment *GPS* $=$ *False and location* $=$ *“base-camp”* is not valid. Starting from an initial configuration with *GPS* $=$ *False and location* $\neq$ *“base-camp”* the only path to reach state *BaseCamp* pass through the configuration *GPS* $=$ *False and location* $\neq$ *“base-camp”* which triggers an adaptation to *Recording*.

In this crafted example, using a Karnaugh map, it is straightforward to spot that there is no direct path from a stable and valid configuration to any configuration triggering *ActivateBaseCamp*; however similar situations may occur in applications with more rules, and they can be difficult to detect. Moreover Karnaugh maps can only represent Cartesian spaces with up to six Boolean dimensions and cannot be used for more complex situations.

5.3.3 Timeriffic's Model

Timeriffic is a simple open source application that we have introduced in Section 4.1.6.

As shown in Figure 5.4, *Timeriffic*, in all the revisions before 86, can be modeled by means of three states: *Init*, *DayMode* and *NightMode*. When the application starts it immediately adapts to state

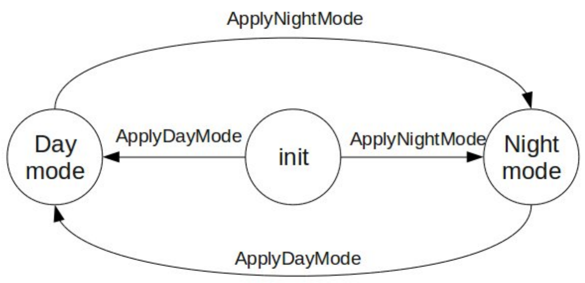

Figure 5.4: Timerrific's model.

Table 5.7: Adaptation Rules of GPS-Recording

Rule Name	Current States	New State	Full Predicate	Priority	Action
ApplyNightMode	Init, DayMode	NightMode	time $>$ NightStarts $\wedge$ time $<$ DayStarts	5	volume=high (human error)
ApplyDayMode	Init, NightMode	DayMode	time $>$ DayStarts $\wedge$ time $<$ NightStarts	5	volume=off (human error)

DayMode or *NightMode* depending on the current time, then it sequentially adapts from one to the other at the scheduled time. Time-related PCVs must respect the following constraints:

$$time > NightStarts \Rightarrow time > DayStarts$$

$$time < DayStarts \Rightarrow time < NightStarts$$

meaning that the configuration variable DayStarts must be lower than the variable NightStarts, and that is the time is lower than DayStarts then it is also lower than NightStarts and if it is greater than NightStarts then it is also greater than DayStarts.

In the initial stages of its development, *Timeriffic* was affected by a fault causing the phone to be muted during the day instead of at night. This fault appears to be a violation of the state invariant property, caused by bug in the code in which a developer inverted the actions of the adaptation rules of day and night mode. As defined in Section 5.2.4, state invariant violations happens when a certain PCVs is stable in a state in which it should not be (e.g. in which it violates one or more state invariants). The bug was introduced in revision 7 and fixed over two months later in revision 86. We modeled this bug in the "Action" column of Table 5.7 where it appears clearly that the actions of the two rules are inverted.

With the A-FSM model that bug can be easily exposed by imposing on the ringtone's configuration the state invariants shown in Table 5.7. This particular fault fault was caused by human mistake and the A-FSM, even if applied at design time would not have prevented it from happening. However the A-FSM

Table 5.8: State invariants in Timerrific's A-FSM model

State	State invariant
Init	
DayMode	volume$\neg$off
NightMode	volume$=$off

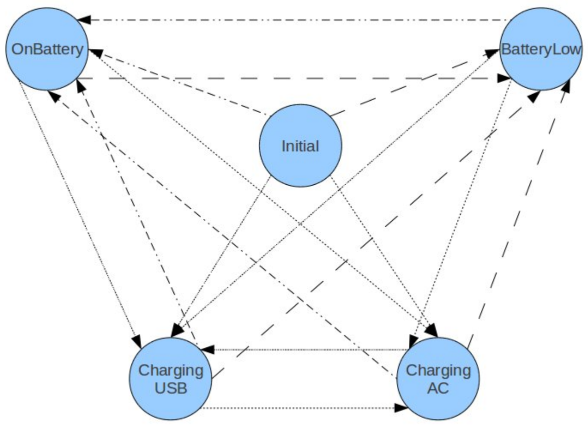

Figure 5.5: Power Manager's model.

would have helped developers in exposing, detecting and fixing the fault as soon as it was introduced in the code.

5.3.4 Power Manager's Model

Power Manager is a commercial Android application available from the Android's Market that we have described in Section 4.1.5. Out of the box, *Power Manager* maximises the battery by switching from various user-defined configurations according to environmental parameters.

Extracting the A-FSM model from *Power Manager* simply requires us to represent its configuration with the A-FSM. On the Android G1 Phone *Power Manager* stores his configuration in the default Sqlite DB as recommended by the Android development guidelines. In our research we use a consumer version of the G1 phone in which access to that DB is disabled. Indeed in an unlocked development device in which the Sqlite DB would be reachable this model extraction could be easily automated. In the configuration files the user can specify various profiles. For each profile the user has to specify a condition of activation and the effect that has to be applied when that profile is applied. The generated A-FSM represents each one of these profiles with a state. Those states are fully connected to each other. For each state all the incoming adaptations have, as trigger predicate, the predicate with which the profile is applied and as action the effect that has to be applied when applying the new profile.

Figure 5.5 and Table 5.9 depicts the A-FSM model that we have obtained from our custom configuration. The model has five states:

- *Initial*: the initial state when the application starts;
- *ChargingAC*: the phone is charging using an AC charger;
- *ChargingUSB*: the phone is charging using a USB cable;
- *OnBattery*: the phone is running on battery and the battery charge is above 30%; and
- *BatteryLow*: the phone is running on battery and the battery charge is under 30%.

Table 5.9: Adaptation Rules of PowerManager

Rule Name	Current States	New State	Full Predicate	Priority	Action
ApplyOnBattery	Initial, BatteryLow, ChargingUSB, ChargingAC	OnBattery	$(\neg$ battery $< 30\%) \wedge$ charging=no	5	
ApplyBatteryLow	Initial, OnBattery, ChargingUSB, ChargingAC	BatteryLow	battery $< 30\% \wedge$ charging=no	5	$\neg$ BT.isEnable(), $\neg$ WiFi.isEnable(), GPS.isEnable(), screen_brightness=low
ApplyChargingUSB	Initial, OnBattery, BatteryLow, ChargingAC	ChargingUSB	charging=USB	5	BT.isEnable(), WiFi.isEnable(), GPS.isEnable(), screen_brightness=high
ApplyChargingAC	Initial, OnBattery, BatteryLow, ChargingUSB	ChargingAC	charging=AC	5	BT.isEnable(), WiFi.isEnable(), GPS.isEnable(), screen_brightness=high

Table 5.10: State invariants in PowerManager's A-FSM model

State	State invariant
Initial	
OnBattery	$\wedge$ (BT.isOn() $\wedge$ WiFi.isOn() $\wedge$ GPS.isOn()) $\wedge \neg$ screen_brightness=high
BatteryLow	($\neg$ BT.isOn()) $\wedge$ ($\neg$ WiFi.isOn()) $\wedge$ ($\neg$ GPS.isOn()) $\wedge$ screen_brightness=low
ChargingUSB	
ChargingAC	

All the rules adapting to the same state have the same trigger predicate and apply the same action and, in the picture, they have been represented with arrows with the same style. The model also constraints the value of its PCVs by forcing that:

$$screen\_brightness = high \Rightarrow \neg(screen\_brightness = medium \vee screen\_brightness = low)$$

$$screen\_brightness = medium \Rightarrow \neg(screen\_brightness = high \vee screen\_brightness = low)$$

$$screen\_brightness = low \Rightarrow \neg(screen\_brightness = high \vee screen\_brightness = medium)$$

$$charging = no \Rightarrow \neg(charging = USB \vee charging = AC)$$

$$charging = AC \Rightarrow \neg(charging = no \vee charging = USB)$$

$$charging = USB \Rightarrow \neg(charging = no \vee charging = AC)$$

In this custom configuration, adaptations to *BatteryLow* turn off Bluetooth, GPS and WiFi to extend the battery life. Adaptations to *ChargingUSB* and *ChargingAC*, instead, are defined with the associated actions of turning on all the sensors, since the phone can use them without exhausting the battery, which is on charge. Adaptations to *OnBattery* do not perform any action since the battery is full and the phone can be used as it was configured.

To find the malfunction that exists in in our test configuration we added to the A-FSM model state invariants for the two states running on battery as shown in Table 5.10. When the phone is in state

BatteryLow, we require that all the sensors must be off, and the back-light should be low. When the phone is in state *OnBattery*, we require that at least one of the sensors should be off or the back-light should not be high, to avoid an excessive battery consumption.

5.4 Considerations About the A-FSM Model

We have designed the A-FSM model to give developers a general-purpose, high-level view of the adaptation logic of CAAAs. Indeed, we have good results in applying this model to various applications and, as we will present in the rest of this Thesis, we also manage to use it as base for several validation techniques. There are two considerations which need to be underlined.

First, the A-FSM model is general-purpose. To model correctly or more efficiently applications belonging to particular application domains, the model may need to be extended with additional domain-specific features. Imagine, for instance, industrial applications made for controlling mechanical external devices with some sort of electro-mechanical actuator. In those CAAAs, the delay between when an adaptation rule is triggered until its effect has been applied and the adaptation completed may be considerable. We call that gap time an *adaptation transitory*. During the adaptation transitory the application is in an intermediate state in which its behavior may be different from the origin and destination state. Indeed, it would be possible to model those transitories with additional intermediate states, but it would be better to extend the A-FSM in order to model, this and any other required concept, properly.

Secondly, the A-FSM is not trying to impose to developers a certain design, and instead it just offers tools to model CAAAs with a level of abstraction high enough to identify faults in the adaptation logic. Thus, the model itself is not preventing developers from introducing faults in their applications. Consider for instance Locale, an application winner of the Android Contest 2009 [Two09]. Locale allows users to define through its GUI rules, trigger predicates, and effects, with a very similar level of abstraction as the one given by the A-FSM. However, to prevent the most common faults, Locale imposes limitations. A strong priority order between rules is forced, only conjunctions can be used in the trigger predicates, and sensors cannot be turned off as the effect of an adaptation. Those limitations increase the robustness of the created configurations but also reduce their flexibility. Indeed, any Locale configuration can be modeled with an A-FSM, but not all A-FSM models can be applied in Locale.

5.5 Related Work

The work by Nilsson and Offut [NO07] uses patterns of potential faults to detect missed scheduling of sporadic and periodic time-critical tasks. Our approach employs a similar idea, in which the different refresh rates of asynchronously updated context variables may trigger incorrect decisions in an application's adaptation behavior. We analyze the impact of asynchronous updates on the evaluation and triggering of adaptation rules, allowing us to determine where faults can occur.

CAAAs also suffers from real-time issues. Timing problems in real-time systems have been analyzed by using methods based on specialized finite-state models. Alur and Dill propose a timed automata model that incorporates time constraints for specifying real-time systems [AD94]. Timed automata have been utilized as well by several test case generation techniques, which exploit timing constraints specified between actions or events [ENDK02, HNTC99, KT04, NS03]. In a similar way, we apply constrains not only to time, but to all kinds of context inputs utilized by the adaptation rules.

Timing problems in real-time systems have been analyzed by using methods based on specialized finite-state models.

Alur and Dill propose a timed automata model that incorporates time constraints for specifying real-time systems [AD94]. Timed automata have been utilized as well by several test case generation techniques, which exploit timing constraints specified between actions or events [ENDK02, HNTC99, KT04, NS03]. In a similar way, we apply constrains not only to time, but to all kinds of context inputs utilized by the adaptation rules.

In 1971 Akiyama [Aki71] published the first attempt to use metrics for software quality prediction by proposing a regression-based model for module defect density, measuring the number of defects per KLOC (kilo lines of code). The statistical model proposed by Akiyama and all the ones predicting the quality of a system in terms of his complexity differ from our A-FSM first because they are stochastic models while the A-FSM is deterministic, second because they predict either the fault occurrence rate or the number of faults to be expected, while the A-FSM model detects an input or a sequence of inputs which triggering behaviors considered faulty. It would be possible to create a statistical model for the CAAAs using the number of states rules and PCVs as metric of complexity for example by estimating the number of human introduced faults by number of rule.

Probabilistic model checkers such as PRISM [KNP02] are capable of validate probabilistic automata (PAs) and probabilistic timed automata (PAs) using a discrete-event simulation engine. By modeling a CAAAs with PRISM developers could answer questions such as “How often a certain adaptation will occur?” or “Which state will be applied in the first 3 hours of usage?”. Contrarily our A-FSM model assumes that all the given PCVs can be satisfied and verifies what happens when they do. The A-FSM model could be extended with the PRISM formalism in order to estimate the occurrence rate of all the detected faults. Developers could benefit of this addition by addressing first all those fault which are most likely to appear withing a given amount of time (i.e. all the fault which are more likely to appear within the first 24 hours of usage).

Chapter 6

Validation Techniques

The A-FSM model defined in Chapter 5 gives developers a high-level representation of the adaptation logic. Based on such representation, properties capable of proving (or disproving) the correctness of the CAAA under test can be defined. In Section 5.2 we introduced a set of unique properties which we used to validate our case studies. Such properties are capable of isolating novel classes of faults which affect CAAAs and allow developers to correct the adaptation logic. Developers may also introduce additional properties if needed.

In this chapter we introduce three novel techniques which, based on the A-FSM model, validate the CAAA under test by applying the properties we have defined in Section 5.2 and report instances of faults. We classify the first two of these techniques as model checking approaches, because, although they do not use a specific model checker, they work in a similar way. The *Enumerative Approach*, presented in Section 6.1, enumerates all the possible inputs and verifies their effect on the CAAA. The *OBDD-based Approaches* presented in Section 6.2 use a symbolic approach to very our properties without looping on all the possible inputs. Finally, our *PDDL-based approach*, presented in Section 6.3 employs a planner to validate the A-FSM model, removing the need of writing specific fault detection algorithms for each property.

In the following chapters these techniques will be applied to our case studies, and both the results they provide and their performance will be evaluated. In Chapter 7 we will validate our case studies by applying our properties. Detected faults will be isolated and discussed. In Chapter 8 we will discuss the effectiveness of these three techniques, and we will compare their performances.

6.1 Enumerative Approach

Our initial algorithms for detecting faults in an A-FSM employ enumerative exploration of an *explicit* representation of the state space of the A-FSM. The idea behind the algorithms is to observe the adaptive behavior of the A-FSM independently in each state using a chosen set of assignments of values to the propositional context variables. By probing all states and all possible assignments of values, we construct a derivative representation called a *state matrix* that is associated with each state S in M. This data structure is used as a base for all the enumerative algorithms.

Conceptually, the state matrix of S enumerates *bit strings* for associated sets of rules. A bit string is a sequence of ordered bits, of which each one specifies a set of truth assignments to the set $\mathcal{C} \subset \mathcal{V}$ of propositional context variables that can cause the predicates of the associated rules to become satisfied and the destination states of those rules to be entered. Bit strings that do not satisfy the predicates of any active rules of a state S are not included in the state matrix of S.

Imagine an A-FSM model M_1 composed by two states S_0 and S_1 with a rule R_1 adapting from S_0 to S_1 and a rule R_0 adapting from S_1 to S_0 both with the same level of priority. This is a simple model with two states allowing adaptations to each other. Also imagine two PCVs P_a and P_b where the trigger predicate of R_1 is defined as $P_a \wedge P_b$ and the trigger predicate of R_0 is $P_a \oplus P_b$. For completeness suppose that in M_1 there is a constrain C_1 imposing that $\neg P_b \Rightarrow P_a$. Simply when both P_a and P_b are satisfied M_1 should adapt state S_1, otherwise it should adapt to S_0. In addition, due to the constraint, P_a and P_b cannot be simultaneously not satisfied.

The Listing 6.1 depicts the state matrices for M_1. There are two state matrices, on for each state. Both the state matrices contains all the entries representing PCVs assignments triggering an adaptation with the exclusion of the assignment which violates C_1. For clarity in Listing 6.1 we represented each entry of the two state matrices with two notations: one explicit and one compact (the one inside the brackets). The compact one is the notation produced in output by our implementation of this technique. In the compact notation the bit string represents the assignment of each PCV in a fixed order, which in this case is the order in which they were defined. For instance, *01* stands for P_a = *false,* P_b = *true*. The remaining part of the compact notation represents an array of triggered adaptation rules, where the array *[R_1]* indicates that R_1 is triggered.

Listing 6.1: State matrices for M_1 with extended notation

```
state S_0:
P_a = true, P_b = true: triggers R_1 (11, [R_1])

state S_1:
P_a = false, P_b = true: triggers R_0 (01, [R_0])
P_a = true, P_b = false: triggers R_0 (10, [R_0])
```

Listing 6.2 depicts examples of entries from two state matrices for *PhoneAdapter*, the one for state *General* and the one for state *Outdoor* as they are reported by our Java implementation using a compact notation illustrated previously. Notice that state matrices are used as input for our detection algorithms and are not . The print out in 6.1 is only a representation of how the data structure looks like.

Listing 6.2: Example State Matrix Entries

```
state General:
(110000000000, [ActivateHome])
(110001000000, [ActivateHome, ActivateDriving])
...
...

state Outdoor:
(100100000000, [ActivateJogging])
(100101000000, [ActivateJogging, ActivateDriving])
```

...
...

As mentioned for the previous example the previous example also in Listing 6.2 each entry in the state matrix of a certain state S depicts a bit string of variable assignments (with 1 indicating true and 0 indicating false) along with the names of the active rules of S whose predicates become satisfied upon the variables obtaining those assignments. Note that while the order of variables does not matter, the same order must be used for all state matrices. In our experiment we ordered all the PCVs in phone adapter first by context (i.e. GPS, Bluetooth, time) and then alphabetically by their simplified names as shown in Table 5.1 (e.g. A_{gps}, B_{gps}, ...).

By constructing each state matrix over *all* propositional context variables, the size of the state matrix grows exponentially in the number of variables. For instance, considering a CAAA with the same number of states and PCV of *PhoneAdapter*, in the worst case scenario, the number of such entries would be $O(S * 2^{PCV}) = 9 * 2^{12} = 36864$.

6.1.1 Localization

It is possible to reduce the memory footprint of the state matrices and therefore the execution time of some of our algorithms by excluding from the state matrix of a state whose propositional context variables which are not relevant to that particular state. For each state we may therefore construct a projection of the state matrix in that state, obtaining a *reduced state matrix* that contains bits for only the relevant variables of the state (with the remaining bits essentially set to the value “don’t-care”).

For faults related to properties whose occurrence is local to a single state, such as the Nondeterminism property, there are no side effects in using the reduced state matrices. Instead, when detecting faults resulting from sequential adaptations between various states, using reduced state matrices requires additional effort because their detection requires to monitor multiple states at once and therefore it also requires to switch from a state matrix to the other.

The solution we embody in our implementation is able to use the reduced state matrices by ”localizing” (i.e. considering a state-local subset of PCVs starting from a PCVs assignment) and ”globalizing” (i.e. applying changes to a PCVs assignment starting from a state-local PCVs assignment) between the CAAA and one of the reduced state matrices. Consider a global assignment $a_g \in G$ containing all the PCVs of the A-FSM model. As such assignments are composed by an ordered sequence of PCVs their validation does not require computing the global state matrix. Indeed it is possible to enumerate all of them by counting from 0 to $2^n - 1$, where n is the total number of PCVs in the A-FSM.

Starting from each of such global assignments from the initial state, we can localize a_g in the initial state S_I obtaining a_{S_I}. The *localize* function is a surjection associating to each $a_g \in G$ at most one $a_{S_I} \in M_{S_I}$. This association is performed by removing all the PCVs not in S_i, and by preserving the original order of the remaining ones. For instance an assignment $v_1v_2v_3v_4$ could become v_1v_3 if localized into a state using only v_1 and v_3. If a_{S_I} does not exists in M_{S_I} it means that that specific assignment is stable in S_I and that the execution path starting from a_g has terminated in S_I. Otherwise, if a_{S_I} exists, it identifies an adaptation to a state S_j with an action k.

To continue the exploration we need to follow the adaptation in S_j. Intuitively we could apply k to a_{S_i} obtaining a'_{S_j} then either *migrate* a'_{S_j} to S_j or, more intuitively, *globalize* a'_{S_j} to G and re-*localize* it in M_{S_j}. Unfortunately the two functions migration and globalization are both not injective and not surjective, therefore to each a'_{S_j} the migration associates a set of assignments $A_{S_j} \subset M_{S_j}$ and the globalization associates a set $A_g \subset G$. The cardinality of this sets is equal to the number of assignments of PCVs not in S_i but respectively in S_j or in the whole A-FSM. For instance, assuming that v is the number of PCVs in $\mathcal{V}_{S_j} - \mathcal{V}_{S_i}$ where $\mathcal{V}_{S_i}$ is the set containing only those PCVs appearing in state S_i, A_{S_j} will have size s^v. Following the adaptation from a single assignment to a set will increase the computation exponentially and sequential validation will validate the same configuration multiple times.

In our implementation we use a workaround which solves this issue and which maintains the complexity of exploring sequential adaptations linear in the number of states. Once we have identified S_j and k by applying a_{S_i}, we apply k to the global assignment $a_g \in G$, obtaining one and only one $a'_g \in G$. If a'_g does not violate any constraint, that is if it is a valid assignment we can re apply the *localize* function from G to M_{S_j} obtaining at most one $a'_{S_j} \in M_{S_j}$. If a'_{S_j} does not exists, the application is stable in S_j and the we can terminate the exploration. Otherwise we iterate this process.

Note that if we hit the same state twice with the same assignment, meaning that two assignments $a^y_g = a^z_g$ exist and are applied to the same state, then the adaptation is looping and the execution will not terminate without an external input. To avoid looping and to report such anomalies, it is necessary to trace both visited states and global assignments with which they have been explored. An algorithm using this implementation is shown in Section 7.6.1.

6.1.2 Applying the Enumerative Approach

Once the state matrices for the various states have been computed, enumerative algorithms can use them to detect faults by iterating on their assignments. Property designers willing to validate an application with novel properties have to implement algorithms to detect them. Such algorithms have a common structure iterating on each assignment of each state matrix. For each assignment a particular validation is performed. The base complexity of these algorithms is $O(S * 2^n)$ where n is the number of PCVs. In Chapter 7 we present algorithms to detect the properties discussed in Section 5.2

6.2 OBDD-Based Approaches

In examining some of the open source CAAAs we found on the Web, we noticed that the number of states in a CAAA is generally limited, as each state corresponds typically to a different behavioral modality. On the other hand, we also noticed that the number of propositional context variables one would identify for these CAAAs varies between 15 and 20. The size of the state matrix and the time required to analyze it are in the worst case exponential in the number of variables, and so 20 variables may require too much memory to store all the state matrices.

To address this problem, we have defined algorithms that detect faults in an A-FSM through exploration of a *symbolic* representation of the state space of the A-FSM. In particular, we encode A-FSMs with Ordered Binary Decision Diagrams (OBDDs) using a technique similar to those used by model checking [CGP99] and planing [JV99] tools. The key idea is that states and transitions are encoded by means of Boolean formula, and the Boolean formula are then manipulated using OBDDs, which are a compact and efficient representation for Boolean formula; typically, OBDDs allow for a reduction of several order of magnitude in the size of a corresponding explicit-state representation.

The Boolean formula used to encode an A-FSM is composed by various sets of Boolean variables representing respectively: K_C for the current PCVs assignment, K'_C for the PCVs assignment after an adaptation rule has been applied, K_S for the current state, K'_S for the destination state after an adaptation rule has been triggered and K_R for the adaptation rules. We introduce $K_C = |\mathcal{C}|$ variables in a vector $\overline{c} = (c_1, \ldots, c_{K_C})$ to represent the truth values of the Boolean propositional context variables when a rule is triggered and an additional set of K'_C variables to describe the context after the action has been applied; for the A-FSM of *PhoneAdapter* there are 24 (12 plus 12) such variables. Note that we do not use Boolean variables to encode priority, since priority is handled as part of the derivation algorithm, as explained below. The number of Boolean variables required to encode the states $\mathcal{S}$ of an A-FSM M is $K_S = \lceil log_2|\mathcal{S}|\rceil$; for instance, for the A-FSM of *PhoneAdapter*, $K_S = \lceil log_2|9|\rceil = 4$. Let $\overline{s} = (s_1, \ldots, s_{K_S})$ be the vector of K_S Boolean variables encoding $\mathcal{S}$; for instance, the vector $(0, 0, 0, 0)$ can be used to encode the first state (corresponding to the Boolean formula $\neg s_1 \wedge \neg s_2 \wedge \neg s_3 \wedge \neg s_4$), the vector $(0, 0, 0, 1)$ the second state, and so on. We introduce K'_S more variables to encode the destination state of a transition by means of a vector $\overline{s'} = (s'_1, \ldots, s'_{K_S})$. Finally we introduce $K_R = |\mathcal{R}|$ variables in a vector $\overline{r} = (r_1, \ldots, r_{K_R})$ to identify the different rules in $\mathcal{R}$; for the A-FSM of *PhoneAdapter* there are four such variables.[1]

The Boolean variables introduced above support the encoding of a *state activation BDD* that characterizes the evolution of the CAAA from a particular state as the transitions in that state are triggered.

Consider the example M_1 in Section 6.1. In state S_1 there is only one adaptation rule R_0 adapting to S_0 when only one of the PCVs P_a and P_b is satisfied. Figure 6.1 represent the state activation BDD for state S_1. All the path leading to 1 trigger an adaptation, while all the paths leading to 0 represents stable configurations in S_1 To represent the two states we need one boolean variable S for the current state and one variable S' for the future state. When S or S' are assigned with 0 they represent S_0, otherwise when they are assigned with value 1 they represent S_1. Note that since we are composing the state activation BDD of state S_1 the boolean variable S has always value 1 therefore it values does not contribute to the state activation BDD. The value of S will be used when merging multiple states together. To discriminate between the two rules we need one boolean variable R. For each PCV x we use one boolean variable P_x to describe the current assignment and another boolean variable P'_x to describe the context after the adaptation has been applied. Notice that since there is a constraint saying that if P_b is not satisfied then P_a must be, two paths in the BDD have been removed because they where violating such constraint. Also notice that since the action of R_1 does not change the PCVs assignments the subtree representing the future assignments (the one with the variables P'_x) is the same as the subtree with the current PCVs assignments.

[1]This is an example of how we can exploit the similarity of the four different rules labeled *ActivateDriving* in Table 5.1, effectively reducing 19 rules to 16 rules represented by four variables.

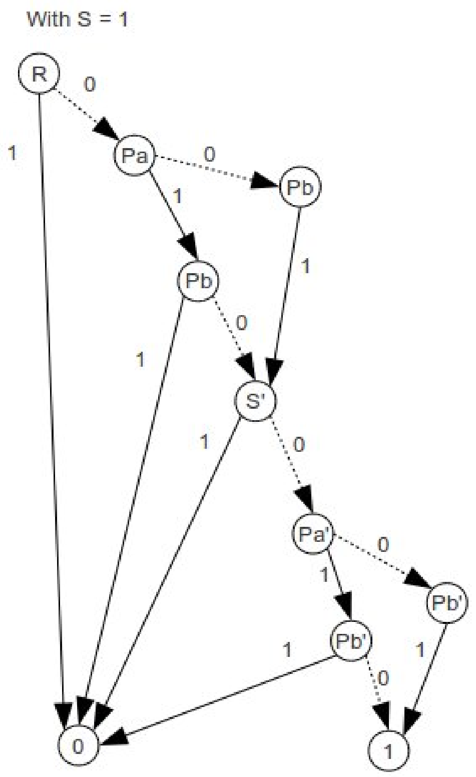

Figure 6.1: Activation BDD for state S_1.

Algorithm 3 computes the state activation BDD as follows. For each priority level (Line 3), starting from the highest priority to the lowest, we compute, for each rule at that priority level (Line 5), a BDD representing the activation of each rule (Lines 7–9). The disjunction of all of these BDDs (Line 10) is the state activation BDD returned by the algorithm. The activation of a rule within a certain priority level prevents rules with lower priorities to be triggered with the same inputs. The BDD defined in Line 2 records executions of all the rules predicates at higher priority and excludes them from being reused (Line 7 and Line 12). Note that the space of feasible inputs is further reduced by removing all those solutions that do not satisfy the global constraints. The BDD *GcBDD* given as input contains the conjunction of all the global constraints. The activation of each single rule (Lines 7-9) is encoded as the conjunction of the current state's encoding, the rule's encoding, the destination state encoding, the trigger input and the context after the rule's action has been applied. The trigger input is the one computed in Line 7. The context resulting from the action is computed by *applyActionToPredicate* in Line 8. This function takes as input a BDD representing the trigger input, reassigns all the variables modified by the action, reapplies all the constraints and swaps the predicate boolean variables into the action variables. Note that since the post action assignment is computed from the BDD in Line 7 it is also affected by constraints and by higher priority rules.

The complexity of this algorithm is $O(P * |\mathcal{R}|)$, where P is the number of priority levels.

In the rest of this thesis we use the following notation to represent BDDs:[2] $\langle V_i{:}0\rangle$ means that

[2]This is the notation used by the open source library JavaBDD [Wha07] to print BDDs as strings.

Algorithm 3 State Activation BDD Generation

Input: *S*: a state of the A-FSM;

GcBDD: global constraint BDD.

Output: StateActivationBDD: state activation BDD.

```
1: BDD activation = 0
2: BDD exclusion = 0
3: for $i$ = VALUE_OF_HIGHEST_PRIORITY to VALUE_OF_LOWEST_PRIORITY do
4:    BDD esclusionAtPriority $= 0$
5:    for each $R \in S$.getActiveRulesAtPriority($i$) do
6:       esclusionAtPriority $=$ esclusionAtPriority $\wedge$ $R$.getPredicate()
7:       BDD triggerInput $=$ ($R$.getPredicate() $\wedge\neg$exclusion) $\wedge$ GcBDD
8:       BDD futureAssignments $=$ $R$.applyActionToPredicate(triggerInput)
9:       BDD ruleActivation $=$ triggerInput $\wedge$ $R$.getDestState() $\wedge$ $R$.getEncoding() $\wedge$
         $R$.getDestState().getFutureEncoding() $\wedge$ futureAssignments
10:      activation $=$ activation $\vee$ ruleActivation
11:   end for
12:   exclusion $=$ exclusion $\vee$ exclusionAtPriority
13: end for
```

the variable V_i must be false to satisfy the enclosing BDD, while $\langle V_i{:}1\rangle$ means V_i must be true. We indicate a conjunction by placing multiple variable assignments within the same pair of angle brackets and disjunction by a sequence of angle-bracketed variable assignments. Thus, $\langle V_i{:}0, V_j{:}\ 0\rangle$ indicates that both V_i and V_j must be false to satisfy the BDD, while $\langle V_i{:}\ 0\rangle\langle V_j{:}0\rangle$ means that at least one of V_i and V_j must be false to satisfy the BDD.

The state activation BDD for state *Sync* of *PhoneAdapter* can be represented using this notation as follows:

$$\langle 0{:}0,\ 9{:}0,\ 12{:}0,\ 21{:}0,\ 28{:}1,\ 29{:}0,\ 30{:}0,\ 31{:}1,\ 32{:}0,\ 33{:}1,\ 34{:}1,\ 35{:}1\rangle$$

In this and all subsequent examples for *PhoneAdapter*, i : 1 means that variable number i is true and i : 0 means that variable number i is false. Table 6.1 describes all the set of variables used to encode *PhoneAdapter*. Variables 0–11 represent the propositional context variables. The current state is encoded with variables 24–27, which are absent here because, as described above, a state activation BDD does not explicitly encode its associated state. The destination state is encoded in variables 28–31, and rule identities are encoded in variables 32–35. Variables 12–23 represent the PCVs assignment after the active rules have been applied. In this example the only active rule is DeactivateSync. Thus, the state activation BDD of *Sync* encodes fact that when the predicate ⟨0:0, 9:0⟩ is satisfied, rule ⟨32:0, 33:1, 34:1, 35:1⟩ will adapt the A-FSM to state ⟨28:1, 29:0, 30:0, 31:1⟩. with no changes to the PCVs assignment (variables ⟨12:0, 21:0⟩). Table 6.2 shows in details the BDD encoding of the current and future states. Table 6.3 shows the rule BDD encoding. This encoding will be used in Chapter 7 to report faults detected with the

Table 6.1: Variable encoding in PhoneAdapter's global activation BDD

Indexes	Variables
0–11	$K_C = A_{gps}, B_{gps}, C_{gps}, D_{gps}, E_{gps}, A_{bt}, B_{bt}, C_{bt}, D_{bt}, E_{bt}, A_t, B_t$
12–23	$K'_C = A'_{gps}, B'_{gps}, C'_{gps}, D'_{gps}, E'_{gps}, A'_{bt}, B'_{bt}, C'_{bt}, D'_{bt}, E'_{bt}, A'_t, B'_t$
24–27	Current state K_S
28–31	Destination state K'_S
32–35	Rules K_S

Table 6.2: State encoding in PhoneAdapter's global activation BDD

State	Current encoding	Future encoding
General	$< 24:0, 25:0, 26:0, 27:0 >$	$< 28:0, 29:0, 30:0, 31:0 >$
Outdoor	$< 24:1, 25:0, 26:0, 27:0 >$	$< 28:1, 29:0, 30:0, 31:0 >$
Jogging	$< 24:0, 25:1, 26:0, 27:0 >$	$< 28:0, 29:1, 30:0, 31:0 >$
Driving	$< 24:1, 25:1, 26:0, 27:0 >$	$< 28:1, 29:1, 30:0, 31:0 >$
DrivingFast	$< 24:0, 25:0, 26:1, 27:0 >$	$< 28:0, 29:0, 30:1, 31:0 >$
Home	$< 24:1, 25:0, 26:1, 27:0 >$	$< 28:1, 29:0, 30:1, 31:0 >$
Office	$< 24:1, 25:1, 26:1, 27:0 >$	$< 28:1, 29:1, 30:1, 31:0 >$
Meeting	$< 24:0, 25:0, 26:0, 27:1 >$	$< 28:0, 29:0, 30:0, 31:1 >$
Synch	$< 24:1, 25:0, 26:0, 27:0 >$	$< 28:1, 29:0, 30:0, 31:1 >$

symbolic algorithms

The conjunction of a state activation BDD and a BDD encoding an assignment of values to the propositional context variables produces a *contextual BDD* encoding the adaptations triggered by that input from the state associated with the state activation BDD.

Finally, the *global activation BDD* $\mathcal{B}$ encodes all the states and transitions of the A-FSM M and is derived by first conjoining each state activation BDD with the encoding of its associated state, and then computing the disjunction of the resulting state-specific BDDs over all states S:

$$\mathcal{B} = \bigvee_{S \in \mathcal{S}} (S.\text{getEncoding()} \wedge S.\text{getStateActivationBDD()})$$

Figure 6.2 depicts the global activation BDD for the example A-FSM M_1. The two state activation BDDs for states S_0 and S_1 (which is the one in Figure 6.1) have been put together in conjunction with the boolean variables used to encode the states. The resulting BDD represents the formula:

$$(\neg S \wedge S_0.getActivationBdd()) \vee (S \wedge S_1.getActivationBdd())$$

.

Independently from the state all the paths of Figure 6.2 reaching 1 represent a PCV assignment triggering an adaptation. Vice-versa all the paths reaching 0 are stable.

Table 6.3: Rule encoding in PhoneAdapter's global activation BDD

Rule	Encoding
ActivateOutdoor	$< 32:0, 33:0, 34:0, 35:0 >$
DeactivateOutdoor	$< 32:1, 33:0, 34:0, 35:0 >$
ActivateJogging	$< 32:0, 33:1, 34:0, 35:0 >$
DeactivateJogging	$< 32:1, 33:1, 34:0, 35:0 >$
ActivateHome	$< 32:0, 33:0, 34:1, 35:0 >$
DeactivateHome	$< 32:1, 33:0, 34:1, 35:0 >$
ActivateOffice	$< 32:0, 33:1, 34:1, 35:0 >$
DeactivateOffice	$< 32:1, 33:1, 34:1, 35:0 >$
ActivateMeeting	$< 32:0, 33:0, 34:0, 35:1 >$
DeactivateMeeting	$< 32:1, 33:0, 34:0, 35:1 >$
ActivateDriving	$< 32:0, 33:1, 34:0, 35:1 >$
DeactivateDriving	$< 32:1, 33:1, 34:0, 35:1 >$
ActivateDrivingFast	$< 32:0, 33:0, 34:1, 35:1 >$
DeactivateDrivingFast	$< 32:1, 33:0, 34:1, 35:1 >$
ActivateSynch	$< 32:0, 33:1, 34:1, 35:1 >$
DeactivateSynch	$< 32:1, 33:1, 34:1, 35:1 >$

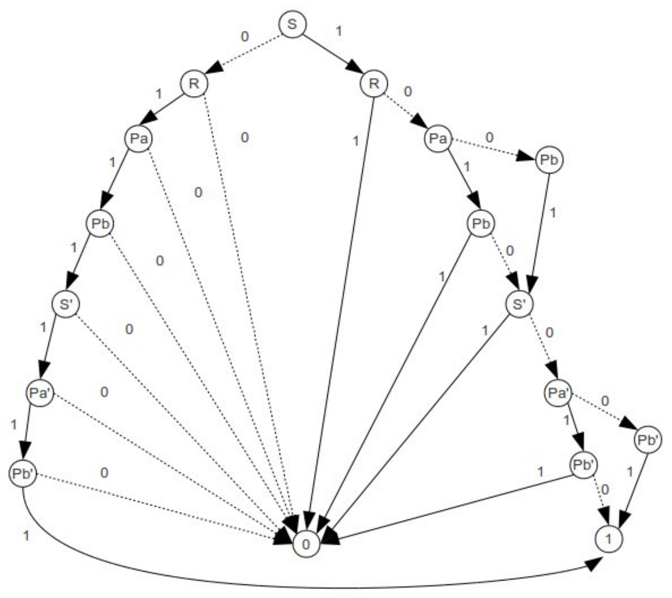

Figure 6.2: Global activation BDD for M_1.

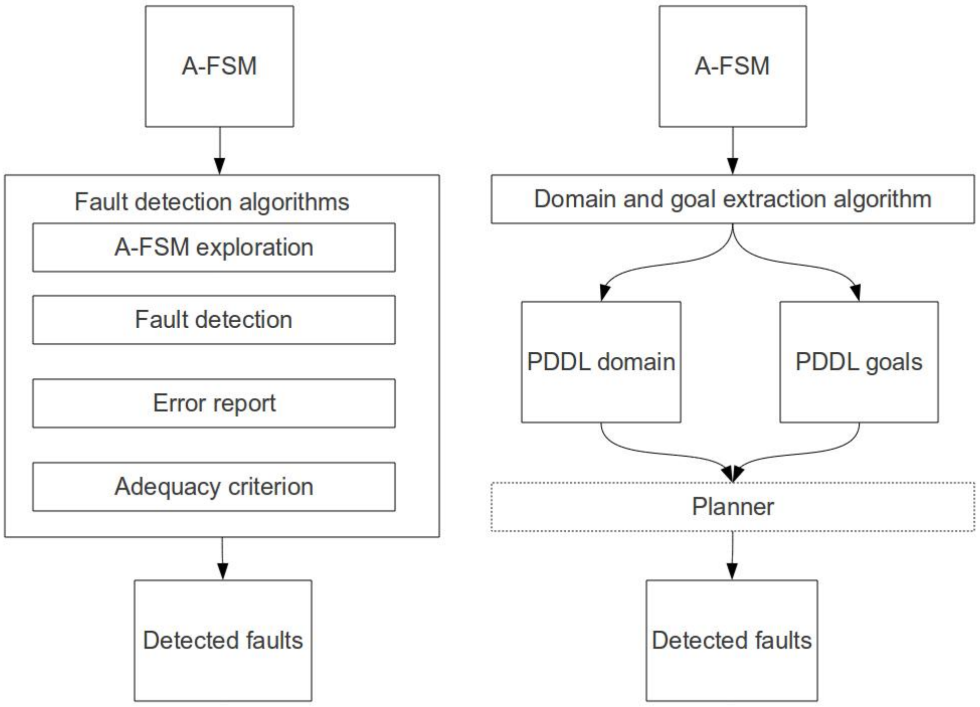

Figure 6.3: Differences between fault detection and PDDL extraction algorithms.

6.2.1 Applying the BDD-based approach

Once the state activation BDDs for the various states and the global activation BDD have been computed symbolic algorithms can use them to detect faults by iterating on their assignments.

All the BDD-based fault detection algorithms work in a similar way. At first they compute a BDD representing the violation of the property under validation. Once this BDD is computed, in most of the cases, its conjunction with the state activation BDD generates a disjunction of all the configurations violating the property (e.g., detected faults). In other term the algorithms generate a BDD containing all the detected faults. In order to be used by developers this BDD has to be converted into a human readable form. This can be done simply by looping state by state (and eventually rule by rule) and by extracting the detected faults for each state.

In Chapter 7 we present algorithms to detect the fault classes discussed in Section 5.2 implemented using both the global activation BDD an the state activation BDDs. In chapter 8 we evaluate the performance of our implementations of these algorithms.

Similarly to the enumerative approach, developers willing to validate an application with their own custom properties have to implement the related validation algorithms.

6.3 Planner-Based Approach

The third approach presented in this Section differs from the previous in how the A-FSM model is involved in fault detection. Figure 6.3 represent an overview of the execution flow of the previous techniques (on the left) and of this new approach (on the right). To validate an A-FSM model against a certain class of faults both enumerative and symbolic approaches explore the model with a fault-specific algorithm (the central block on the left). Fault detection algorithms performs four main activities: explore the A-FSM model, detect faults, report detected faults and decide when to stop searching.

Differently than enumerative and symbolic approaches the PDDL-based approach uses the A-FSM to extract a PDDL domain and PDDL goals then delegates a planner to perform the validation. For each class of faults a goal extractor algorithm uses the A-FSM model and the fault definition to generate a set of PDDL goals representing possible faults. In comparison with the fault detection algorithms the goal extraction algorithms are simpler because they only perform one of the four activities: defining how to detect a fault. The remaining three activities, exploring the A-FSM model, reporting detected faults and deciding when to stop the exploration are delegated to a planner.

The extracted PDDL domain represents the A-FSM model using the PDDL language. The extracted PDDL goals represent critical conditions which have to be validated. The delegated planner attempts to reach all the extracted goals starting from a given initial configuration. To facilitate the reader we describe how planners works in Section 2.4.

Planner reports successfully computed plans as the sequence of PDDL actions they had to perform to reach the goals from the given initial state. Due to a similarity between the A-FSM model and the PDDL language, plans generated by our technique contains a sequence of context-changes and adaptations which we encoded as PDDL actions. Such plans can be easily interpreted by a human and do not need any further computation.

In the remaining of this Section we introduce our PDDL-based approach which can be used as a fast prototyping technique.

6.3.1 Creating the Domain

The first step in using a planner for verifying a CAAA is to extract from the A-FSM a PDDL domain describing the CAAAs. The extraction includes the definition of types of variables in the planning domain, predicates and actions. We represent an A-FSM using only one variable type: *context*, which represents the current configuration of the context. We also introduce the concept of *state* and *priority* using fluents of PDDL 3.0 [GL97] by defining two non-parametric functions and by re-assigning their values when needed. States are identified with a unique numeric index. Priority is defined as an integer where 0 represent the highest and 10 the lowest.

The notion of *state* is obtained by means of a fluent [Thi05]. As mentioned in Section 2.4, the fluent calculus provides the concept of atomic properties with which values can be set, retrieved and compared. A fluent (*assign* (*state*) *s*) is applied whenever the A-FSM adapts into the state represented by s. The current state can be used in preconditions by using (= (*state*) *x*) which is true if x is the index representing the current state. For each propositional context variable $v \in \mathcal{V}$ we create a predicate (*is-true-v ?c context*), which if satisfied (or negated) encodes the fact that a certain propositional context variable is satisfied (or not). Similarly to those identifying the current state, these predicates have an instance of context as a variable. In contrast to state predicates, multiple of these predicates can be satisfied or negated at the same time.

In the real environment something may occur which will change the truth value of a PCV. We model these environmental events as actions that a planner may apply. For instance an event such as “the user has turned the GPS on” will be modelled as an action *satisfy-v* where v is the propositional context variable satisfied when GPS is on. We have $2 * |\mathcal{V}|$ such actions. If all the propositional context variables are independent of each other, the preconditions and effects of those actions would simply make the predicate either true or false:

```
(:action unsatisfy_C_bt
    :parameters (?c - context)
    :precondition (and
        (= (priority) 10) (is_true_C_bt ?c))
    :effect (and
        (not (is_true_C_bt ?c)) (assign (priority) 0)))
```

If some variables depend on each other (i.e., if the relevant entries exist in $\mathcal{G}$), preconditions and effects must apply these constrains. For instance, turning off GPS makes false all the other predicates using GPS; in the example below, `is_true_A_gps` encodes the fact that the GPS is on, and all the other predicates `is_true_*_gps` encode other GPS properties such as “the location is home”, etc.:

```
(:action unsatisfy_A_gps
    :parameters (?c - context)
    :precondition (and
        (= (priority) 10) (is_true_A_gps ?c))
    :effect (and
        (not (is_true_B_gps ?c)) (not (is_true_C_gps ?c))
        (not (is_true_D_gps ?c)) (not (is_true_E_gps ?c))
        (not (is_true_A_gps ?c)) (assign (priority) 0)))
```

These represent all the contextual changes that the system can apply independently from the rule’s effects, typically as a result of users’ actions. The verification of certain systems may require some of these changes to be disabled. For instance, if we are not interested in plans in which the user turns on (or off) the GPS manually, the predicates *satisfy-GPS-on* and *unsatisfy-GPS-on* can be removed from the domain.

Additionally, we create an action in PDDL for each adaptation rule in the A-FSM, and the planner can apply these actions appropriately. These actions accept two variables, one for the state and the other for the context. The preconditions encode the fact that the state should be one in which the rule that the action is modeling is active, and moreover the preconditions include the triggering predicate that should be satisfied for the rule to be active. The effect of these PDDL actions results in the transition to a new state and in the execution of the A-FSM actions from `ActionList`. The following is an example of an action deactivating the synchronization process of a mobile device with a base station, such as a home PC:

```
(:action rule_DeactivateSync
    :parameters (?c - context)
    :precondition (and
        (= (state) 8) (= (priority) 9)
        (not (or (is_true_B_bt ?c) (is_true_C_bt ?c))))
    :effect (and (assign (state) 0) (assign (priority) 0)))
```

The planning domain loops through a priority queue in which actions with higher priority are always applied first (where each adaptation rule encoded as a PDDL action is assigned the priority from the A-FSM model). This encodes the fact that if an adaptation rule can be performed in the current state with the current priority, nothing else can be applied. Otherwise the planner increases the priority value. To increase the level of priority an action *set_priority_p*, where p is the new level of priority, is extracted from the A-FSM. The precondition of these actions is that there are no adaptation rules satisfiable in the current state, with the current level of priority. Every time a rule is applied or the context changes, the priority is reset to 0.

```
(:action set_priority_5
    :parameters (?c - context)
    :precondition (and
        (= (priority) 4)
        (or
            (= (state) 0) (= (state) 1)
            (= (state) 2) (= (state) 3)
            (= (state) 4) (= (state) 5)
            (and (= (state) 6)
                 (not (and (is_true_A_t ?c)
                           (is_true_E_bt ?c))))
            (= (state) 7) (= (state) 8)))
    :effect (assign (priority) 5))
```

In our implementation we assigned to all the context changes a priority lower than all the priority assigned to the adaptation rules. This force the planner to evaluate all the adaptation rules before applying new contextual changes. We forced this execution because it is similar to the behavior of most the system we observed, in which the thread evaluating the adaptation logic updates the sensed context only at the beginning of its computation.

Architectures in which the adaptation logic can abort and restart the computation if a change in the context has been sensed can be represented by giving top priority to those PDDL actions representing environmental changes.

Notice that mixing the priority of rules and context changes is potentially dangerous because the priority of rules only describes the order with which they are evaluated and not their reactivity. Saying that a contextual changes has an higher priority value than a certain adaptation rule means that before evaluating a certain rule the adaptation logic will sense the context and apply changes to the PCVs representing that specific context. For instance, in the conveyor belt example by Lu at al. [LCT08] RFIDs values are read multiple times during the computation, to follow the package movements on the belt. In this and similar cases developers need to assign the correct priority to each context changes.

6.3.2 Applying the PDDL-Based Approach

Starting from the A-FSM model of the CAAA under test PDDL domains and goals are extracted. For each fault class various goals are extracted one for each fault's instance that can be detected. The number of PDDL goals is finite and it is proportional to the number of rules and states in the original A-FSM model.

Plans reaching the extracted PDDL goals represent either the prove that a certain property, is satisfied, either contains the counter-example of a detected fault, depending on the extracted goals. We provide an implementation of the goal extraction algorithms for the properties discussed in Section 5.2.

The nondeterministic activation goal generation algorithm creates a goal for each pair of rules which are active with the same priority in any of the states. A fault is found when the planner is in the state under test with the correct priority and when both the rules under test are satisfied. In the worst case the number of these goals is $\sum_{s=0}^{|S|} |R_s|^2$, where $|R_s|$ is the number of active rules in state s.

The state liveness violation goal generation algorithm creates a goal for each state, and a fault is found if there is no plan satisfying (= (*state*) *s*) where s is the state under test.

Similarly the rule liveness violation goal generation algorithm creates a goal to impose that a rule associated with a state is active, and this is repeated for all rules associated with all states. A fault is reported if no plan can be generated for a particular rule to be active in a particular state having the rule's priority. The number of generated goals for this property is $\sum_{s=0}^{|S|} |R_s|$.

The state invariant violation goal generation algorithm creates a goal for each state in which an invariant has been specified. A fault is reported if there is a path reaching the state under test and negating the invariant.

Finally, each goal is initialized with an initial condition in which the state is the initial state and the priority is the highest (i.e. (= (*state*) 0) (= (*priority*) 0))

6.4 Related Work

Among the validation techniques there are some which validate the application with a set of given patterns. Roman et al. define Mobile UNITY, an extension of the UNITY notation and proof logic to the verification of mobile systems [RMP97]. Given mobile applications specified in Mobile UNITY and associated specified properties, Mobile UNITY is able to verify the application against the specified properties. This work mainly focuses on verifying the mobility aspects of the application, whereas our approach is concerned with discovering faults in an application's context-awareness and adaptation behavior.

Xu and Cheung propose inconsistency detection in context-aware applications whereby patterns identify conflicts among context inputs at run-time before they are fed to an application [XC05, XCC06]. The patterns are defined by engineers based on their understanding of relevant mathematical and physical laws. This work focuses mainly on verifying the correctness of the context inputs themselves. In contrast, we assume context inputs to be consistent and then evaluate them within the predicates of adaptation rules to check whether there are faults in the formulation or triggering of rules. We also consider intrinsic relationships among context variables, in particular the delays resulting from asynchronous update of context variables having different refresh rates, and we identify the adaptation faults that may arise as a result.

Several researchers from the testing community have begun to target the validation of CAAAs [LCT06, WER07a]. Although we share their goal of detecting faults in CAAAs, our approach is fundamentally different, employing static analysis of adaptation models, while theirs are centered primarily on test selection and runtime analysis.

Efforts for testing rule-based systems (the main adaptation mechanism used by CAAAs) have focused on the development of coverage criteria for exercising single rules or rule chains [Bar97, Gup98]. In contrast, we statically analyze a set of rules to identify variables that may trigger multiple rules concurrently or multiple commutations of variables within the same rule, leading to adaptation failures.

When designing sequential digital circuits in which multiple signals are input to a network of logic gates, engineers must avoid *hazards* and *races*, which may produce incorrect outputs. Unger [Ung95] and Hauck [Hau95] summarize timing problems in sequential circuits and describe techniques for predicting and correcting them. Our work is based in part on the insight that processing of context inputs induces similar kinds of hazards and races in CAAAs, and we provide an appropriate formulation of such faults using our adaptation models. In particular, in our work we detect faults in which the choice of adaptation rules to trigger and the order in which to trigger them depends on how long a context input value holds.

Finite-state models have been used extensively to represent and verify properties of systems. Some work similar to our own has been done in the context of requirements engineering. Heitmeyer et al. use finite-state models to discover inconsistencies in SCR specifications [HJL96], and Heimdahl and Leveson use finite-state models to discover inconsistencies in RSML specifications [HL96]. While the classes of inconsistencies that they detect are characteristic of requirements specifications, the fault patterns that we detect are characteristic of CAAAs. Thus, although there are some similarities between our fault patterns and their classes of inconsistencies, certain classes appear to arise only in CAAAs, notably instability faults (described in Section 5.2.5) and context hazards (described below in Appendix A). Finally, we also note that model checkers use finite-state representations extensively to model concurrent systems and to verify their temporal properties [CES86], and static checkers for meta-programming languages use such models to detect potential vulnerabilities in generated code [WGSD07]. Similarly, our analysis operates on a finite-state model, but we have extended it to incorporate context information and have tailored the analysis to focus on properties that are of particular relevance to CAAAs.

As we mention, several failures are caused by errors in the sensed context variables which are computed and used for adaptations. This opens the problem of verifying the correctness of the read data. Tse et al. [LCT06] suggest a formal language to create constraints between inputs and outputs and apply metamorphic testing on those constraint in order to find out how the program handles noises. This is a sort of contract-based context-aware testing but as it happens for other context based approaches, if the logic becomes too complicated it may be difficult or impossible to represent it with a formal language. Moreover, this formal description tends to become a twin implementation which will need to be maintained.

The idea of using a planner instead of a model checker to validate a model or to find a suitable plan to a desired configuration exists in the literature. Stefan Edelkamp has studied limit and possibilities of using planners to validate models with particular focus on protocol validation [Ede08]. In this work Edelkamp exposes benefits and drawbacks of planners, underlining the simplicity with which planners can be used. Albarghouthi et al. have shown the potential of using planners for solving verification tasks [ABM09].

Chapter 7

Detecting Faults

This chapter shows how concepts introduced in the previous chapters can be applied to detect faults. In Chapter 4 we described seven applications suffering from context awareness faults and we have classified these faults according to various aspects. In Chapter 5 we presented a model representing behaviors and adaptations, and we described various properties that when violated, lead to faults. In Chapter 6 we introduced three novel techniques that analyze the A-FSM to detect violations of our properties.

In this chapter we discuss in detail the algorithms and setups required to detect faults in the A-FSM models with our techniques. To facilitate the comprehension and the comparison we do not discuss the three techniques separately. Instead we compare them by fault in order to underline differences and benefits for each different fault.

In Chapter 8 we evaluate our novel validation techniques by applying their implementation to the applications presented in Chapter 4. We will show which faults have been detected, and we compare both performance and error reports.

7.1 Base Implementation and Setup

Figure 7.1 depicts the validation flow for the three techniques. Our techniques start from the A-FSM model of the application under validation. We provide a simple Java 6 API capable of representing the A-FSM in memory.

Our A-FSM generation API provides a simple but effective program interface to create an A-FSM programmatically. This API is intended for developers willing to write parsers to extract the A-FSM from the configuration files of their CAAAs. The API can also be used to create the A-FSM programmatically within a Java application.

In this research we have used the A-FSM generation in two ways. First we used it to encode the A-FSMs of all the application we validated. Second, we used it to generate the A-FSM model of a set of synthetic models generated randomly for benchmarking purposes.

Listing 7.1: Example use of the A-FSM Generation API

```
/** Define an A–FSM */
AdaptationFSM afsm = new AdaptationFSM();

/** Define context variables */
ContextVariable time = context(afsm, "Time", 1);
```

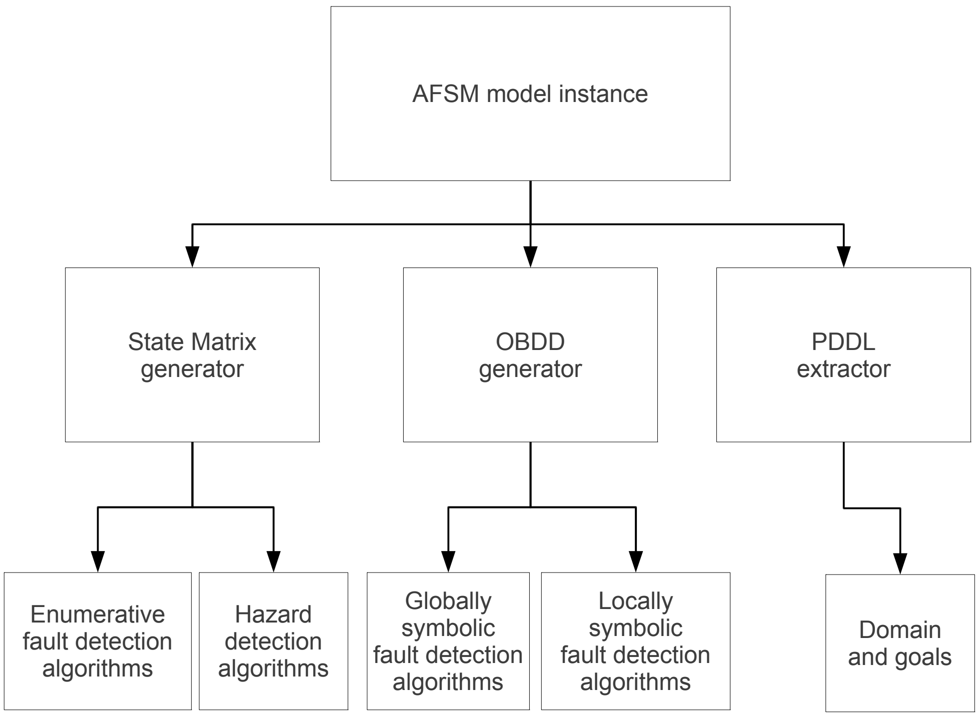
AFSM model instance
State Matrix generator
OBDD generator
PDDL extractor
Enumerative fault detection algorithms
Hazard detection algorithms
Globally symbolic fault detection algorithms
Locally symbolic fault detection algorithms
Domain and goals

Figure 7.1: Sub-models extraction from the A-FSM.

```
ContextVariable alarm = context(afsm, "Alarm", 2000);

/** Define PCV */
// start <= stop
Predicate startLesserEqualStop =
                variable(afsm, "StartLesserEqualStop", time);
// time < stop_time
Predicate beforeStop =
                variable(afsm, "BeforeStop", time);
// time >= start_time
Predicate afterEqualStart =
                variable(afsm, "AfterEqualStart", time);
Predicate enableSound = variable(afsm, "EnableSound", alarm);
Predicate enableVibration = variable(afsm, "EnableVibration", alarm);

/** Define constraint */
// beforeStop != afterStart
Constraint c_time = new Constraint(beforeStop, afterEqualStart);
afsm.addConstrain(c_time);

/** Define states */
State initial = new State("Initial");
afsm.addState(initial);
afsm.setInitialState(initial);

State day = new State("Day");
afsm.addState(day);
```

```
33
34  State night = new State(”Night”);
35  afsm.addState(night);
36
37  /** Define predicates */
38  // time >= start_time && time < stop_time
39  Predicate isDay  = or(and(startLesserEqualStop, afterEqualStart, beforeStop),
40                 and(not(startLesserEqualStop), or(beforeStop, afterEqualStart)));
41  Predicate isNight = not(isDay);
42  Predicate dayMode = and(enableSound, not(enableVibration));
43  Predicate nightMode = and(enableVibration, not(enableSound));
44
45  /** Define rules */
46  AdaptationRule activateDayMode = new AdaptationRule(”ActivateDayMode”, day);
47  activateDayMode.setCondition(isDay);
48  activateDayMode.setAppliedAction(dayMode);
49  activateDayMode.setPriority(Rule.DEFAULT_PRIORITY);
50
51  AdaptationRule activateNightMode = new AdaptationRule(”ActivateNightMode”, night);
52  activateNightMode.setCondition(isNight);
53  activateNightMode.setAppliedAction(nightMode);
54  activateNightMode.setPriority(Rule.DEFAULT_PRIORITY);
55
56  /**    Add rules into states */
57  //(initial,{activateDayMode, activateNightMode})
58  initial.addRule(activateDayMode);
59  initial.addRule(activateNightMode);
60
61  //(day,{activateNightMode})
62  day.addRule(activateNightMode);
63  day.setHoldCondition(isDay);
64  day.setInStateAssumptions(enableSound);
65
66  // (night,{activateDayMode})
67  night.addRule(activateDayMode);
68  night.setHoldCondition(isNight);
69  night.setInStateAssumptions(not(enableSound));
70
71  /** Compute the input matrices */
72  afsm.loadInputSpaces();
```

Listing 7.1 shows the Java code to create the A-FSM for Timerrific. In line 2 a new instance of the object representing the A-FSM model is created. In lines 5–6, context variables are created and added to the model. In this example there are only two context variables; one representing the time, and one indicating the alarm status.

Once the context variables have been initialized, they can be used to define PCVs, in lines 9-19. To avoid false positives in lines 23-24 a global constraint is defined, to ensure that the two predicates identifying the time during the day and during the night cannot be simultaneously satisfied.

In lines 25-35 state *Initial*, *Day* and *Night* are added to the A-FSM. State *Initial* is also flagged as the initial state. Once those states and PCVs are defined it is possible to create rules. In lines 46-69 two rules are defined and added to the states. Notice that the formal definition of rule in the A-FSM model identifies with a rule a single directed transition between two states. With this representation adaptations with the same trigger predicate and with the same final state are represented with a different rule for each source state. This representation is formally correct, because in each state each rule can have a different priority or a different action. However in practice, if two rules differ only in their source state, it is a waste of memory to represent them with two separate instances. To minimize the memory usage the Java class AdaptationRule defines a prototype of rule that can be associated with multiple source states. In lines 46-54 two prototypes of rules are defined, and in lines 56-69 they are used to define four different adaptation rule instances.

At this point the A-FSM is ready to use. For instance we could use it to generate reduced state matrices as is done in line 72. Similarly the A-FSM could be used to generate an OBDD or to create the PDDL domain and goals files. Separate API calls are provided for all these operations.

7.1.1 From the A-FSM to State Matrices

Starting from the A-FSM, the creation of the state matrices is a an iterative process that is linear in the number of states and exponential in the number of PCVs in that state.

For each state, all PCVs that are part of either a predicate or an action of at least one adaptation rule are sorted and associated with an increasing index. Next all possible assignments to the indexed PCVs are enumerated forming bit strings of all the assignments. Finally bit strings are applied to predicate of the rules defined for the state. If a bit string triggers at least one adaptation then it is inserted in the local state matrix along with the triggered rule(s).

As explained in the rest of this chapter, enumerative algorithms use stable PCV configurations of a state as starting points for the verification, simulating changes in the context and looking up the rules that are triggered by the changes in the state matrix. Stable configurations are accessed sequentially, while the state matrices are accessed randomly.

Our Java implementation uses two data structures for better performance. State matrices are represented with *HashMaps* using the bit strings as indexes, and the list of triggered rules as values. An additional *HashSet* is used to store all the configurations violating a constraint. Our algorithms enumerates over all the possible bit strings discarding those which are in the *HashSet* and using the *HashMaps* to quick access the triggered adaptations rules.

Detected faults are generally reported as tuples in the form $(S, \mathcal{V}, \{r_i, r_j, ...\})$ where S is the state matrix under test, $\mathcal{V}$ is the bit string corresponding to the faulty assignment and $\{r_i, r_j, ...\}$ is the set of rules triggered by $\mathcal{V}$ in state S. Such tuples are a compact representation of adaptations in the state matrix. Faults caused by a missing adaptation are reported by mentioning that an expected tuple is missing. Faults involving a sequence of adaptations are reported with the sequence of tuples representing such adaptations. This representation has the advantage of being readable. Developers can use it without any additional computation. One fault is reported for each assignment in $\mathcal{V}$ that can trigger the fault. We notice that in certain situations, several assignments may produce the same fault, causing it to be reported several times. Although this is formally correct, it makes it hard for developers to distinguish faults that are different from faults that have been reported multiple times. To minimize this side effect, in our implementation we introduced in the bit strings the concept of “don’t care” (“*”) for PCVs not involved in the fault, hereby logarithmically reducing the number of reported tuples by the number of unrelated PCVs. This is similar to what OBDDs do to minimize the number of internal nodes. With this optimization the number of tuples reported for each fault is $O(2^v - 1)$ where v is the number of PCV causing the fault. For instance, assuming PCVs A, B, C, a fault caused by $A \wedge C$ will be reported with one tuple containing the bitstring $1 * 1$. A fault cause by $A \vee C$ will be represented by three tuples containing $0 * 1$, $1 * 0$ and $1 * 1$.

7.1.2 From the A-FSM to OBDDs

We demonstrate that OBDD-based approaches scale better as the size of the A-FSM increases, because instead of iterating on all inputs, states and rules explicitly, such approaches group all solutions symbolically using OBDDs. In this chapter we describe our two classes of OBDD-based algorithms: (1) *globally symbolic* algorithms that use the global activation BDD and that computes the whole A-FSM symbolically, and (2) *locally symbolic* (*hybrid*) algorithms that verify the A-FSM symbolically using the state activation BDD when convenient and iterates otherwise.

We call *globally symbolic* algorithms those algorithms using our OBDD-based representation of the state machine which to validate the whole A-FSM symbolically by using the global activation BDD. The benefit of this approach is that the computation is done once symbolically over an OBDD representing the whole A-FSM. The disadvantage of this approach is that creating, using and decoding example solutions from an OBDD all have a computational cost.

We call *locally symbolic* (*hybrid*) those OBDD-based algorithms that verify the A-FSM symbolically using the state activation BDD when convenient and iterates otherwise. The base idea behind the locally symbolic algorithms is to minimize the overhead cause d by the composition and decodification of the OBDD by iterating on the states instead. The intuition is that, since the number of states is relatively small, it would be feasible to explore them iteratively and also, since state activation BDDs strongly differ from each other then composing and using the global activation BDD should be computationally more expensive than simply iterating on few states. In other terms the locally symbolic algorithm represent an attempt to minimize the overhead of using a symbolic approach by applying a symbolic computation only when computationally convenient.

In our implementation we start from our Java representation of the A-FSM, and we generate five sets of Boolean variables as explained in Section 6.2. Then we compute the state activation BDD used by the locally symbolic algorithms by means of Algorithm 3. For the globally symbolic algorithms we also need to compute the global activation BDD by merging all the state activation BDD with the formula presented at the end of Section 6.2 Our implementation is based on JavaBDD [Wha07] which is a Java wrapper for the popular BuDDy [Coh04] package for OBDDs.

Both the locally and globally symbolic algorithms reports faults as OBDDs, which need to be decoded into a more comprehensible representation to be used by developers. The locally symbolic approach reports one OBDD for each state containing all the faults detected in that state. Such faults are then extracted from the OBDD. The method of extraction changes from fault class to fault class and thus cannot be generalized. In our implementation each fault detection algorithm also extracts and reports faults at the end of the computation. The globally symbolic approach encodes all the faults of the whole A-FSM in a single OBDD. To extract them, faults are first divided state by state and then are decoded as for the locally symbolic approach. In the rest of this chapter we show the computational cost of such operations.

7.1.3 From the A-FSM to PDDL

The PDDL-based approach differs from the previous ones because the validation is not performed sequentially rather than exhaustively.

Unlike the previous approaches, which employ our custom algorithms, the PDDL solution uses third-party planners to perform the verification. The domain and goal extractor compile the A-FSM model into a domain containing actions and predicates representing the A-FSM, as described in Section 6.3.1, and into a set of goals. Each goal represents a possible instance of a fault the planner can detect. The number of generated goals increases with the number of states and rules, but notice that each goal is independent, and therefore goals can be explored in parallel.

The generated goals can be divided into two categories: (1) those that when achieved show a fault, and (2) those that when achieved imply that the application is correct. For testing purposes the former are more useful than the latter because the produced plan is an execution trace showing how to trigger the discovered fault. In the latter case, if the goal cannot be reached, it means that there is a fault but no output is produced. Out of the four patterns of fault described in Section 5.1.2, rule liveness and reachability fault are revealed when no plan is found, while determinism and state invariant faults are revealed when a plan can be found.

Since the verification is performed by third-party components, the details of the results may depend on the actual planner being used. In particular, if a certain planner has limitations in the size of the goals that can be analyzed, the planner may terminate with an error. We ran our experiments using MIPS-XXL [EJN06] and SGPlan [CwHW04]. The PDDL 3.0 language specification does not allow disjunction in goal definitions. However, in certain goals we had to check if a certain predicate, containing a disjunction, was satisfiable. This issue can be addressed by applying De Morgan's laws, but this results in an increase of the goal files beyond 1024 bytes of PDDL code, which seems to be the upper limit for certain planners, and thus care must be taken in their choice.

When a fault is found, planners report the sequence of changes in the context and of adaptations from the initial state to the violation. Although a single fault is reported for each goal, the error report is clear and contains a counterexample for the detected fault.

Listing 7.2: Extracted Domain for PhoneAdapter

```
(define (domain PhoneAdapter)
        (:requirements :strips :typing :equality :disjunctive-preconditions :fluents)
        (:types context)

        (:predicates
                (is_true_B_bt ?c - context)
                (is_true_E_bt ?c - context)
                (is_true_D_gps ?c - context)
        (is_true_B_gps ?c - context)
        (is_true_A_gps ?c - context)
        (is_true_D_bt ?c - context)
        (is_true_A_bt ?c - context)
        (is_true_A_t ?c - context)
        (is_true_B_t ?c - context)
        (is_true_C_bt ?c - context)
        (is_true_E_gps ?c - context)
        (is_true_C_gps ?c - context)
    )
    (:functions (state ) (priority ))

    (:action rule_ActivateOutdoor
        :parameters (?c - context)
        :precondition (and
            (= (state) 0)
            (= (priority) 5)
            (is_true_A_gps ?c)
            (not (is_true_B_gps ?c))
            (not (is_true_C_gps ?c))
        )
        :effect (and
            (assign (state) 1)
            (assign (priority) 0)
        )
    )
    (:action rule_DeactivateOutdoor
        :parameters (?c - context)
        :precondition (and
            (= (state) 1)
            (= (priority) 5)
            (not
                (and
                    (is_true_A_gps ?c)
                    (not (is_true_B_gps ?c))
                    (not (is_true_C_gps ?c))
                )
            )
        )
        :effect (and
            (assign (state) 0)
            (assign (priority) 0)
        )
    )
; Omitted actions...
    (:action satisfy_D_gps
        :parameters (?c - context)
        :precondition (and
            (= (priority) 10)
            (is_true_A_gps ?c)
            (not (is_true_E_gps ?c))
            (not (is_true_D_gps ?c))
        )
        :effect (and
            (is_true_D_gps ?c)
            (assign (priority) 0)
        )
```

```
 66      )
 67      (:action unsatisfy_D_gps
 68          :parameters (?c - context)
 69          :precondition (and
 70              (= (priority) 10)
 71              (is_true_D_gps ?c)
 72          )
 73          :effect (and
 74              (not (is_true_E_gps ?c))
 75              (not (is_true_D_gps ?c))
 76              (assign (priority) 0)
 77          )
 78
 79      )
 80  ; Omitted actions...
 81      (:action set_priority_1
 82          :parameters (?c - context)
 83          :precondition (and
 84              (= (priority) 0)
 85              (or
 86                  (= (state) 0)
 87                  (= (state) 1)
 88                  (= (state) 2)
 89                  (and
 90                      (= (state) 3)
 91                      (not
 92                          (and
 93                              (is_true_A_gps ?c)
 94                              (is_true_E_gps ?c)
 95                          )
 96                      )
 97                  )
 98                  (and
 99                      (= (state) 4)
100                      (is_true_A_gps ?c)
101                      (is_true_E_gps ?c)
102                  )
103                  (= (state) 5)
104                  (= (state) 6)
105                  (= (state) 7)
106                  (= (state) 8)
107              )
108          )
109          :effect (assign (priority) 1)
110      )
111  ; Omitted actions...
112  )
```

Listing 7.2 contains part of the PDDL domain extracted automatically from the A-FSM model of PhoneAdapter. The whole domain is composed by 905 lines of PDDL code of which here we report only some fragments. The comments in Line 53, Line 80 and Line 111 indicate where we cut the generated document.

The concepts of state and priority have been represented in Line 19 with two fluents *state* and *priority*. These fluents are reassigned by adaptation actions and are read by trigger predicates. Alternatively, instead of using fluents it would have been possible to encode both state and priority as variables, to read their value with predicates and to set them with actions. We adopted the fluent notation because it is more compact and because both planners we used in our experiments support fluent calculus.

In Line 2 the context is represented with a single variable "context". This variables encodes the current assignment of all PCVs. The truthfulness of a single PCV can be checked by using a variable-specific predicate. In Lines 5-18 we have defined a PDDL predicate for each PCV in the model.

Starting from the initial state with priority 0, the planner tries to apply an adaptation. If no adaptation can be applied in the current state with the current priority level then the planner can trigger a PDDL action to increase the priority. Lines 81-110 show an example of such actions. The planner loops on all the priority levels until it reaches priority 10. If an adaptation rule is applied, the state changes to the target state of the adaptation, the context changes according to the rule, the priority is reset to 0, and the planner restarts looping over the priority levels in the new state with the new context. The priority level 10 represents a level of priority lower than the levels 0 to 9 with which the adaptation rules have been defined. The planner uses priority level 10 to apply changes in context. Contextual changes have been modeled at priority level 10 with the assumption that rule predicates react more quickly than sensed changes in the context. After a contextual change has been applied the priority is reset to 0 and the loop starts over with the new context. The assumption stating that the application has enough time to check all the rule predicates in between contextual changes is normally satisfied due to the refresh rate of sensors which is longer than the time that the application needs to check the predicates. Software developers in need of validating contextual changes that are faster than the adaptation logic can prioritize such changes by increasing their priority level.

The planner can apply an adaptation by means of PDDL actions. Lines 21-52 show two such actions, each one representing a single adaptation rule. The planner can apply such actions if the state and the priority match with the ones for which the adaptation rule's trigger predicate is satisfied. The trigger predicate is encoded as a composition of PDDL predicates. Lines 23-29 show the precondition for the action "rule_ActivateOutdoor". It can only be applied if the state is 0, the priority is 5, and if the trigger predicate is satisfied.

Lines 54-77 show an example of two contextual changes that set and unset the GPS location. In PhoneAdapter changes in the GPS location affects various PCVs whose values are restricted by certain global constraints. These global constrains are encoded as PDDL preconditions and affect the rule actions modeling contextual changes.

7.2 Detecting Nondeterministic Activations

As explained in Section 5.2.1, nondeterministic adaptation faults happen as a violation of the determinism property when two rules are satisfied for the same time and their trigger predicates are satisfied by the same set of PCV values.

7.2.1 Detecting Nondeterministic Activations with the Enumerative Approach

Algorithm 4 detects nondeterministic adaptation faults by using the enumerative approach.

Those faults can be easily spotted by observing the state matrix. For instance in the fragment of

Algorithm 4 Nondeterministic Activation Detection (Enumerative)

Input: AFSM M: an A-FSM.

Output: Set faults: set of detected faults.

```
1: for each State $S$ in $M$ do
2:     StateMatrix stateMatrix = $S$.getStateMatrix()
3:     for each BitString bitString $\in$ stateMatrix do
4:         rules = $S$.getSatisfiedRules(bitString)
5:         if rules.count() $> 1$ then
6:             faults $=$ faults $+ \{S$, rules, bitString$\}$
7:         end if
8:     end for
9: end for
```

state matrix listed in Listing 6.1 there are no nondeterministic activations because none of the PCVs assignments triggers more than one adaptation rule. Contrarily two of the four example PCVs assignments reported in Listing 6.2 contains a nondeterministic activation. Intuitively the state matrix allow to detect this pattern of fault simply by iterating over all its entries and by reporting those ones with more than one satisfied adaptation rule. This is done by Algorithm 4.

More in details Algorithm 4 identifies the existence of bit strings that satisfy the predicates of multiple rules which would produce a nondeterministic activation. The algorithm does not have to take in account the priorities since that has already been computed during the composition of the state matrix. Note that nondeterministic activations could also be detected when the state matrix is derived from the A-FSM by requiring that each input satisfies the predicate of at most one rule. However, since the state matrices are used by several algorithms, we decided to separate their generation from their analysis.

In Line 2 of the algorithm, S.getStateMatrix() returns the local state matrix for state S, which contains a list of pairs (*bit string*, *satisfied rules*) for S. In Line 4, S.getSatisfiedRules(bitString) the algorithm fetches from the state matrix all the highest priority satisfied rules for the current bit string. Finally, in Lines 5–7, if there is more than one such highest-priority rule, then the affected rules and state are reported along with the current bit string, which can be used to diagnose and eliminate the discovered fault.

The enumerative Algorithm 4 returns a list of all the bit strings and corresponding rules that can lead to a nondeterministic adaptation fault. Listing 7.3 is a short fragment of the generated report. The bit-strings reported as faulty (e.g. $101 * *0100 * **$) encode the PCVs of phone adapter in the order in which they where defined in the A-FSM model which is: $A_{gps}, B_{gps}, C_{gps}, D_{gps}, E_{gps}, A_{bt}, B_{bt}, C_{bt}, D_{bt}, E_{bt}, A_t, B_t$. For clarity the bit-string $101 * *0100 * **$ indicates that a nondeterministic adaptation occurs when A_{gps}, C_{gps} and B_{bt} are satisfied and when B_{gps}, A_{bt}, C_{bt} and D_{bt} are not. The assignment of all the remaining PCVs, i.e. the ones indicated with ”*” do not contribute to the occurrence of this fault. The same notation will be used in all the error reports of the enumerative approach.

Listing 7.3: Fragment of output of Algorithm 4

```
===========================================
Nondeterministic Adaptations [Enumerative]:
===========================================
============= State: General =============
101**0100*** [ActivateOffice, ActivateHome]
100**0010*** [ActivateOffice, ActivateOutdoor]
100**0110*** [ActivateOffice, ActivateHome,
              ActivateOutdoor]
[...]
```

Algorithm 4 has worst-case complexity $O(|\mathcal{S}| * 2^{|\mathcal{V}|})$, since for each state it potentially analyzes all possible assignments of values to the propositional context variables.

The remaining enumerative algorithms assume that the set of rules is deterministic; therefore, nondeterministic activation faults must be eliminated before applying the other algorithms.

7.2.2 Detecting Nondeterministic Activations with the Globally Symbolic Approach

For each state and for each rule in the global activation BDD there is a nondeterministic activation for each boolean configuration in which the sub-tree encoding the rule's activation overlaps with the activation sub-BDD of any other rule. This is possible because the global activation BDD already contains for each rule in each state an activation sub-BDD in which those configurations violating a constraint or covered by rules at higher priority have already been removed.

Algorithm 5 implements the detection of nondeterministic activations faults using a globally symbolic approach. However, the creation of the activation sub-BDD and the faults report enumerates on states and rules.

This algorithm starts from the global activation BDD (Line 2) that already encodes overlapping adaptations and filters unnecessary information and reports faults in a comprehensible form. Interfering rules normally have different destination states, which are also encoded in the global activation BDD. It is necessary to collapse all the sub-graphs representing activations from the same PCV assignment independently from their destination. In Line 3 we use the existence quantification over predicate to remove the boolean variables representing the destination states and the rules action PCVs thereby allowing the analysis to ignore destination states and actions. The resulting BDD contains only the variables encoding states, rules and trigger inputs.

In the rest of the algorithm we iterate state by state and rule by rule reporting faults. This algorithm isolates satisfying inputs for all states (Line 5), detects which rules are satisfied by those (Lines 7–19), and if there is more than one reports a fault (Lines 15–18). In Lines 5–6 the algorithm isolates activations of the current state and then remove state variables from the BDD. We then isolate overlaps between a rule and all the others by iterating on all the combinations of that rule with the others. In Lines 8–9 we restrict the activation of the current state only to the activation of a certain rule and we remove rule variables. We obtain a BDD containing exclusively the trigger input for the current rule in the current state. In Line 10 we compute the conjunction between the state activation and the inputs

Algorithm 5 Nondeterministic Activation Detection (Globally Symbolic)

Input: AFSM *M*: an A-FSM encoded using OBDDs.

Output: Set faults: set of detected faults.

```
1: faults = $\{\}$
2: BDD gActivation = $M$.getGlobalActivation()
3: gActivation = gActivation.exist($M$.getDestStateVars(), $M$.getActionVars())
   $\{$gActivation now contains states rules and the PCV assignments$\}$
4: **for** each StateBDD $S \in M$ **do**
5:    BDD activationInState = gActivation $\wedge$ $S$
6:    activationInState = activationInState.exist($M$.getStateVars())
      $\{$activationInState contains rules and the PCV assignments for all the activations in state $S\}$
7:    **for** each RuleBDD $R \in S$.getActiveRules() **do**
8:       BDD ruleActivation $=$ activationInState $\wedge$ $R$
9:       ruleActivation $=$ ruleActivation.exist($M$.getRuleVars())
         $\{$ruleActivation contains PCV assignments$\}$
10:      BDD activationInStateForRule = activationInState $\wedge$ ruleActivation
         $\{$activationInStateForRule characterizes all the inputs triggering $R$ in $S\}$
11:      Set faultyRules $= \{\}$
12:      BDD faultyInput $= 0$
13:      **for** each RuleBDD $R1 \in S$.getActiveRules() - $\{R\}$ **do**
14:         BDD overlap = activationInStateForRule $\wedge$ $R1$
            $\{$overlap contains PCV assignments satisfying both $R$ and $R1$ in $S\}$
15:         **if** overlap != 0 **then**
16:            faultyInput = faultyInput $\vee$ overlap
17:            faultyRule $=$ faultyRule $+$ $R1$
18:         **end if**
19:      **end for**
20:      **if** ( faultyRules.size() $>$ 1) **then**
21:         faults $=$ faults $+$ $\{S$, faultyRules, faultyInput$\}$;
22:      **end if**
23:   **end for**
24: **end for**
```

triggering the current rule. By looping on all the other rules in Lines 13–19 we can extract overlaps between the current rule and any other. Note that this inner iteration is only required to report explicitly which rules are interfering. The overlap of inputs could be computed by simply isolating which inputs in $ruleActivation$ are also satisfying any other rule. That can be done by replacing Lines 10–19 with $BDDoverlap = activationInState \wedge ruleActivation \wedge \neg R.getEncoding()$, and then reporting any not empty overlaps.

The complexity of this algorithm is $O(|\mathcal{S}| * |\mathcal{R}|^3)$.[1]

The symbolic Algorithm 5 returns an OBDD describing the fault. An OBDD is returned for each rule involved in a nondeterministic activation faults, along with an indication of the interfering rules. Such OBDDs encode both inputs and rules because the nondeterminism exists among a set of rules. Listing 7.4 is a sample fault report as it is generated by our tool. In this and in all the other error reports generated by the symbolic algorithms all the variables are encoded as illustrated in Table 6.1. For instance 0 : 0 means that A_{gps} if not satisfied.

Listing 7.4: Output of Algorithm 5

```
Nondeterministic Adaptations Globally Symbolic: 20ms.

Found 5 faults:

NondeterministicAdaptation [
   state = 'General'
   rules = [ActivateOutdoor -> ActivateHome, ActivateOffice]
   BDD = <0:0, 3:0, 4:1, 6:0, 9:1, 11:0, 32:0, 33:1, 34:1, 35:0>
         <0:1, 3:0, 4:1, 6:0, 9:0, 11:0, 32:0, 33:0, 34:1, 35:0>
         <0:1, 3:0, 4:1, 6:0, 9:1, 11:0, 32:0, 34:1, 35:0>
]

NondeterministicAdaptation [
   state = 'General'
   rules = [ActivateHome -> ActivateOutdoor, ActivateOffice]
   BDD = <0:0, 3:1, 4:1, 5:0, 6:0, 9:1, 32:0, 33:1, 34:1, 35:0>
         <0:0, 3:1, 4:1, 5:1, 6:0, 9:0, 11:1, 32:0, 33:1, 34:1, 35:0>
         <0:0, 3:1, 4:1, 5:1, 6:0, 9:1, 32:0, 33:1, 34:1, 35:0>
         <0:1, 3:0, 4:0, 6:0, 9:1, 32:0, 33:1, 34:1, 35:0>
         <0:1, 3:0, 4:1, 5:0, 6:0, 9:0, 11:0, 32:0, 33:0, 34:0, 35:0>
         <0:1, 3:0, 4:1, 5:0, 6:0, 9:1, 11:0, 32:0, 33:0, 34:0, 35:0>
         <0:1, 3:0, 4:1, 5:0, 6:0, 9:1, 11:0, 32:0, 33:1, 34:1, 35:0>
         <0:1, 3:0, 4:1, 5:0, 6:0, 9:1, 11:1, 32:0, 33:1, 34:1, 35:0>
         <0:1, 3:0, 4:1, 5:1, 6:0, 9:0, 11:0, 32:0, 33:0, 34:0, 35:0>
         <0:1, 3:0, 4:1, 5:1, 6:0, 9:0, 11:1, 32:0, 33:1, 34:1, 35:0>
         <0:1, 3:0, 4:1, 5:1, 6:0, 9:1, 11:0, 32:0, 33:0, 34:0, 35:0>
         <0:1, 3:0, 4:1, 5:1, 6:0, 9:1, 11:0, 32:0, 33:1, 34:1, 35:0>
         <0:1, 3:0, 4:1, 5:1, 6:0, 9:1, 11:1, 32:0, 33:1, 34:1, 35:0>
         <0:1, 3:1, 4:0, 6:0, 9:1, 32:0, 33:1, 34:1, 35:0>
         <0:1, 3:1, 4:1, 5:0, 6:0, 9:1, 32:0, 33:1, 34:1, 35:0>
         <0:1, 3:1, 4:1, 5:1, 6:0, 9:0, 11:1, 32:0, 33:1, 34:1, 35:0>
         <0:1, 3:1, 4:1, 5:1, 6:0, 9:1, 32:0, 33:1, 34:1, 35:0>
]
```

[1]The complexity of the exist() operation in Line 3 of the algorithm is exponential in the number of variables quantified and thus linear in the number of rules.

```
NondeterministicAdaptation [
    state = 'General'
    rules = [ActivateOffice -> ActivateOutdoor, ActivateHome]
    BDD = <0:0, 3:0, 4:1, 6:0, 9:1, 11:0, 32:0, 33:0, 34:0, 35:0>
          <0:0, 3:1, 4:1, 5:0, 6:0, 9:1, 32:0, 33:0, 34:1, 35:0>
          <0:0, 3:1, 4:1, 5:1, 6:0, 9:0, 11:1, 32:0, 33:0, 34:1, 35:0>
          <0:0, 3:1, 4:1, 5:1, 6:0, 9:1, 32:0, 33:0, 34:1, 35:0>
          <0:1, 3:0, 4:0, 6:0, 9:1, 32:0, 33:0, 34:1, 35:0>
          <0:1, 3:0, 4:1, 5:0, 6:0, 9:1, 11:0, 32:0, 33:0, 35:0>
          <0:1, 3:0, 4:1, 5:0, 6:0, 9:1, 11:1, 32:0, 33:0, 34:1, 35:0>
          <0:1, 3:0, 4:1, 5:1, 6:0, 9:0, 11:1, 32:0, 33:0, 34:1, 35:0>
          <0:1, 3:0, 4:1, 5:1, 6:0, 9:1, 11:0, 32:0, 33:0, 35:0>
          <0:1, 3:0, 4:1, 5:1, 6:0, 9:1, 11:1, 32:0, 33:0, 34:1, 35:0>
          <0:1, 3:1, 4:0, 6:0, 9:1, 32:0, 33:0, 34:1, 35:0>
          <0:1, 3:1, 4:1, 5:0, 6:0, 9:1, 32:0, 33:0, 34:1, 35:0>
          <0:1, 3:1, 4:1, 5:1, 6:0, 9:0, 11:1, 32:0, 33:0, 34:1, 35:0>
          <0:1, 3:1, 4:1, 5:1, 6:0, 9:1, 32:0, 33:0, 34:1, 35:0>
]

NondeterministicAdaptation [
    state = 'Outdoor'
    rules = [DeactivateOutdoor -> ActivateJogging]
    BDD = <2:1, 3:0, 4:1, 6:0, 11:1, 32:0, 33:1, 34:0, 35:0>
          <2:1, 3:1, 4:1, 6:0, 32:0, 33:1, 34:0, 35:0>
]

NondeterministicAdaptation [
    state = 'Outdoor'
    rules = [ActivateJogging -> DeactivateOutdoor]
    BDD = <2:1, 3:0, 4:1, 6:0, 11:1, 32:1, 33:0, 34:0, 35:0>
          <2:1, 3:1, 4:1, 6:0, 32:1, 33:0, 34:0, 35:0>
]
```

Algorithm 5 decodes nondeterministic overlaps state by state and rule by rule, therefore each overlap is reported once per rule. For instance in state General there are three rules interfering and the fault is reported three times. The OBDD encoding the faults contains the PCVs assignment and the rule with which the current rule is overlapping. Consider for instance the nondeterministic activation in state General between rule ActivateOutdoor with ActivateHome and ActivateOffice. The reported OBDD states that ActivateOutdoor overlaps with ActivateOffice (encoded as $< 32 : 0, 33 : 1, 34 : 1, 35 : 0 >$) when the PCVs are assigned with $< 0 : 0, 3 : 0, 4 : 1, 6 : 0, 9 : 1, 11 : 0 >$, with ActivateHome (encoded as $< 32 : 0, 33 : 0, 34 : 1, 35 : 0 >$) when the PCVs are assigned with $< 0 : 1, 3 : 0, 4 : 1, 6 : 0, 9 : 0, 11 : 0 >$ and with both ($< 32 : 0, 34 : 1, 35 : 0 >$) when the PCVs are assigned with $< 0 : 1, 3 : 0, 4 : 1, 6 : 0, 9 : 1, 11 : 0 >$. Also note that the OBDD reported for the faults for rules ActivateOffice and ActivateHome are different than this one because different are the overlaps that they identify.

7.2.3 Detecting Nondeterministic Activations with the Locally Symbolic Approach

For each state Algorithm 6 explores all pairs of rules at the same priority level (Line 4,5,7–9)

Algorithm 6 Nondeterministic Activation Detection (Locally Symbolic)

Input: AFSM *M*: an A-FSM encoded using OBDDs.

Output: Set faults: set of detected faults.

```
1: faults = $\{\}$
2: for each $S \in M$ do
3:     Set unexploredRules = $S$.getActiveRules()
4:     for RuleBDD $R1 \in S$.getActiveRules() do
5:         unexploredRules = unexploredRules - $\{R1\}$
6:         for RuleBDD $R2 \in$ exploredRules do
7:             if $R1$.getPriority() != $R2$.getPriority() then
8:                 continue;
                   $\{$Skips different priorities$\}$
9:             end if
10:            BDD overlap = $R1$.getPredicate() $\wedge$ $R2$.getPredicate()
               $\{$Only overlaps belonging to the activation BDD are faults$\}$
11:            BDD fault = overlap $\wedge$ $S$.getActivation() $\wedge$ $R1$.getEncoding()
12:            if fault != 0 then
13:                faults $=$ faults $+$ $\{(S, R1, R2,$ fault$)\}$
14:            end if
15:        end for
16:    end for
17: end for
```

and generates a fault report if their predicates can be satisfied at the same time (Line 13). This algorithm enumerates on states and on distinct pair of rules and on Line 10 uses the symbolic representation of their activation to detect interferences. Note that not all the overlaps between two rule predicates generate a fault. Only those ones which happen with the higher priority level are reachable and, when triggered, generate a fault. In Line 11 all the overlaps which are not part of the activation tree are pruned.

Note that if the predicates of three or more rules are satisfied then the algorithm generates a solution for each pair. The complexity of this algorithm is $O(|\mathcal{S}| * |\mathcal{R}|^2)$.

The locally symbolic Algorithm 6 does not return a single group of interfering rules, but instead detects pairs of such rules. Faults involving three or more rules are reported as multiple faults involving different pairs of rules, reporting more faults but with a more compact OBDDs. Listing 7.5, if compared with Listing 7.4 , shows that the same fault is reported with a clearer and more compact notation. As for the globally symbolic algorithm the error report use the variable codification illustrated in Table 6.1.

Listing 7.5: Output of Algorithm 6

```
Nondeterministic Adaptations Locally Symbolic: 0ms

Found 4 faults:

NondeterministicAdaptation [
    state = 'General'
    rules = [ActivateOutdoor, ActivateHome]
    BDD = <0:1, 3:0, 4:1, 11:0>
]

NondeterministicAdaptation [
    state = 'General'
    rules = [ActivateOutdoor, ActivateOffice]
    BDD = <3:0, 4:1, 9:1, 11:0>
]

NondeterministicAdaptation [
    state = 'General'
    rules = [ActivateHome, ActivateOffice]
    BDD = <0:0, 3:1, 4:1, 5:0, 9:1>
          <0:0, 3:1, 4:1, 5:1, 9:0, 11:1>
          <0:0, 3:1, 4:1, 5:1, 9:1>
          <0:1, 4:0, 9:1>
          <0:1, 4:1, 5:0, 9:1>
          <0:1, 4:1, 5:1, 9:0, 11:1>
          <0:1, 4:1, 5:1, 9:1>
]

NondeterministicAdaptation [
     state = 'Outdoor'
     rules = [DeactivateOutdoor, ActivateJogging]
     BDD = <2:1, 3:0, 4:1, 11:1>
           <2:1, 3:1, 4:1>
]
```

For the same detected faults in *PhoneAdapter*, Algorithm 6 generates four OBDDs with an average of 5.25 internal nodes, while Algorithm 5 reports five OBDDs with an average of 20.2 internal nodes.

Thus, the latter are roughly four times as large as the former. From the OBDD reported by the Locally symbolic Algorithm it can be easily inferred that the pairs [ActivateOutdoor, ActivateHome] and [ActivateOutdoor, ActivateOffice] have a single overlap while the pair [ActivateHome, ActivateOffice] has 7 different ones (because the predicate representing the reported OBDD is the disjunction of 7 conjunctions). In the error report of the globally symbolic algorithm this information is not as explicit as it is here.

Developers interested in understanding which rules are interfering can benefit from the output produced by the symbolic Algorithm 5 because it groups interfering rules. The enumerative Algorithm 4 shows faulty inputs more clearly but it does not group them by rule. Algorithm 6 lies somewhat between these extremes by organizing faults by pair of rules.

7.2.4 Detecting Nondeterministic Activations with Planners

Nondeterministic activation faults can be detected by instructing the planner to search for solutions in which, from a certain state two rules with the same priority can be triggered. To do so a set of goal is extracted from the A-FSM, one for each couple of distinct rule with the same priority and in the same state.

Listing 7.6 depicts the extracted goal to detect the nondeterministic fault between *ActivateHome* and *ActivateOffice* in state *General*. Simply the goal is the conjunction between the fluent representing the state (Line 9), the fluent representing the priority (Line 10), and the predicates of the two rules (Lines 11-17 and Lines 18-25) Intuitively if $state = 0$ and $priority = 5$ which is the priority level of *ActivateHome* and *ActivateOffice* then if it is possible to reach a configuration in which both their trigger predicate are satisfied then the planner has found a plan to a nondeterministic activation. Note that the planner only reports the first plan found.

Listing 7.6: Extracted Goal for PhoneAdapter

```
1  (define (problem Nondeterministic_ActivateHome_ActivateOffice)
2    (:domain PhoneAdapter)
3    (:requirements :strips :typing :equality
4      :disjunctive-preconditions :fluents)
5    (:objects c - context)
6    (:init (= (state) 0)(= (priority) 0))
7    (:goal
8      (and
9        (= (state) 0)
10       (= (priority) 5)
11       (not (and
12         (not (is_true_B_bt c))
13           (not
14             (and
15               (is_true_A_gps c)
16               (is_true_B_gps c)
17             ))))
18       (not (and
19         (not (is_true_C_bt c))
20           (not
21             (and
22               (is_true_D_bt c)
```

```
                (is_true_A_gps c)
                (is_true_C_gps c)
              )))))
    ))
```

If we run the goal in Listing 7.6, the planner MIPS-XXL returns the plan in Listing 7.7 containing three environmental changes and two adaptations. Recall that in this context we are trying to reach a state in which both rules are active, and the planner produces a path from the initial state to achieve this goal:

Listing 7.7: Output of Algorithm 7.6

```
Nondeterministic Adaptations
[ActivateOffice , ActivateHome]:
==========================================
0: (SET−PRIORITY−1 C ) [1]
...
9: (SET−PRIORITY−10 C ) [1]
10: (SATISFY−A−GPS C ) [1] // Gps valid
11: (SET−PRIORITY−1 C ) [1]
...
15: (SET−PRIORITY−5 C ) [1]
16: (RULE−ACTIVATEOUTDOOR C ) [1]
17: (SET−PRIORITY−1 C ) [1]
...
26: (SET−PRIORITY−10 C ) [1]
27: (SATISFY−C−BT C ) [1] // BT = {"OfficePC"}
28: (SET−PRIORITY−1 C ) [1]
...
37: (SET−PRIORITY−10 C ) [1]
38: (SATISFY−B−GPS C ) [1] // GPS = {"Home"}
39: (SET−PRIORITY−1 C ) [1]
...
43: (SET−PRIORITY−5 C ) [1]
44: (RULE−DEACTIVATEOUTDOOR C ) [1]
45: (SET−PRIORITY−1 C ) [1]
...
49: (SET−PRIORITY−5 C ) [1]
```

The plan reported above is read as follows: Starting from state *General* (the initial state), GPS is enabled (possibly by the user) and the application adapts into state *Outdoor* (possibly because the GPS is detecting an unknown location). From state *Outdoor*, the Bluetooth device detects the office laptop (maybe because the laptop has been switched on by the user), causing no adaptation. Then the GPS location becomes *Home*, and the application adapts back to *General*, thus returning to the initial state. After this chain of adaptations, due to the effect of the actions following each adaptation, in state *General* it becomes now possible to trigger both *ActivateOffice* and *ActivateHome*, causing a nondeterministic activation. Note that in the fault detected in this example the planner loops initially from state *General* to state *General* without pruning the loop. This happens because the planner considers the truthfulness of all the predicates which is changed due to the execution of the loop.

Table 7.1: Comparison between the Nondeterministic fault detection algorithms

Algorithm	Reported faults	Aggregates assignments	Detects multiple overlaps	Traceback	Duplicated reports
Enumerative	All	No	Yes	Local violation	for each couple
Globally symbolic	All	Yes	Yes	Local violation	No
Locally symbolic	All	Yes	No	Local violation	No
PDDL based	One per pair rules	Unnecessary	No	Trace from the initial state	No

7.2.5 Comparison

There is no superior implementation among these four algorithms. They all find all the faults and they perform similarly. They differ in the way in which the fault is reported. Table 7.1 summarizes their differences by showing which assignments are reported, how they are aggregated, if interferences involving more than two rules are reported and if a trace back is generated.

Once the PDDL-based algorithm has hit a fault it reports a whole trace from the initial state to the fault. This is very useful to trace faults, and to identify the cause of each failure. However, the planner stops as soon as a fault is found, so this algorithm only identifies a single faulty assignment and not the whole fault space.

In contrast the enumerative algorithm not only detects all the faulty assignments but also detects when there are more than two interfering rules. The drawback is that the enumerative algorithm reports each single faulty assignment as a distinguished fault. In addition, faults are only reported by identifying the faulty assignment and not a complete trace.

The OBDD-based approaches are superior in aggregating the faulty assignments and report a single BDD for each fault space. In particular the globally symbolic one reports multiple faulty rules together, while the locally symbolic one reports them pairwise. However neither of them reports a complete trace but only the faulty assignments. Notice that for each rule the symbolic algorithms report all the interferences with that specific rule. Since interferences are generated by two rules at time, faults are reported twice, one time for each rule.

7.3 Detecting Liveness Violations

Due to the variety of possible adaptations and to number of PCVs, increasing with the introduction of more complex applications, certain rules may not be satisfiable, and certain states may be unreachable. This phenomenon is analogous to dead code in the source code of a program. The four algorithms in this section detect such states and rules.

7.3.1 Detecting Liveness Violations with the Enumerative Approach

In terms of state matrices, a Dead Rule fault is indicated by the absence of bit strings that satisfy the predicate of some rule. If for a state S all its active rules are dead, then there exists a Dead State fault. Algorithm 7 checks, for each state, whether the predicates for all rules are satisfiable for at least one bit string. The algorithm iterates over all states and, for each state, executes two loops, the first one identifying the live rules, and the second one reporting any remaining dead rules. Like Algorithm 4, this algorithm considers each state independently of the others, and so Line 2 retrieves the state matrix for

Algorithm 7 Dead Predicate and Dead State Detection (Enumerative)

Input: AFSM M: an instance of A-FSM.

Output: Set faults: set of detected faults.

```
1:  for each State $S$ in $M$ do
2:    StateMatrix stateMatrix = $S$.getStateMatrix()
3:    Set untriggered = $S$.getActiveRules()
4:    for each BitString bitString $\in$ stateMatrix do
5:      if untriggered $==$ $\{\}$ then
6:        break
7:      end if
8:      Rule $R$ = $S$.getSatisfiedRules(bitString)
        $\{$Deterministic: $R$ is unique$\}$
9:      untriggered = untriggered $-$ $\{R\}$
10:   end for
11:   for each Rule $R \in$ untriggered do
12:     faults $=$ faults $+$ $\{S, R\}$
        $\{$Dead Rule$\}$
13:   end for
14:   if untriggered $==$ $S$.getActiveRules() then
15:     faults $=$ faults $+$ $\{S\}$
        $\{$Dead State$\}$
16:   end if
17: end for
```

Algorithm 8 Dead Rule and Dead State Detection (Globally Symbolic)

Input: AFSM M: an A-FSM encoded using OBDDs.

Output: Set deadStates, deadRules: sets of detected faults.

```
1:  deadStates = $\{\}$
2:  deadRules = $\{\}$
3:  BDD gActivation = $M$.getGlobalActivation();
4:  for each StateBDD $S \in M$ do
5:      BDD stateActivation = gActivation $\wedge$ $S$
6:      if stateActivation $==$ 0 then
7:          deadStates $=$ deadStates $+$ $\{S\}$
8:          for each RuleBDD $R \in S$.getActiveRules() do
9:              deadRules $=$ deadRules $+$ $\{(S, R)\}$
10:         end for
11:     else
12:         for each RuleBDD $R \in S$.getActiveRules() do
13:             BDD ruleActivation = stateActivation $\wedge$ $R$
14:             if ruleActivation $==$ 0 then
15:                 deadRules $=$ deadRules $+$ $\{(S, R)\}$
16:             end if
17:         end for
18:     end if
19: end for
```

state S. Line 3 initializes a set with all the rules active in the current state. Line 9 removes the associated rule of each bit string in the state matrix, and it assumes that there should be at most one such rule since, as mentioned in Section 5.2.1, the algorithm assumes that any Nondeterministic Activation faults have been eliminated. Line 12 reports any rules that are not satisfied after searching through the state matrix. If all the rules in a state are not satisfiable, then in Line 15 the current state is reported as being dead. Algorithm 7 explores potentially all the bit strings for the propositional context variables and thus has worst-case complexity $O(|\mathcal{S}| * (2^{|\mathcal{V}|} + |\mathcal{R}|))$.

7.3.2 Detecting Liveness Violations with the Globally Symbolic Approach

Algorithm 8 starts from the global activation BDD of the input A-FSM (Line 3), which encodes all the information about which rules can be triggered. The algorithm then iterates over states and rules (Line 4 and Line 12). If a state does not exist in the global activation BDD (Line 6), then the state and all its active rules are dead and are added to the fault set in Lines 8–10. Otherwise the decoding adds as a fault each single rule that is active in the state at the current iteration but does not appear in the global activation BDD (Lines 14–16). The complexity of this algorithm is $O(|\mathcal{S}| * |\mathcal{R}|)$.

Note that the fault extraction is linear, because we aim to log faults by state and by rule. It would

Algorithm 9 Dead Rule and Dead State Detection (Locally Symbolic)

Input: AFSM M: an A-FSM encoded using OBDDs.

Output: Set deadStates, deadRules: sets of detected faults.

```
1:  deadStates = {}
2:  deadRules = {}
3:  for each StateBDD $S \in M$ do
4:      BDD stateActivation = $S$.getActivation();
5:      if stateActivation $==$ 0 then
6:          deadStates $+= \{S\}$
7:          for each RuleBDD $R \in S$.getActiveRules() do
8:              deadRules $=$ deadRules $+ \{(S, R)\}$
9:          end for
10:     else
11:         for each RuleBDD $R \in S$.getActiveRules() do
12:             BDD ruleActivation = stateActivation $\wedge R$;
13:             if ruleActivation $==$ 0 then
14:                 deadRules $=$ deadRules $+ \{(S, R)\}$
15:             end if
16:         end for
17:     end if
18: end for
```

be possible to identify if there is at least one instance of a fault in the whole A-FSM by generating one OBDD containing all the rules for each state and by verifying that its conjunction with the global activation OBDD is not null. However the generation of such an OBDD is also linear in the number of states and rules.

7.3.3 Detecting Liveness Violations with the Locally Symbolic Approach

Algorithm 9 detects rule and state liveness violations with the locally symbolic approach. The algorithm loops from the state activation BDD of each state in Line 3, checking for dead states in Lines 4–9. If the state is live, then the algorithm decodes each faulty rule from the state activation BDD (Lines 11–16). If a rule has a null activation BDD then that rule cannot be triggered and it is reported as faulty. Like its fully symbolic counterpart, the complexity of this algorithm is $O(|\mathcal{S}| * |\mathcal{R}|)$ but over a smaller data structure.

7.3.4 Detecting Liveness Violations with Planners

Listing 7.8: Extracted Goal for PhoneAdapter

```
(define (problem RuleLiveness_ActivateSynch)
    (:domain PhoneAdapter)
    (:requirements :strips :typing :equality :disjunctive-preconditions :fluents)
    (:objects c - context)
    (:init (= (state) 0)
        (= (priority) 0))
    (:goal
        (and (= (state) 0) (= (priority) 9)
            (not (and
                (not (is_true_B_bt c))
                (not (is_true_C_bt c)))))
    ))
```

Planners are designed to prove the reachability of given goals where the goal is a desired configuration. Planners are successful in proving that a goal is satisfiable if at least one feasible path exists that achieves the goal. In our PDDL algorithms we use planners to detect configurations causing faults. If the planner is able to reach the faulty configuration then the fault is reachable and therefore its existence is proven. However, for the liveness property, faults arise if no path satisfying a certain predicate is found.

Accordingly, our implementation checks if all the rules are reachable, and reports as faulty the unreachable ones. The drawback of this implementation is that no counterexample is reported for the unreachable rules. Listing 7.8 shows the goal generated to validate the liveness of rule *Sync* from state *General* of PhoneAdapter. One such goal is generated for each adaptation rule. The goal is satisfied if from the source state and with the priority of the rule, the trigger predicate of the rule is satisfied.

States are dead if all its active rules are dead. Dead states are simply detected by validating their outgoing rules.

7.3.5 Comparison

Table 7.3 shows a comparison between the four algorithms. In terms of reported errors all the four algorithms behave almost identically. However, contrarily to the others, the PDDL-based one does not directly report dead states but requires one to aggregate results from rule liveness validations.

Even if in the A-FSM a certain adaptation rule can be satisfied, it can become unreachable at runtime if the A-FSM adapts prematurely. We call such phenomenon a static hazard [SRWE08a]. In Chapter 8 we show that the PDDL-based approach is the only one capable of detecting such faults.

7.4 Detecting Reachability Violations

The reachability property asserts that all states are reachable. To detect violations of this property, detection algorithms have to prove that no paths reaching a certain state exists.

7.4.1 Detecting Reachability Violations with the Enumerative Approach

Detecting which states are reachable and which are not requires the exploration of the state machine starting from the initial state and then iteratively applying all adaptations to states that have not been

Table 7.2: Comparison between the Liveness fault detection algorithms

Algorithm	Reports dead rules	Reports dead states	Detects faults caused by hazards
Enumerative	Yes	Yes	No
Globally symbolic	Yes	Yes	No
Locally symbolic	Yes	Yes	No
PDDL based	Yes	No	Yes

Algorithm 10 Unreachable State Detection (Enumerative)

Input: M: an A-FSM.

Output: faultsVector: vector of detected faults.

```
1:  next = $M$.getInitialState()
2:  toExplore = $\{\}$
3:  unreached = $M$.getStates()
4:  **while** next != $\{\}$ **do**
5:      **for** each $S \in$ next **do**
6:          unreached $-=$ $S$
7:          stateMatrix = $S$.getStateMatrix()
8:          **for** each bitString $\in$ stateMatrix **do**
9:              $R$ = $S$.getSatisfiedRules(bitString)
                $\{$Deterministic: $R$ is unique$\}$
10:             toExplore $+=$ $R$.getDestState() $\cap$ unreached
11:         **end for**
12:     **end for**
13:     next = toExplore
14:     toExplore = $\{\}$
15: **end while**
16: **for** each $S \in$ unreached **do**
17:     faultsVector $+=$ $S$
18: **end for**
```

reached earlier in the exploration.

Algorithm 10 starts with a collection of unexplored states containing all states (Line 2) and a collection of states to explore that contains only the initial state (Line 1). In each iteration in Line 5 we explore consecutively all states in the collection to explore by analyzing the state matrix and, in Line 13, by adding to the collection to explore any additional states contained in the state matrix under consideration that are still contained in the set of unreached states. Additionally, in Line 6 we remove the current state from the collection of unreached states.

Intuitively, in the first iteration we explore the initial state, in the second one all the states at distance one from the initial state, and in the i-th iteration all the states at distance $i - 1$ from the initial state. The total number of iterations is equal to the maximum length of the shortest path from the initial state to any other state. Since each state is explored at most once, the worst-case complexity of the algorithm is $O(|\mathcal{S}| * 2^{|\mathcal{V}|})$.

We can modify this algorithm to account for the possibility that Algorithm 7 does not report any Dead Rule faults. In such a situation, this algorithm simply can loop on all rules within each iteration and add the destination state of the rule to the collection of states to explore, which would reduce the complexity to $O(|\mathcal{S}| * |\mathcal{R}|)$.

Algorithm 11 Unreachable State Detection (Globally Symbolic)

Input: AFSM M: an A-FSM encoded using OBDDs.

Output: Set faultsVector: set of detected faults.

```
1: faultsVector = $\{\}$
2: BDD toExplore = $M$.getInitialState().getActivation()
3: BDD unexplored = 1
   $\{$All the states$\}$
4: BDD gActivation = $M$.getGlobalActivation();
5: while toExplore $! = 0$ do
6:     unexplored = unexplored $-$ toExplore
7:     toExplore = gActivation $\wedge$ toExplore
8:     toExplore = toExplore.exist($M$.getStateVars(), $M$.getRuleVars(), $M$.getPcvVars())
9:     toExplore = toExplore.swap($M$.getDestStateVars(), $M$.getStateVars())
10:    toExplore = toExplore $\wedge$ unexplored
11: end while
12: for each StateBDD $S \in M$ do
13:    BDD faultInstance = unexplored $\wedge$ $S$
       $\{$If a state exists then it is unreachable$\}$
14:    if faultInstance $! = 0$ then
15:        faults $=$ faults $+$ $\{S\}$
16:    end if
17: end for
```

7.4.2 Detecting Reachability Violations with the Globally Symbolic Approach

Algorithm 11 detects unreachable states using a symbolic approach. The base idea is that since the global activation BDD already encodes all the destination states the algorithm just have to verify that those will be connected to the initial state.

Starting from the initial state (Line 2), Algorithm 11 iterates over a set of reached states (Line 5) until all the reached states have been explored. At each iteration the algorithm symbolically creates from the BDD representing the set of states to explore a new BDD representing the set of states which are reachable and which have not yet been explored (Lines 7–10). The BDD representing the set of unexplored reachable states to visit in the next iterations is extracted from the global activation BDD by swapping the destination states variables with the state variables (Line 9) only for adaptations which are active in the states which are currently being explored (Line 7).

Once There are no reachable unexplored states the algorithm iterates over all the states and checks whether any state is in the set of unexplored states (Lines 13–17). This algorithm has a worst case complexity of $O(|\mathcal{S}| * (|\mathcal{S}| + |\mathcal{R}| + 2^{|\mathcal{V}|}) + |\mathcal{S}|) = O(|\mathcal{S}|^3)$.

7.4.3 Detecting Reachability Violations with the Locally Symbolic Approach

Algorithm 12 Unreachable State Detection (Locally Symbolic)

Input: M: an A-FSM encoded using OBDDs.

Output: Set unreached: set of detected faulty states.

```
1: unreached = $\{M$.getStates()$\}$
2: Set toExplore = $\{M$.getInitialState()$\}$
3: Set toExploreNext = $\{\}$
4: while toExplore $! = \{\}$ do
5:    unreached -= toExplore
6:    for each StateBDD $S \in$ toExplore do
7:       BDD activation = $S$.getActivation()
8:       for each StateBDD $K \in$ unreached do
9:          BDD reached = activation $\wedge$ $K$.getDestStateEncoding()
10:         if reached $! = 0$ then
11:            toExploreNext $=$ toExploreNext $+ K$
12:         end if
13:      end for
14:   end for
15:   toExplore = toExploreNext
16:   toExploreNext = $\{\}$
17: end while
```

Algorithm 12 detects unreachable states with a locally symbolic approach. The base idea is exactly the one implemented by Algorithm 11 but the main iteration is performed over a set of states and not over a BDD

Similarly to Algorithm 11, Algorithm 12 starts from a set of states to explore (Line 1) initialized with the initial state and in Lines 8–13 iteratively explores all the states reachable from the states explored in the current iteration. In contrast to Algorithm 11, this algorithm only uses the local activation BDDs to check which states are reached in each iteration (Line 9). The complexity of this Algorithm is $O(|\mathcal{S}|!)$. The worst case with complexity $|\mathcal{S}|!$ is reached when all the states are in a chain. When this happens the algorithm explores one ring of the chain at each iteration.

7.4.4 Detecting Reachability Violations with Planners

Listing 7.9: Extracted Goal for PhoneAdapter

```
(define (problem Reachability_Office)
    (:domain PhoneAdapter)
    (:requirements :strips :typing :equality :disjunctive-preconditions :fluents)
    (:objects c - context)
    (:init (= (state) 0) (= (priority) 0))
    (:goal (= (state) 6))
)
```

Listing 7.9 shows the goal generated to validate the reachability of state *Office* of PhoneAdapter. Similarly the algorithm in Section 7.3.4 to the goal is satisfied if the state is reached, and a fault is reported if the planner does not find any feasible plan.

7.4.5 Comparison

In terms of detected faults Enumerative, globally symbolic and locally symbolic algorithms are equivalent. These algorithms consider reached each destination state of each rule which can be satisfied from a previously reached state. However it is possible that a runtime a certain set of PCVs assignments will never be reachable due to runtime issues that we have named *Static Hazards*[SRWE08a]. If such phenomenon affects all the assignments triggering adaptations to a certain state that state, even if conceptually reachable may be, in practice, unreacheable. Enumerative, symbolic and hybrid algorithms report those states as a false positives. On the contrary, the PDDL-based algorithm, by simulating the runtime behavior, successfully report such states as unreachable. Table 7.3 summarizes the comparison.

7.5 Detecting State Invariant Violations.

Each adaptation applies a new behavior to the application. Sequences of adaptations, may sequentially modify the application's internal configuration and may induce unwanted behaviors.

Table 7.3: Comparison between the Reachability Fault Detection Algorithms

Algorithm	Reports unreachable states	Reports partitioned A-FSM	Suffers from Static Hazards
Enumerative	Yes	Yes	No
GLobally symbolic	Yes	Yes	No
Locally symbolic	Yes	Yes	No
PDDL based	Yes	Yes	Yes

To detect such misconfigurations we have introduced in each state the concept of *invariant*. Similarly to a contract in contract base testing, an invariant is a predicate which should be always respected as long as the application is in a certain state.

7.5.1 Invariant Detection via the Enumerative and OBDD-Based Approaches

Detecting if the current configuration is respecting the invariant of the current state simply requires us to test the invariant on the current PCV assignment. However, in order to verify if the invariant can be violated it would be necessary to:

- **define an exploration algorithm**: The A-FSM must be explored simulating sequences of context changes and adaptations until a fault is reached.
- **define an adequacy criterion**: The exploration algorithm by definition could continue looping in the state forever. To stop it, if no faults are found, it is necessary to define a criterion by which the exploration can be considered complete.

We decided to go no further in the enhancement of these fault detection algorithms because the efficiency in terms of fault detection completely depends on the search algorithm and on the adequacy criteria used. In the literature several search algorithms and adequacy criteria already exist. The implementation and the evaluation of such algorithms is out of the scope of this thesis.

7.5.2 Detecting State Invariant Violations with Planners

We initially started experimenting with planners specifically to detect invariant violations. Indeed, the computation required to detect the violation of an invariant is close to what planners normally do to reach a goal.

The fault is defined as a goal in which the invariant is violated, for instance with the application being stable in the state in which the invariant is defined but the invariant being not satisfied. Starting from the initial state the exploration algorithm applies changes to context and adaptations until a violation is found or until the adequacy criterion for the exploration is found. This search strategy is entirely delegated to the planner, since planners are specifically optimized for this kind of task.

Listing 7.10: Extracted Goal for PowerManager

```
(define (problem StateInvariant_BatteryFull)
        (:domain PowerManager)
        (:requirements :strips :typing :equality :disjunctive-preconditions :fluents)
        (:objects c - context)
        (:init (= (state) 0)(= (priority) 0))
        (:goal
                (and (= (state) 2)
                        (is_true_GpsEnabled c)
                        (is_true_BtEnabled c)
                        (is_true_WiFiEnabled c)
                ))
)
```

Listing 7.10 depicts a state invariant validation goal extracted from PowerManager. The goal verifies that in state *BatteryFull* GPS, Bluetooth and WiFi are never active simultaneously, in order to prevent the battery from being drained too quickly.

If we run the planner with the goal in Listing 7.10 we obtain the plan in Listing 7.11. A detailed description of this plan can be found on Section 8.3. From the algorithm point of view what the planner did was first to find adaptations satisfying the goal conditions on the context then to adapt on the desired state.

Listing 7.11: Plan generate by the goal in Listing 7.10

```
State invariant violation [OnBattery]:
==========================================
0: (SET-PRIORITY-1 C ) [1]
...
10: (SATISFY-ONACCHARGE C ) [1] // Charging
11: (SET-PRIORITY-1 C ) [1]
...
15: (SET-PRIORITY-5 C ) [1]
16: (RULE-ACTIVATECHARGINGAC C ) [1]
17: (SET-PRIORITY-1 C ) [1]
...
26: (SET-PRIORITY-10 C ) [1]
27: (SATISFY-BATTERYFULL C ) [1] // the battery is full
28: (SET-PRIORITY-1 C ) [1]
...
37: (SET-PRIORITY-10 C ) [1]
38: (UNSATISFY-ONACCHARGE C ) [1] // not charging
39: (SET-PRIORITY-1 C ) [1]
...
43: (SET-PRIORITY-5 C ) [1]
44: (RULE-ACTIVATEONBATTERY C ) [1]
```

7.6 Detecting Stability Violations

7.6.1 Detecting Stability Violations with the Enumerative Approach

If an A-FSM is deterministic, it is possible to search for adaptation races and cycles by looking for *paths* of transitions among multiple states whose active rules contain predicates that are satisfied by the same bit string. Thus, in order to detect these faults, it is necessary to consider all propositional context variables, not just the subset relevant to the active rules of a single state.

Algorithm 13 checks, for each state S, whether a particular bit string in the state matrix of S can trigger a path of at least two transitions out of the state. Note that this algorithm is not local to a single state, in the sense that it iterates over paths through multiple states. Therefore the algorithm must consider the propositional context variables relevant to *all* the states in the A-FSM, and thus it constructs a state matrix over all such variables (Line 2). Line 3 selects the next bit string to be searched. In Lines 4–5, the variable *rvector* is set up to store the affected rules of any detected adaptation race or adaptation cycle, while *svector* is set up to store the affected states. In Line 6, the variable *isCycle* is used to differentiate between adaptation races and Adaptation Cycles, and it is also used to force the algorithm to terminate. Lines 7–8 find the destination state for the highest-priority rule of the current bit string and store it in variable *destState*. Lines 10–15 set *isCycle* true because, after at least one iteration of the innermost enclosing loop, at least one repeated state has been detected at that point. Line 18 looks

Algorithm 13 Adaptation Race and Cycle Detection (Enumerative)

Input: M: an instance of A-FSM.

Output: faultsVector: vector of detected faults.

```
1:  for each state $S$ in M do
2:    stateMatrix = $S$.getStateMatrix().toGlobal()
3:    for each bitString $\in$ stateMatrix do
4:      rvector = $\{\}$ $\{$explored rules$\}$
5:      svector = $\{\}$ $\{$reached states$\}$
6:      isCycle = false
7:      $R$ = $S$.getSatisfiedRules(bitString)
        $\{$Deterministic: $R$ is unique$\}$
8:      destState = $R$.getDestState()
9:      while destState != null || !destState $\in$ svector do
10:       if destState $\in$ svector then
11:         isCycle = true
12:         rvector $+=$ $R$
13:         svector $+=$ destState
14:         break
15:       end if
16:       rvector $+=$ $R$
17:       svector $+=$ destState
18:       $R_1$ = destState.getSatisfiedRules(bitString)
          $\{$Deterministic: $R_1$ is unique$\}$
19:       destState = $R_1$.getDestState()
20:     end while
21:     if size(svector)$>2$ then
22:       if isCycle then
23:         faultsVector $+=$ $\{S$, rvector, “cycle”, bitString$\}$
24:       else
25:         faultsVector $+=$ $\{S$, rvector, “race”, bitString$\}$
26:       end if
27:     end if
28:   end for
29: end for
```

for highest-priority rules whose source state is destState and whose predicate is satisfied on the *same* bit string under consideration, thus indicating the presence of an adaptation race or cycle. Line 19 updates the bit string with the appropriate corresponding actions. If a sequence of two or more states is detected after searching the bit strings for all active rules of the current state, then Lines 22–28 report the rules that form adaptation races and adaptation cycles along with the bit strings that cause them.

The worst-case complexity of Algorithm 13 is $O(|\mathcal{S}|^2 * 2^{|\mathcal{V}|})$, since it explores potentially all bit strings for paths containing potentially all states of the A-FSM.

If we run Algorithm 13 on PhoneAdapter we obtain the results in Listing 7.12. Note that Listing 7.12 only contains a fraction of the reported errors. In the error report with the symbol $->$ metastable adaptations occurring after the first one. In the example consider the bit-string $101{*}{*}{*}000111$. As for the previous examples that bit-string represent the assignment of PhoneAdapter PCvs in the order of their definition:$A_{gps}, B_{gps}, C_{gps}, D_{gps}, E_{gps}, A_{bt}, B_{bt}, C_{bt}, D_{bt}, E_{bt}, A_t, B_t$ Once that configuration is applied it triggers a sequence of adaptations first from state Home to state General by triggering the rule DeactivateHome, then to states Office and then Meeting. Once in state Meeting the sequence of adaptations starts looping between Office and Meeting.

Listing 7.12: Fragment of output of Algorithm 13

```
============================================
Adaptation Races and Cycles [Enumerative]:
============================================
============= State = Home =============

000***01011* Race  (Home, DeactivateHome)
                   ->(General , ActivateOffice)
                   ->(Office , ActivateMeeting)
                   ->Meeting

101**1001*** Race  (Home, DeactivateHome)
                   ->(General , ActivateOffice)
                   ->(Office , ActivateDriving)
                   ->Driving

101***010*** Race  (Home, DeactivateHome)
                   ->(General , ActivateOffice)
                   ->Office

101*11010*** Race  (Home, DeactivateHome)
                   ->(General , ActivateOffice)
                   ->(Office , ActivateDriving)
                   ->(Driving , ActivateDrivingFast)
                   ->DrivingFast

101***000111 Cycle (Home, DeactivateHome)
                   ->(General , ActivateOffice)
                   ->(Office , ActivateMeeting)
                   ->(Meeting , DeactivateMeeting)
                   ->Office
```

7.6.2 Detecting Stability Violations with the Globally Symbolic Approach

Algorithm 14 Adaptation Race and Cycle Detection (Globally Symbolic)

Input: AFSM M: an A-FSM encoded using OBDDs.

Output: Set faults: set of detected faults.

```
1:  faults = $\{\}$
2:  BDD gActivation = $M$.getGlobalActivation()
3:  BDD futureActivation = gActivation.exist($M$.getStateVars(), $M$.getRuleVars(), $M$.getPcvVars())
4:  futureActivation = futureActivation.swap($M$.getDestStateVars(), $M$.getStateVars())
5:  futureActivation = futureActivation.swap($M$.getActionVars(), $M$.getPcvVars())
6:  BDD fault = gActivation $\wedge$ futureActivation
7:  for each StateBDD $S \in M$ do
8:      for each RuleBDD rule $\in S$.getActiveRules() do
9:          StateBDD $D$ = $R$.getDestState()
10:         BDD stateRuleDestFilter = $S \wedge R \wedge D$
11:         BDD faultInstance = fault $\wedge$ stateRuleDestFilter
12:         if faultInstance $! = 0$ then
13:             faults $=$ faults $+ \{(S, R, D)\}$
14:         end if
15:     end for
16: end for
```

Algorithm 14 detects stability violations with a globally symbolic approach. The base idea is that there is to compare the global activation BDD with one of its transpositions in which all the adaptations have been applied and to isolate those configurations which are active both before and after the adaptation.

Starting from the global activation BDD (Line 2) Algorithm 14 creates a BDD representing the A-FSM after all the adaptations have been applied. This is done symbolically by removing all the variable representing the current state (Line 3), by swapping present and destination state variables (Line 4) and by swapping the action variables with input variables (Line 5).

The resulting BDD in Line 5 the disjunction of all the configuration applied after any possible adaptation was applied in conjunction with the state in which the application adapted. In Line 6 by computing the conjunction between the BDD in Line 5 and the global activation BDD we obtain a BDD containing for each state those inputs which violates the stability property.

The rest of the algorithm (Lines 7–16) decodes all the detected faults and organizes them by state and rule. The complexity of this algorithm is $O(|\mathcal{S}| + |\mathcal{R}| + (|\mathcal{S}| * |\mathcal{R}|))$.

Listing 7.13 is a fragment of the error report of Algorithm 14 showing one of the 15 faults that it detected. The reported OBDD says that in state General ($< 24 : 0, 25 : 0, 26 : 0, 27 : 0 >$) with the assignment $< 2 : 1, 3 : 0, 4 : 1, 6 : 0, 11 : 0 >$ PhoneAdapter applies rule ActivateOutdoor ($< 32 : 0, 33 : 0, 34 : 0, 35 : 0 >$) and adapts in state Outdoor where the configuration $< 13 : 1, 14 : 0, 15 : 1, 17 : 0, 22 : 0 >$ will trigger another rule.

Algorithm 15 Adaptation Race and Cycle Detection (Locally Symbolic)

Input: AFSM M: an A-FSM encoded using OBDDs.

Output: Set faults: set of detected faults.

```
1: faults = $\{\}$
2: **for** each StateBDD $S \in M$ **do**
3:     BDD activation = $S$.getActivation()
4:     **for** each RuleBDD $R \in S$.getActiveRules() **do**
5:         BDD ruleActivation = activation $\wedge$ $R$
6:         ruleActivation = activation.exist($M$.getDestStateVars(), $M$.getRuleVars(), $M$.getStateVars())
7:         ruleActivation = ruleActivation.swap($M$.getActionVars(), $M$.getPcvVars())
           $\{$compares with the destination activation$\}$
8:         StateBDD $D$ = $R$.getDestState()
9:         BDD futureActivation = $D$.getActivation()
10:        BDD faultInstance = futureActivation $\wedge$ ruleActivation
11:        **if** faultInstance $! = 0$ **then**
12:            faults $=$ faults $+$ $\{(S, R, D)\}$
13:        **end if**
14:    **end for**
15: **end for**
```

Listing 7.13: Fragment of output of Algorithm 14

```
Metastability Globally Symbolic: 27ms.

Found 15 faults:

Metastability [
    state = 'General'
    rule = 'ActivateOutdoor'
    futureState = 'Outdoor'
    activation = <2:1, 3:0, 4:1, 6:0, 11:0,
                  13:1, 14:0, 15:1, 17:0, 22:0,
                  24:0, 25:0, 26:0, 27:0,
                  28:1, 29:0, 30:0, 31:0,
                  32:0, 33:0, 34:0, 35:0>
]
```

7.6.3 Detecting Stability Violations With the Locally Symbolic Approach

Algorithm 15 detects stability violations with a locally symbolic approach

Intuitively the algorithm iterates over the set of all states (Line 2) and for each state checks whether any of the possible adaptations is not stable in its destination state (Line 10). If that happens an error is reported.

In order to identify configuration leading to this stability violations the algorithm first isolates, state by state, all the activations of the current rule(Line 5), then discard information about the future state, the rule and the current input, which are not necessary (Line 6), then swap the action variables with the input variables. The resulting BDD (Line 7) contains all the inputs representing the context after rule R has been triggered from state S and its action has been applied. If the conjunction of such input with the activation of the destination state is not empty it means that there are metastable configurations. Such configurations are reported in Line 12. The complexity of this algorithm is $O(|\mathcal{S}| * |\mathcal{R}| * (|\mathcal{S}| + |\mathcal{R}|))$.

Listing 7.14 is a fragment of the error report of Algorithm 15 when applied to PhoneAdapter. Note that Algorithm 15 computes the same BDD as Algorithm 14, but the former perform the same computation once for each state, while the latter does is only once but on a higher number of variables. Also note that the BDD reported by Algorithm 15 do not contain variables 22–27 because they encode the current state which is not included in the local activation BDD.

Listing 7.14: Fragment of output of Algorithm 15

```
Metastability Locally Symbolic: 5ms.

Found 15 faults:

Metastability [
    state = 'General'
    rule = 'ActivateOutdoor'
    futureState = 'Outdoor'
    activation = <2:1, 3:0, 4:1, 6:0, 11:0,
                  13:1, 14:0, 15:1, 17:0, 22:0,
                  28:1, 29:0, 30:0, 31:0,
                  32:0, 33:0, 34:0, 35:0>
]
```

7.6.4 Detecting Stability Violations with Planners

Detecting metastabilities with planners requires once again to add a support variable to give the planner the notion of consecutive adaptations. In our PDDL domain planners iterate through all the priority levels until either an adaptation is triggered or the context changes, and then the priority is re-set to its highest value and the iteration restarts.

A correct execution path would contain contextual changes between each adaptation. A metastability happens when two or more adaptations are applied without any other contextual changes in between them. Similar in the cause but worse in the side effects, a cycle happens when in the chain of sequential adaptations the same adaptation is re-triggered creating a closed loop.

To detect metastability we instrumented the extracted PDDL domain with a PDDL predicate *adaptation_triggered*, to trace the sequence of adaptations by flagging if the last PDDL action in the plan was an adaptation. PDDL effects extracted from adaptation actions satisfy *adaptation_triggered*. Similarly *adaptation_triggered* is negated by the PDDL effects extracted from contextual changes. During a correct execution the *adaptation_triggered* is never satisfied twice in a row. Consequently for each rule R we generate a goal, reporting a fault if there exists an execution path in which both *adaptation_triggered* and R are satisfied. Listing 7.15 shows the goal generated to detect races involving the rule ActivateMeeting. The number of the generated goals is equals to the number of rules. These goals detect both races and cycles without distinguishing them.

Listing 7.15: Extracted goal to detect races involving ActivateMeeting

```
(define (problem Race_ActivateMeeting)
        (:domain PhoneAdapter)
        (:requirements :strips :typing :equality :disjunctive-preconditions :fluents)
        (:objects c - context)
        (:init (= (state) 0)(= (priority) 0))
        (:goal
                (and
                        (= (state) 6)(= (priority) 4)
                        (is_true_A_t c)(is_true_E_bt c)
                        (adaptation_triggered)
                ))
)
```

It is also possible by further instrumenting the PDDL domain and by defining specific PDDL goals to detect only cycles on a given adaptation rule. For each rule R the PDDL effect can be instrumented by introducing a new predicate *R_triggered* satisfied only when S is satisfied and negated every time a contextual change is applied. Cycles on R can be detected by with a PDDL goal reporting a fault if there exists an execution path in which both *R_triggered* and R are satisfied. Listing 7.16 shows the goal generated to detect cycles involving the rule ActivateMeeting.

Listing 7.16: Extracted goal to detect cycles involving ActivateMeeting

```
(define (problem Cycle_ActivateMeeting)
        (:domain PhoneAdapter)
        (:requirements :strips :typing :equality :disjunctive-preconditions :fluents)
        (:objects c - context)
        (:init (= (state) 0)(= (priority) 0))
        (:goal
                (and
                        (= (state) 6)(= (priority) 4)
                        (is_true_A_t c)(is_true_E_bt c)
                        (activate_meeting_triggered)
                ))
)
```

Both these implementations detect one fault per rule. In certain cases developers may need to know all the possible metastable path for a given rule. This can be done by generating a PDDL goal for each path, in a similar way in which Wang at al. [WER07c] generate drivers for each CAPP.

For instance, assume to be interested in validating all the paths contaning a rule and all the rules in its destination state. To do so it is possible to generate a goal for each pair. Listing 7.17 shows the goal detecting a metastability involving both ActivateOffice and ActivateMeeting. The number of generated goals in this case is $O(R^2)$. Once again it is important to notice that two rules may be affected by multiple metastabilities, but only one per pair will be detected.

Listing 7.17: Extracted Goal for PhoneAdapter

```
(define (problem Metastability_ActivateOffice_ActivateMeeting)
        (:domain PhoneAdapter)
        (:requirements :strips :typing :equality :disjunctive-preconditions :fluents)
        (:objects c - context)
        (:init (= (state) 0)(= (priority) 0))
        (:goal
```

Table 7.4: Comparison between the Metastability fault detection algorithms

Algorithm	Detects multiple faults per rule	Detects full sequence	Identifies cycle	Reports full trace back	Scalability
Enumerative	Yes	Yes	Yes	No	Low
Globally symbolic	Yes	Yes	No	No	Medium
Locally symbolic	Yes	Yes	No	No	High
PDDL based	No	No	Yes*	Yes	High

```
            (and
                  (= (state) 6)(= (priority) 4)
                  (is_true_A_t c)(is_true_E_bt c)
                  (activate_office_triggered)
            ))
)
```

7.6.5 Comparison

The four algorithms to detect metastabilities differ greatly from each other, in terms of both detected faults and performance. The enumerative algorithm is the slowest and least scalable. Its complexity is exponential in the number of variables. On the other hand the enumerative algorithm explores all the possible sequences of activations and reports all the possible instances of faults.

The globally symbolic and locally symbolic approaches are the fastest algorithms. In particular the locally symbolic uses very little memory and scales well. More details on memory consumption can be found in Chapter 8. Both these algorithms simply identify the second of each metastability without exploring it further, therefore they are not capable of distinguishing between races and cycles.

The PDDL-based approach is the weakest in terms of detected faults. It reports one metastability per rule (or one for each pair in the second implementation). Moreover, while the other approaches can foresee metastabilities starting from the first adaptation of the chain the PDDL-based approach detects faults only when the planner is applying the second adaptation. On the other hand, the PDDL-based algorithm is superior to the others in terms of readability of error reports.

Table 7.4 summarizes the comparisons.

Chapter 8

Evaluation

In this section we explore three research questions:

- RQ1: How effective are the analysis algorithms in detecting faults? To begin answering this question we use our algorithms to analyze two open source applications, one commercial application and one example application that we crafted to analyze some borderline cases. In Sections 8.2 and 8.4 we provide a summary of the faults found, highlight some of the interesting faults and their impact, and describe the reports provided by the algorithms.
- RQ2: How do the algorithms scale as the complexity of the A-FSM increases? In Section 8.6 we again use *PhoneAdapter* and also a suite of synthetic A-FSMs to measure the performance of the algorithms as we increase the number of states, rules and variables.
- RQ3: What is the memory consumption behavior of the algorithms in limited memory environments such as would be found in end-user devices? To answer this question, in Section 8.7 we use *PhoneAdapter* in a series of configurations with small and gradually decreasing amounts of memory to measure the threshold under which execution aborts or performance degrades excessively.

8.1 Preliminaries

Our primary analysis artifact is *PhoneAdapter*, which is implemented on top of ContextNotifier, a J2ME rule-based adaptation framework and middleware for CAAAs [SRa], and targeted for deployment on the Nokia N95 cellphone. We ran the application and its adaptation rules within TestingEmulator, an emulator we have built for CAAAs [SRb]. Our implementation of *PhoneAdapter* has nine states and 19 rules, and it uses 12 propositional context variables.

Our algorithm implementations are Java 6-compliant.[1] For the fully symbolic and locally symbolic algorithms we relied on the JavaBDD library version 2.0 and on its default Java implementation of the OBDD library [Wha07].

In addition to running the algorithms on *PhoneAdapter*, *PowerManager*, *Timeriffic* and *GPS-Recorder* to count and compare the number of detected faults, we also ran two performance experiments with multiple executions of the different algorithms. The first experiment was designed to answer RQ2,

[1]The full code for our implementation and the models used in the evaluation are available online [Sam].

to measure the scalability of the algorithms. The second experiment was designed to answer RQ3, to measure the algorithms' behavior in the presence of a limited amount of main memory (i.e., JVM heap). Both the experiments and all the validations were executed on an Intel i7 920 equipped with 6GB of DDR3 RAM running Ubuntu 9.04 with the OpenJDK 6 at 64 bits. We set a timeout of 30 minutes for all analyses. For the first experiment we configured the Java virtual machine with a heap size of 4 GB. For the second experiment we decreased the JVM heap over the range from 32 MB down to 2 MB. In all the other executions we adopted the default JVM configuration in which the heap size is set to 256 MB.

8.2 Validating PhoneAdapter

We created PhoneAdapter in the early stages of this research to have a real application helping us to isolate and to identify faults. During the development various versions have been deployed on a Nokia N95, and their failures have been examined and used to identify those fault patterns discussed in Section 5.2. We used those faults to validate the correctness of our fault detection algorithms. The version of PhoneAdapter presented in this Thesis contains the most significant faults that we found during its development.

Table 8.1 summarizes the number of faulty input configurations found in *PhoneAdapter*. The first

Table 8.1: Faulty Input Configurations Reported for PhoneAdapter

State	Nondeterministic	Dead	Adaptation		Unreachable
	Adaptations	Predicates	Races	Cycles	States
General	37	1	45	13	0
Outdoor	3	0	135	23	0
Jogging	0	0	97	19	0
Driving	0	0	36	13	0
DrivingFast	0	0	58	19	0
Home	0	0	76	19	0
Office	0	0	29	1	0
Meeting	0	0	32	1	0
Sync	0	0	27	5	1

column shows the nine states of the A-FSM. The remaining columns present the number of faulty configurations found when we apply the fault detection algorithms. In general, the enumerative, globally symbolic and locally symbolic set of algorithms detects the same faulty configurations but reports them in a different fashion, as described in greater detail below. The PDDL-based algorithms instead finds up to one fault for goals, and, unless specifically instructed to do so, cannot discriminate between two faulty PCV configurations leading to the same fault. For instance if searching for nondeterministic adaptations between two rules on a given state, the PDDL-based technique will report a single detailed error trace to the first faulty PCVs configuration it encounters. This happens because goals are expressed in terms of fault definitions and not in terms of PCV assignments. The PDDL-based approach can be instructed to report more fault instances by refining the goal definition algorithms.

Detecting Nondeterministic Adaptations

This pattern of faults appears when predicates of multiple rules with the same priority and active in the same state can be satisfied by the same assignments to the propositional context variables.

By applying the nondeterministic adaptation algorithms (Algorithms 4, 5 and 6) we obtain the results shown in column "Nondeterministic Adaptations" in Table 8.1. Most of the non-deterministic adaptations were found in state *General*. The analysis for this state discovered 37 different assignments to propositional context variables for GPS and Bluetooth that simultaneously satisfy the predicates for rules ActivateOffice, ActivateHome and ActivateOutdoor.

Although the different types of algorithms detect the same faults, the way they report they varies enough to warrant presenting further details.

The enumerative Algorithm 4 returns a list of all the variable assignments and corresponding rules that can lead to a nondeterministic adaptation fault. A fragment of the reported output is shown in Listing 7.3.

The globally symbolic Algorithm 5 organizes the results in a similar but more compact way, by returning an OBDD describing the faults. An OBDD is returned for each rule containing nondeterministic activation faults, along with an indication of the interfering rules. Such OBDDs encode both inputs and rules because the nondeterminism exists among a set of rules. The error report is listed in Listing 7.4.

The locally symbolic Algorithm 6 does not return a single group of interfering rules, but instead detects pairs of such rules. While the same fault can involve more than one pair, the OBDDs containing the faults are simpler because they contain only the assignments of propositional context variables that cause the faults and not the rule variables. Faults are reported in Listing 7.5.

For the same detected faults in *PhoneAdapter*, Algorithm 6 generates four OBDDs with an average of 5.25 internal nodes, while Algorithm 5 reports five OBDDs with an average of 20.2 internal nodes. Thus, the latter are roughly four times as large as the former.

Developers interested in understanding which rules are interfering can benefit from the output produced by the globally symbolic Algorithm 5 because it groups interfering rules. The enumerative Algorithm 4 shows faulty inputs more clearly but may be too verbose. Algorithm 6 lies somewhat between these extremes.

One common way to eliminate these faults is to assign distinct priorities to the affected rules in line with the behavior desired for PhoneAdapter. In our case, we decreased ActivateOutdoor's priority to 6 and increased ActivateOffice's priority to 4.

If we verify the presence of nondeterministic adaptations between ActivateOffice and ActivateHome, the planner MIPS-XXL returns the plan in Listing 7.7 containing three environmental changes and two adaptations; remember that we are trying to reach a state in which both rules are active, and the planner produces a path from the initial state to achieve this goal as described in Section 7.2.4.

We can repeat the same exercise for the rules ActivateOutdoor and ActivateHome, obtaining the plan in Listing 8.1 containing four environmental changes and two adaptations. A similar execution pattern occurs if we look for a state in which both ActivateOutdoor and ActivateOffice can be triggered.

Listing 8.1: Nondeterministic adaptation between ActivateOutdoor and ActivateHome

```
Nondeterministic Adaptations
ActivateOutdoor , ActivateHome ]:
==========================================
0: (SET–PRIORITY−1 C )
...
9: (SET–PRIORITY−10 C )
10: (SATISFY–A–GPS C ) // GPS valid
11: (SET–PRIORITY−1 C )
...
15: (SET–PRIORITY−5 C )
16: (RULE–ACTIVATEOUTDOOR C )
17: (SET–PRIORITY−1 C )
...
26: (SET–PRIORITY−10 C )
27: (SATISFY–B–BT C ) // BT = {"HomePC"}
28: (SET–PRIORITY−1 C )
...
37: (SET–PRIORITY−10 C )
38: (SATISFY–A–BT C ) // BT = {"CarHandsfree"}
39: (SET–PRIORITY−1 C )
40: (RULE–ACTIVATEDRIVING C )
41: (SET–PRIORITY−1 C )
...
50: (SET–PRIORITY−10 C )
51: (UNSATISFY–A–BT C ) // BT != {"CarHandsfree"}
52: (SET–PRIORITY−1 C )
53: (RULE–DEACTIVATEDRIVING C )
54: (SET–PRIORITY−1 C )
...
58: (SET–PRIORITY−5 C )
```

One possible scenario for this case is as follows: The user goes in the garden and turns on his laptop. The GPS reading becomes valid while the laptop is still booting and the application adapts to *Outdoor*. When the laptop is detected the application keeps its current state and nothing happens. Someone turns on the car (maybe a user's family member), the car's handsfree is detected and the application immediately adapts to *Driving*. As the car drives away the handsfree is out of range and the application adapts to *General*, in which state the fault occurs.

The same execution pattern occurs if we look for a state in which both ActivateOutdoor and ActivateOffice are active; in this case we obtain the plan in Listing 8.2 containing four environmental changes and three adaptations.

Listing 8.2: Nondeterministic adaptation between ActivateOutdoor and ActivateOffice

```
Nondeterministic Adaptations
[ActivateOutdoor , ActivateOffice ]:
==========================================
0: (SET–PRIORITY−1 C )
...
9: (SET–PRIORITY−10 C )
10: (SATISFY–A–BT C ) // BT = {"CarHandsfree"}
11: (SET–PRIORITY−1 C )
12: (RULE–ACTIVATEDRIVING C )
13: (SET–PRIORITY−1 C )
```

```
...
21: (SET–PRIORITY−9 C )
22: (SET–PRIORITY−10 C )
23: (SATISFY–C–BT C ) // BT = {"OfficePC"}
24: (SET–PRIORITY−1 C )
...
33: (SET–PRIORITY−10 C )
34: (SATISFY–A–GPS C ) // Gps valid
35: (SET–PRIORITY−1 C )
...
44: (SET–PRIORITY−10 C )
45: (UNSATISFY–A–BT C ) // BT != {"CarHandsfree"}
46: (SET–PRIORITY−1 C )
47: (RULE–DEACTIVATEDRIVING C )
48: (SET–PRIORITY−1 C )
...
52: (SET–PRIORITY−5 C )
```

The execution trace in this case is as follows: From state *General* the car's Bluetooth handsfree is detected, forcing an adaptation to *Driving*. While driving, the office PC is detected (because maybe someone else turned it on) and the GPS becomes valid (as is most likely to happen). If the car's handsfree becomes unavailable, then the application adapts to state *General* again, where the nondeterministic adaptation can now occur.

Notice how the information provided by the planner for these examples is fundamentally different from the information provided by the State Matrix and OBDD-based approaches. In the latter cases we are only given a state (*General*, in the example above) and assignments to PCVs which cause nondeterministic activations. However, it is left to the developer to find *how* (and if) those particular assignments are possible for that particular state, as a result of actions associated with rules.

Detecting Dead Predicates

This pattern of faults consists of predicates that cannot be satisfied by any assignments to the propositional context variables, or predicates that could be satisfied but are always preempted by predicates of rules with higher priority.

The column "Dead Predicates" of Table 8.1 shows that one such fault was detected in rule *ActivateSync* in state *General*. By examining this state, we note that it is possible for predicates of the rule *ActivateSync* to be satisfied by certain assignments, but such assignments also satisfy *ActivateOffice* and *ActivateHome*, which have higher priority, and thus *ActivateSync* is never triggered.

All four algorithms for identifying dead predicate faults report these errors in the same format: affected state and rule. As this pattern of fault is caused by the absence of satisfying inputs, there is no error trace to report for the planner apart from the actual state and rule.

Detecting Adaptation Races and Cycles

Faults associated with races and cycles result from assignments that induce sequential or cyclic adaptations, where the last state of the sequence depends on how long an assignment holds.

The algorithms identified many races that produce fluctuations in the states and may disturb the user temporarily. For example, if a user starts to drive and accelerates quickly, the application may or may not reach *DrivingFast* (a state in which all calls are diverted), depending on whether the high speed is maintained long enough to enable the transition from *General* to *Driving* and then to *DrivingFast*.

There are also races generating unwanted behaviors from which the application cannot recover quickly. For instance, while in *Driving*, if Bluetooth loses the connection with the handsfree system, the phone will adapt through *General* to *Outdoor* and then *Jogging*, from where it is impossible to reactivate *Driving* even if the handsfree system is re-detected. (The adaptation rules are defined in such a way that the rule that activates *Driving* never triggers while *Jogging* is active.) Finally, all the detected cycles are produced by the rules *ActivateMeeting* and *DeactivateMeeting* when the state is *Office* and $Time >= meeting\_end$.

As shown in the columns "Adaptation Races/Cycles" of Table 8.1, these are the most common faults in *PhoneAdapter* and in our experience some of the hardest to detect without some form of automated support. With the enumerative Algorithm 13, faults are reported as depicted in Listing 7.12:

Note that Algorithm 13 explores all sequences to identify races and cycles, and for each race or cycle it reports every assignment that can trigger it. The other two algorithms, the globally symbolic Algorithm 14 and the locally symbolic A shown in Listing 7.14 Algorithm 15, report the state where a race begins, the rule causing the race, and to which destination state and with which assignment, all in the form of an OBDD. *PhoneAdapter* has 15 different sets of this kind. Developers interested in knowing *if* races exist in a CAAA will find this compact output more useful than the one produced by Algorithm 13. On the other hand, developers wanting to understand *how* the CAAA will race or cycle will find the error report of Algorithm 13 more suitable.

Detecting Unreachable States

States are unreachable when all the rules of which they are a destination state are dead. Running the unreachable state detection algorithms on an A-FSM without dead rules returns an empty set of faults. As shown in the last column of Table 8.1, state *Sync* is unreachable because the only rule with state *Sync* as destination state, *ActivateSync*, is a dead rule. All three algorithms simply return a list of unreachable states.

8.3 Validating PowerManager and Timeriffic

In this section we describe two commercial applications: *Timeriffic* [tim] and *Power Manager* [pow].

Timeriffic

Timeriffic is a simple application aimed at muting the cellphone at night and un-muteing it during the day. In the initial stages of its development, *Timeriffic* was affected by a fault causing the muting behaviour to be reversed. The fault was caused by a bug in the code in which a developer inverted the actions of the adaptation rules of day and night mode. The bug was introduced in revision 7 and fixed over two months later in revision 86.

Timeriffic, in all revisions before 86, can be modeled by means of three states: *Init*, *DayMode*, *NightMode*. When the application starts it immediately adapts to state *DayMode* or *NightMode* depending on the current time, then it sequentially adapts from one to the other at the scheduled time. We can detect this fault by imposing a state invariant in both *DayMode* and *NightMode*. In *DayMode* we require that the phone should not be on silent. In *NightMode*, instead, we require that the ringtone cannot be turned on. To prevent the planner from finding solutions in which the ringtone is turned on manually the actions *satisfy-sound-enabled* and *unsatisfy-sound-enabled* have been disabled to the user. Note that these actions are used by the adaptation actions to apply the new state. We model the concept of time with a predicate *is-true-isday*, and we initialize the goals at night. As soon as *ActivateNightMode* is applied if the state invariant of *NightMode* is violated then a fault is reported, as follows:

Listing 8.3: State invariants violation applied to NightMode

```
State invariant violation [NightMode]:
============================================
0: (SET−PRIORITY−1 C ) [1]
...
4: (SET−PRIORITY−5 C ) [1]
5: (RULE−ACTIVATENIGHTMODE C ) [1]
```

The violation of *DayMode* is easily spotted. At first, due to the initial goal's configuration, *NightMode* is applied, then it becomes day, and as soon as *ActivateDayMode* is applied then the fault is reported.

Listing 8.4: State invariants violation applied to DayMode

```
State invariant violation [DayMode]:
============================================
0: (SET−PRIORITY−1 C ) [1]
...
4: (SET−PRIORITY−5 C ) [1]
5: (RULE−ACTIVATENIGHTMODE C ) [1]
6: (SET−PRIORITY−1 C ) [1]
...
15: (SET−PRIORITY−10 C ) [1]
16: (SATISFY−ISDAY C ) [1] // It becomes day
17: (SET−PRIORITY−1 C ) [1]
...
21: (SET−PRIORITY−5 C ) [1]
22: (RULE−ACTIVATEDAYMODE C ) [1]
```

PowerManager

Power Manager is a commercial Android application available from the Android Marketplace. Out of the box, *Power Manager* maximizes the battery duration by turning on and off GPS, Bluetooth, Wifi, and by regulating the screen brightness according to the current battery level. The basic idea is to turn off each sensor when the battery is running low and to turn them on again when the battery is full or while the phone is recharging using the AC adapter or USB.

Like most CAAAs, *Power Manager* allows users to configure its adaptation logic as they please. While we were running it on an Android G1 phone with a custom configuration we found that the battery duration was reduced to one half of the normal duration. We thought that this problem might be related to invariant violation problems in the customization, and we attempted to find them with our PDDL-based verification technique. The custom configuration had five states:

1. *Initial*: The initial state when the application starts.

2. *ChargingAC*: The phone is charging using an AC charger.

3. *ChargingUSB*: The phone is charging using a USB cable.

4. *OnBattery*: The phone is running on battery and the battery charge is above 30%.

5. *BatteryLow*: The phone is running on battery and the battery charge is under 30%.

In this custom configuration, adaptations to *BatteryLow* turn off Bluetooth, GPS and WiFi to extend the battery life. Adaptations to *ChargingUSB* and *ChargingAC*, instead, are defined with the associated actions of turning on all the sensors, since the phone can use them without exhausting the battery, which is on charge. Adaptations to *OnBattery* do not perform any action since the battery is full and the phone can be used as it was configured.

Since PowerManager does not have any feature representing state invariant and since our algorithms relies on them for detecting fault we have to extend the extracted model by adding them. We added two state invariants to the two states running on battery. When the phone is in state *BatteryLow*, we require that all the sensors must be off, and the back light should be low. When the phone is in state *OnBattery*, we require that at least one of the sensors should be off or the back-light should not be high, to avoid an excessive battery consumption.

A first execution of the algorithm reports a solution in which a configuration violating the invariant was caused by changes in the environment, this represents a user configuring the phone with all the devices active on purpose, and it could not explain our observed behavior. To avoid this false positive we changed the domain by disabling environmental actions representing a user modifying GPS, Bluetooth, WiFi or the back light, and we configured the device to have all of them off at startup. With this new set-up, the planner reports an error trace for an invariant violation in state *OnBattery* as shown in Listing 7.11:

The planner finds a configuration in which the device is placed on charge. The actions associated with *ChargingAC* turn on all the sensors. When the battery is fully charged, the device is removed from the AC adapter. The application then adapts to *OnBattery*. As no action is associated to this adaptation, the phone keeps the previous configuration with all the sensors enabled and “forgets” its configuration before adapting to *ChargingAC*, thereby violating the requirement that at least one of them should be off. Thus, this sequence of events explains why the battery duration was reduced so dramatically.

In summary, both *Timeriffic* and *Power Manager* show how the PDDL-based technique can detect state invariants efficiently, and that the error traces reported are immediately usable by developers to fix the faults.

8.4 Validating GPS Recorder

In this section we introduce *GPS-Recorder*, a very simple application we developed to show how the PDDL-based approach can detect more faulty conditions than ourprevious approaches. The states and rules of the application are depicted in Figure 8.1. Essentially, this is a simple GPS-based trekking tour recording application. Tourists can rent from a base camp a GPS-enabled device on which the recording

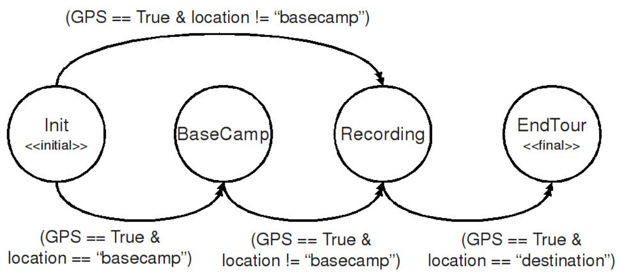
(GPS == True & location != "basecamp")
Init
<<initial>>
BaseCamp
Recording
EndTour
<<final>>
(GPS == True & location == "basecamp")
(GPS == True & location != "basecamp")
(GPS == True & location == "destination")

Figure 8.1: Crafted GPS Recorder application

application is running. When the application starts, it enables the GPS and starts reading the current location. As soon as the user leaves the base camp, the application starts collecting GPS information showing them on a map, and recording statistics about the route, including position, altitudes and speed. At the end of the tour results can be uploaded to a Web Server and stored. We model the application with four states:

1. *Init*: the initial state;
2. *BaseCamp*: the user is still in the starting point of the route and the application is not recording;
3. *Recording*: the user has left the base-camp and the application is recording times and locations; and
4. *EndTour*: in which the user has reached the destination, the recording is terminated and the statistics are uploaded.

We introduce a number of rules: A rule *ActivateBaseCamp* is triggered from *Init* if $GPS = True$ and $location = $ *"base-camp"*; A rule *StartRecording* is triggered from *Init* or from *BaseCamp* if GPS is providing a valid reading and if $location \neq$ *"base-camp"*. The tour ends when from *Recording* the application triggers *ApplyEndTour*, which is satisfied if $GPS = True$ *and* $location = $ *"destination-camp"*. The expected behavior of this application is a first adaptation from *Init* to *BaseCamp*, a second adaptation to *Recording* and a third adaptation to *EndTour*. However it should also be possible to adapt from *Init* directly to *Recording* if a user turns the application on when he has left the base camp already (see Figure 8.1).

Both the State Matrix and the OBDD-based techniques do not report behavioral faults when applied to this scenario. On the contrary, the PDDL-based approach reports that state *BaseCamp* is unreachable and that *ActivateBaseCamp* is a dead rule, which is actually the correct diagnosis. The reason for this is that the State Matrix and OBDD-based techniques find that in *Init* a configuration $GPS = True$ *and* $location = $ *"base-camp"* exists, which satisfies an adaptation to *BaseCamp*. However, the reachability of this configuration from the initial state is not checked. The PDDL-based approach, instead, always starts from a chosen initial configuration and as a consequence it is not able to find any valid path capable of triggering *ActivateBaseCamp* and therefore capable of reaching the state *BaseCamp*.

Even in this simple crafted example it takes some time to spot that there is no direct path from a stable and valid configuration to any configuration triggering *ActivateBaseCamp*; similar situations may occur in applications with more rules and they can be very difficult to detect.

In summary, the *GPS-Recording* application shows how the PDDL-based technique finds dead predicate faults that cannot be detected by the other techniques.

8.5 Random Model Generation

To further assess the algorithms' performance, in addition to *PhoneAdapter*, *Timeriffic*, *Power Manager* and *GPS-Recording*, we generated a set of synthetic A-FSMs of increasing complexity with various numbers of states, rules, and variables. To make the A-FSMs more realistic, we constrained the generation process as follows:

1. All states are the destination state of at least one rule (with the exception of the initial state), which avoids having unreachable states that are trivially detected. The number of active rules in each state is between one and eight, following what we have observed in practice in publicly available tools.

2. The active rules of each state all have different priorities in order to guarantee that the generated A-FSM is deterministic, since determinism is a prerequisite for most of the algorithms.

3. The number of variables used in each rule predicate is less than a specified maximum, and all variables are used in at least one predicate. In the experiment we imposed a maximum of five variables per predicate, in order to generate predicates of a complexity that represents what we have observed in practice.

4. The variables used in each predicate are composed using a combination of negation, conjunction and disjunction according to the following probabilities: Each variable has a 50% probability of being negated, and conjunctions or disjunctions are selected with a 50% probability.

We used these synthetic models in the following experiments to better evaluate the performance of our techniques with models of different complexity.

8.6 Evaluating Performance

Table 8.2 reports the performance of the algorithms running on the A-FSM of *PhoneAdapter* and the randomly generated CAAAs. The size of the random CAAAs is reported in the first column as a triple specifying the number of states, rules and variables. Performance times are reported in milliseconds, represent the average over 10 runs, and include the time both to detect faults and (where applicable) to decode the generated OBDD. For each algorithm we measured the time required to generate the model in memory, which is shown in the columns labelled "MG". For the enumerative algorithms, the "MG" value corresponds to the time to create the matrix. For the fully symbolic and locally symbolic algorithms, the value corresponds to the time to compute the state activation OBDDs, plus in the fully symbolic case, the global activation OBDD.

Note that the three parameter of this experiment are not independent. Indeed the maximum number of rules is limited by the number of variables and the maximum number of states is limited by the number of rules. E.g. to connect n states it necessary to have at least $n-1$ rules. Moreover we observed that in a

Table 8.2: Performance Results for The Three Classes of Algorithms (milliseconds)

States/Rules/Variables	Enumerative					Globally symbolic					Locally symbolic				
PhoneAdapter	82	3	1	207	2	16.9	5.9	0.9	10.5	1.3	13.1	0.2	0.8	8.9	0.8
Step 1															
(10,40,10)	12	6.5	4	13085	8.3	24.3	19.3	1.4	26.4	5.5	13.45	0.15	0.6	10.55	0.5
(10,40,15)	327.3	49.1	39.5	127442	131.8	130.4	109.6	9.5	240.4	37.1	18.6	0.4	1.3	17.2	0.7
(10,40,20)	OUT OF MEMORY					955.2	1033.0	158.0	2654.5	369.4	24.8	0.7	2.0	11.0	1.0
(10,40,25)	OUT OF MEMORY					13522.0	19179.9	2384.8	48032	6939.3	25.1	0.5	1.2	13.0	0.9
(10,40,30)	OUT OF MEMORY					OUT OF MEMORY					26.7	0.3	1.5	10.7	0.9
(10,40,35)	OUT OF MEMORY					OUT OF MEMORY					28.0	1.6	2.8	9.9	1.8
(10,40,40)	OUT OF MEMORY					OUT OF MEMORY					35.3	0.6	5.3	15.7	2.6
Step 2															
(10,45,15)	TIMEOUT					157.2	130.0	14.5	314.6	61.0	22.4	0.4	2.0	12.6	1.1
(10,60,20)	OUT OF MEMORY					4343.4	2616.7	492.4	12099	1366.1	51.7	0.9	3.9	19.8	1.6
(10,75,25)	OUT OF MEMORY					OUT OF MEMORY					54.4	2.2	16.6	29.5	2.6
(10,90,30)	OUT OF MEMORY					OUT OF MEMORY					71.2	2.0	43.9	73.7	4.3
Step 3															
(10,30,10)	11.0	5.4	2.7	349.5	4.6	18.2	15.9	1.8	22.3	3.7	15.7	0.3	0.6	6.2	0.8
(15,45,15)	204.1	41.6	23.6	46718	39.6	223.0	153.6	13.5	336.6	54.0	20.6	1.1	0.9	14.9	1.1
(20,60,20)	OUT OF MEMORY					7737.4	4607.9	374.0	11207	1469.2	33.5	0.2	1.0	6.7	1.7
(25,75,25)	OUT OF MEMORY					OUT OF MEMORY					34.3	1.3	3.1	11.4	5.5
(30,90,30)	OUT OF MEMORY					OUT OF MEMORY					35.7	0.6	3.6	11.4	6.3
(35,105,35)	OUT OF MEMORY					OUT OF MEMORY					39.7	0.7	3.4	10.8	8.4
(40,120,40)	OUT OF MEMORY					OUT OF MEMORY					47.6	4.1	4.9	11.3	17.2
(45,135,45)	OUT OF MEMORY					OUT OF MEMORY					66.3	0.9	2.8	15.5	8.6
(100,300,100)	OUT OF MEMORY					OUT OF MEMORY					169.7	1.5	6.3	18.9	33.3
(200,600,200)	OUT OF MEMORY					OUT OF MEMORY					1024.6	2.6	16.8	43.9	191.8

MG: model generation, ND: Nondeterministic Activation, DR: Dead Rule, AR: Adaptation Race, US: Unreachable State.

real implementation the number of rules is always greater than the number of states and that the number of predicates increase proportionally with the number of states.

The studied A-FSMs are meant to expose the performance of the algorithms under various configurations. In our study we first manipulated the number of variables while keeping the number of rules and states constant. Second, we manipulated the number of variables and rules while maintaining the number of states constant. Third, we manipulated all three factors.

Overall we find that the enumerative algorithms do not scale to larger CAAAs and are generally slower than the others, especially in the detection of races and cycles. The symbolic algorithms scale only slightly better than the enumerative ones. The locally symbolic algorithms are the fastest, which was unexpected since it is normally assumed that "pure" symbolic approaches are more efficient than ones including any type of enumeration like the locally symbolic one. Further study reveals two reasons that make the locally symbolic approach faster for our particular domain.

First, the OBDDs manipulated by the locally symbolic algorithms are generally smaller than the OBDDs manipulated by the globally symbolic algorithms, so even if the number of loops in the locally symbolic algorithms is greater, the smaller size of the OBDDs makes the algorithms faster overall. We can measure the complexity of the state activation OBDDs by the number of nodes and the number of paths that satisfy the three [Bry92]. Table 8.3 reports the average number of nodes and paths for the state activation OBDDs of *PhoneAdapter* (with the average taken over all states) and compares those with the total number of nodes and paths of the global activation OBDD. We note that the globally symbolic algorithms deal with an OBDD of almost two orders of magnitude larger than the locally symbolic ones.

Table 8.3: Comparison of OBDD Complexity (PhoneAdapter)

	State Activation OBDD (state average)	Global Activation OBDD (total)
Nodes	24.2	935
Paths	12.2	6008

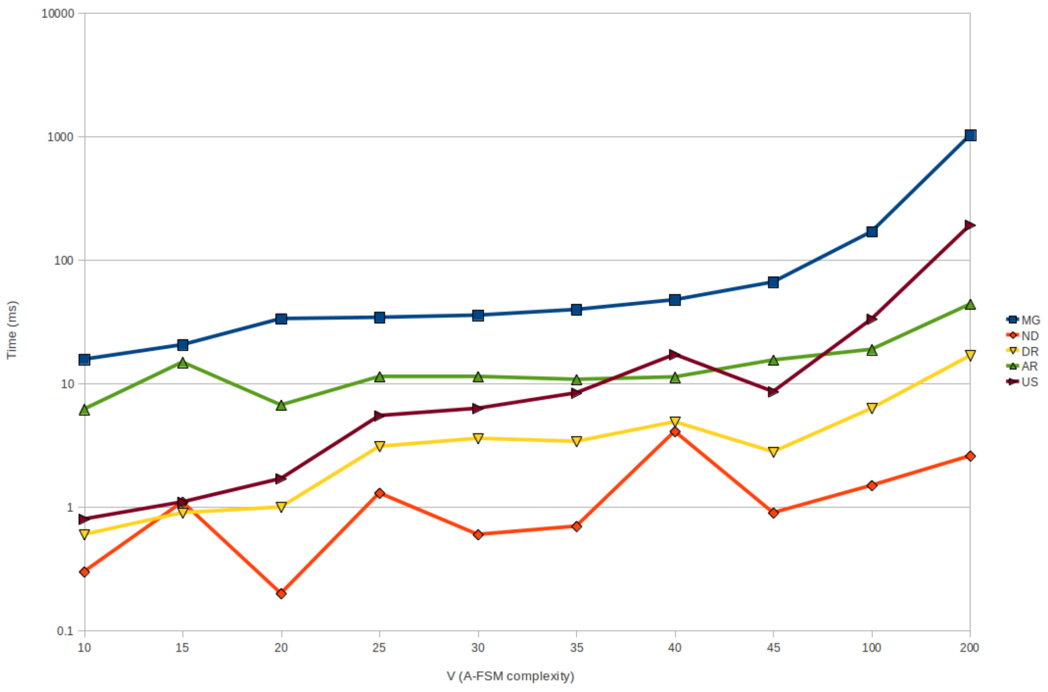

Figure 8.2: Step 3: execution times of the locally symbolic algorithms.

This is due to the fact that, in the worst case, the complexity of an OBDD increases exponentially in the number of variables used.

Second, while a globally symbolic approach can be slightly faster in detecting faults, it returns *all* the faults encoded as a single OBDD, which then must be decoded; this decoding process is a further bottleneck of the globally symbolic approach. In contrast, with the locally symbolic algorithms all the faults are reported as soon as they are found, and despite the fact that the number of iterations is greater, the size of the OBDDs is smaller, and there is no decoding required. More generally, note that we evaluated our algorithms only on a single CPU core. Better results can be obtained with the locally symbolic approach parallelizing the the exploration. For instance faults local to a single state could be detected by exploring each state in a dedicated thread.

In the third step of the experiment in Table 8.2 we fixed the proportion between the number of rules, states and PCVs. In particular we imposed that $S = V$ and $R = 3 * V$ where S is the number of states, R is the number of rules and V is the number of PCVs. Figure 8.2 shows the plot for the executions of the locally symbolic algorithms for Step 3. The Y-axis represent the execution time in logarithmic scale. The X-axis represent both V and the complexity of the A-FSM. From the graph we can observe that as

Table 8.4: Performance while Testing PhoneAdapter with Limited Memory (milliseconds)

	Enumerative					Symbolic					Hybrid				
Memory (MB)	MG	ND	DR	AR	US	MG	ND	DR	AR	US	MG	ND	DR	AR	US
32	24.6	0.6	0.4	66.5	1.1	24.8	16.25	0.85	21.2	1.15	19.8	0.25	0.65	30.3	0.7
24	26.2	0.8	0.3	67.7	1.3	20.7	23.8	0.8	17.25	1.2	23.1	0.25	0.65	31.55	0.8
18	32.3	0.7	0.6	80.6	0.7	31.6	15.4	0.85	OOM	OOM	24.3	0.3	0.65	71.45	0.75
16	42.0	1.6	0.4	83.2	0.6	28.45	31.6	0.85	OOM	OOM	29.05	0.45	0.7	OOM	0.7
12	63.4	0.55	0.9	87.3	0.6	41.25	OOM	OOM	OOM	OOM	34.2	0.15	0.75	OOM	0.75
8	269.5	1.2	0.65	OOM	1.45	54.4	OOM	OOM	OOM	OOM	44.85	0.4	0.7	OOM	0.7
4	268.1	1.65	0.55	OOM	1.65	55.6	OOM	OOM	OOM	OOM	47.55	0.35	0.55	OOM	0.9
2	276.0	1.6	0.6	OOM	1.7	51.15	OOM	OOM	OOM	OOM	48.7	0.15	0.65	OOM	0.65

MG: model generation, ND: Nondeterministic Activation, DR: Dead Rule, AR: Adaptation Race, US: Unreachable State.

expected the model generation grows with complexity $S * R$ which in this case is equals to $3 * V^2$ while the reachability grows as $|S|! = |V|!$. We can also observe as with the increase of the A-FSM complexity none of the execution times remains linear.

8.7 Evaluating Memory Consumption

In the experiments described thus far, we let the algorithms use all the memory at our disposal (4GB). To better understand the trade offs between space and speed, we next explored the performance of the algorithms on *PhoneAdapter* in more constrained memory settings. Such settings are also interesting in practice since more CAAAs are enabling end-users of limited memory devices to redefine adaptation rules, which can also be faulty and targets of our analysis.

Table 8.4 summarizes our findings. We observe that, for all algorithms, as memory decreases the time required to generate the model increases slightly, in part because the garbage collector is invoked more often.

All algorithms can run successfully on *PhoneAdapter* with 24MB of memory. However, with less than 18MB the globally symbolic algorithms to detect races and unreachable states (Algorithms 14 and 11) run out of memory. Since the globally symbolic algorithms use a data structure that is an order of magnitude bigger than the one used by the locally symbolic algorithms, it was expected that they would have run out of memory sooner. If the available memory is decreased to 16MB, then the locally Algorithm 15 (the only locally symbolic algorithm that operates on multiple states) runs out of memory as well. If the memory is further reduced, all globally symbolic algorithms fail. With the exception of Algorithm 15 and Algorithm 13, all the other locally symbolic and enumerative algorithms manage to run with just 2MB of memory. Overall, the enumerative approach with its small data structures seems to be a better fit for constrained settings. This result seems to contradict the one reported in Table 8.2 in which the enumerative approach had the largest footprint. The obtained results can be explained in terms of memory footprints for the single data structure. An instance of OBDD is a complex data structure, which contains several support structures which are needed during the computation but which are initialized at startup. Contrarily the enumerative approach uses only data structures provided by the Java framework such as HashMaps or ArrayLists. The results of this experiment show that for a model of the size of PhoneAdapter the simpler data structures used by the enumerative approach have a smaller memory footprint than OBDDs. Therefore developers willing to validate a small model in a strictly limited memory environment should prefer the enumerative approach.

Note that we run this second experiment in the same machine of the previous one, therefore the obtained performances will out stand any execution on an handset. Similarly our code and all the included libraries were compiled for a desktop machine, therefore they perform better but require more memory as they would if compiled for a mobile device. Also note that we limit the memory by using the *"-Xmx"* flag of the virtual machine which only limit the heap memory and not the stack. Implementation of the Java virtual machine running on mobile device may allocate smaller stack which may cause recursive algorithms to throw a stack overflow exception. All of our algorithms are iterative so we do not have stack problems.

8.8 Summary

We finalize the performance study by qualifying our findings. We observe that today's CAAAs are mostly of the size of *PhoneAdapter*, and that simpler algorithms like the enumerative ones may suffice. Still, the growing complexity of CAAAs we are witnessing indicates that the application of more efficient algorithms will become increasingly important. At the same time runtime validation of user defined specifications is becoming more and more a requirement making also memory consumption a strict constrain.

The need of having validation algorithms both scalable and memory efficient makes it necessary to be able to measure the complexity of a CAAA. In this thesis we show how the complexity of CAAAs depends simultaneously on the number of PCVs, rules and states and how this three parameters are connected to each other. In particular the number of states affects the number of rules and the number of rules affects the number of PCVs. It would possible to define a complexity index as a function combining this three parameters altogether. Such complexity index could be used to compare the complexity of different CAAAs.

The lesson that we have learned in the course of this thesis is that by localizing the fault as close as possible in a state or in a rule we were able to reduce the cost of each detection algorithms. We observed that there several source of complexity which sometimes are unnecessary and that can be avoided. For instance a source of complexity in our algorithms was the need of decoding the detected faults in a human readable form. This was particularly affecting the performances of the symbolic algorithms adding a linear dependency. If the results were not supposed to be used by developers but by other software components then it could be possible to reduce or even remove that overhead.

Similarly the complexity of the algorithm computing the state activation BDD is related to the number of involved boolean variables. However most of the algorithms do not use all of those variables (e.g. the from being included in the action effects, or the future states). but since we use the same activation BDD for all the algorithms we were forced to include them. Not only those extra variables increase the size of the BDD but also sometimes need to be removed or they will interfere with the results. Such unused variables can be removed by means of the existential quantification function *exist()* which in its worst case has exponential complexity. Developers in need of detecting only a certain class of fault could optimize the state activation BDD avoiding those variables from being included in the first place.

Chapter 9

Conclusion

9.1 Contributions

In this Thesis we have investigated a modern class of applications which base part of their behavior on readings from sensors embedded on the hardware on which they are running and which adapt according to such sensor readings. We have called applications belonging to this class *Context-Aware Adaptive Applications*, *CAAAs* for short. In particular we have investigated the most common faults in such applications and how they can be prevented or detected. We fulfilled our investigation by providing five different contributions.

9.1.1 Architecture Model and Fault Taxonomy

In Chapter 3 we have demonstrated that CAAAs have a common base architecture and that such architecture is evolving with the introduction of modern frameworks and mobile operating systems. The same evolution that we have described has been identified also in existing applications, proving the soundness of our intuition.

According to our architectural model, CAAAs suffer from an intrinsic problem: context values may contain errors and are not available instantaneously. This is a direct consequence of the data processing with which the environment is perceived, loaded and represented in the program memory. Sensors acquire context values with a precision and with a refresh rate fixed by their manufacturer. Software applications may try to prevent or correct such errors or to guess intermediate values in between two readings, but if the context changes too quickly for the sensors to correctly perceive the changes then the application will always compute incorrect data. Certain CAAAs use sensor fusion techniques to compensate reading errors by reading from multiple sensors. However, such solutions still suffer from reading errors plus they require the application to deal with inconsistent readings. In addition, in mobile devices, to preserve battery, certain sensors may be turned off while needed and application developers must remember to turn them on.

CAAA developers can benefit from our architectural model to foresee limitations and to prevent sensor-related side effects. For instance, the Locale user support center had to inform their users of intrinsic issues with GPS on Android, and in their user support center they added the following explanations: *"Why does it sometimes take a while to update my situation? For most conditions (battery, time,* *contact), Locale will update immediately. For some conditions, such as location, orientation, or some plug-ins, it may take up to 10 minutes to detect a change in order to conserve battery life."* and *"Why does Locale only seem to change my settings when I tap on the Locale icon? For most conditions Locale will update immediately, although certain conditions may take up to 10 minutes to update in order to conserve battery life. What you've noticed is that Locale will go ahead check your conditions again when the app is opened."*

Our architectural model also underlines to developers all the sequences of conversions and approximations that are applied to contextual information before they become available as variables usable by the different layers of the application. Being aware of the stack of transformations is crucial for developers in order to understand and eliminate bugs. To further help developers in identifying, classifying and fixing bugs we have presented a fault taxonomy. Our taxonomy is meant to be used as an extension to existing bug reporting and tracking tools to better classify the severity and the source of each bug. To prove the usability of our taxonomy we have applied it successfully to the faults we detected in our case studies.

9.1.2 A-FSM Model

According to our architectural model, and according to the applications that we have examined the component on which developers should focus the most is the application logic. With the support of modern operating systems and context-awareness frameworks, most of the components are already available and developers have just to focus in writing the application logic. Moreover, in modern applications the adaptation logic is designed by end users with a GUI, and developers have very little control over the logic itself.

To help developers in improve the design of their application logic or to give them a better representation of the logic designed by their users, we have proposed the A-FSM model. The A-FSM model helps developers in focusing only in the application logic, delegating frameworks and OSs to do the rest, by focusing them in the right abstraction level. We have used the A-FSM to represent all the case studies of this Thesis and we have shown that it can easily represent any CAAAs without loosing any information.

The definition of the A-FSM is a key contribution of this thesis because it is the base for the definition of our fault patterns and our validation techniques.

9.1.3 Fault Patterns

While applying the A-FSM model to our scenarios we noticed that certain faults had similar causes in the application logic. In this thesis we isolated some of these common causes in the form of fault patterns identifying properties which have to be respected in the model and which, if violated, lead to faults.

The definition of these fault patterns is useful to rule developers because it makes them aware of several known issues and helps them in creating a more robust application logic right from the earliest development stage. In applications in which users are defining their own rules, and thus in which developers cannot validate the logic, our properties can help developers in imposing limitations on what users can and cannot define. This is similar to what Locale's developers impose on their users. As discussed in the previous chapter, in Locale the rule editing GUI strongly limits the flexibility of user defined predicates in order to limit the occurrence of faults.

In addition the formal definition of faults patterns has allowed us to design automated techniques to expose automatically those faults in the A-FSM model.

9.1.4 Validation Techniques

Starting from our A-FSM model and built on top of the fault patterns that we have identified, we implemented four automated validation techniques. We also provide an implementation of these automated validation techniques allowing application developers to easily verify their application starting from an A-FSM model.

The four techniques we describe use different approaches. The enumerative approach sequentially inspects each different PCV configuration and reports anomalies at the granularity of individual configurations. For faults involving a sequence of adaptations, fault detection algorithms have to take care of tracing adaptations. Faults affecting multiple configurations have to be aggregated by the algorithms themselves to reduce complexity. Symbolic algorithms aggregate PCV assignments using OBBDs and reported faults are aggregated by design in the result OBDDs. The PDDL based approach delegates planners to identify faults and reports a full trace from an initial configuration to the fault occurrence.

Among those approaches there is no superior technique: each one has its own strengths and weaknesses. Developers should choose the one that best fits their needs. Developers implementing a CAAA with a certain application logic known a priori can design first an A-FSM of their application, validate it, and prevent most of the faults at the design stage. In this case developers could perform an extensive validation on a dedicated server using multiple techniques and comparing the results. For modern CAAAs in which users define their own rules, our validation techniques could be embedded in the application and validate the user-designed configuration at runtime.

9.2 Evaluations

We have evaluated our techniques both by applying them to our case studies and by examining their performance in the experiments described in Chapter 8. We can compare our techniques in terms of reported faults, error reports, performance and memory consumption.

Even if the PDDL-based technique reports at most one instance of a fault per goal we can say that the four techniques have almost the same effectivenes in terms of detected faults. Multiple faulty configurations are normally part of a continuous subspace of PCVs in which several assignments lead to the same fault. The OBDD-based techniques aggregate such subspaces in the reported OBDD while the enumerative approach reports a fault for each faulty assignment. In the PDDL-based approach, if a goal would reveal two of those subspaces (i.e., the faulty subspace is partitioned), then only one partition is reported. However, developers can iterate and fix such subspaces one by one. Moreover for the Reachability property there are some borderline cases in which the Enumerative and OBDD-based approach suffers from missing positives, while the PDDL-based approach is able to properly identify the fault.

Contrarily, in terms of fault reports the PDDL-based approach is superior to the others because it generates a complete trace from an initial configuration to the fault occurrence. Such trace provides useful information to understand the causes of the detected faults. The OBDD-based approaches aggregate faults in OBDDs helping developers in understanding the severity of each fault. Among them the locally symbolic approach is better than the globally symbolic because the OBDDs that it reports are generally smaller than the other. In terms of reported faults the enumerative approach is slightly worst than the others because it does not aggregate faults and does not provide a trace.

Performance of the PDDL-based approach depends on the planner used. Determining which planner is the better or faster is out of the scope of this thesis. Planning related conferences traditionally host performance competitions in which various planners are evaluated against a set of known benchmarks. To fulfill our validation the selected planner needs to support PDDL 3.0 and fluents. We observed that some planners have issues with complex predicates or with predicates containing double negations. We do not know if that is due to some bug in the implementation or to some internal optimization. To avoid these issues our implementation automatically compacts predicates using De Morgan's laws. Among the planners we tried we recommend MIPS-XXL [EJN06] because it revealed itself as the most stable one. The goal generation execution time takes a few milliseconds and the planning takes a few seconds, depending on the detected faults. Note that planners stop searching for valid plans according to a given heuristic function. The execution time (as well as the detected faults) are affected also by that heuristic function. Evaluating performance and trade offs of such heuristic functions is out of the scope of this thesis. We noticed that the default ones managed to detect faults in a reasonable time.

The scalability with respect to performance of the enumerative, globally symbolic and locally symbolic approaches has been evaluated by means of the experiment in Section 8.6. CAAAs scale in three dimensions: number of PCVs, number of rules and number of states. Note that such dimensions are not independent from each other. Theoretically the enumerative approach is exponential in the number of PCVs, the locally symbolic is linear in the number of states, and the globally symbolic should do all the computation in one single symbolic computation (with a complexity depending on the computation itself). According to our results the enumerative approach follows such expectations and does not scale beyond a small number of PCVs. Surprisingly the locally symbolic approach scales nicely and manages to handle even huge models with two hundred states. The globally symbolic approach does not scale as nicely as the locally symbolic approach and can hardly handle models with more than 25 states. Therefore, from the scalability point of view, we can claim that the locally symbolic approach is superior to the others, and that enumerating on the states and validating them symbolically appears to be the best solution. As explained in Chapter 8, the globally symbolic approach does not scale because its OBDDs are two orders of magnitude bigger than the local ones generated by the locally symbolic approach. If we look at the problem from a different perspective, states represent independent behaviors that the application can assume. According to this it makes sense to validate them separately. Moreover it would be possible to further improve the scalability of the locally symbolic approach by validating states in parallel, for instance in a multi-core machine or in a cloud.

CAAAs in which users can define their own application logic need to validate such rules before they are applied. This need leads to the question of which approaches are better to be applied on a memory limited device such as a smartphone. Since none of the existing planners runs on any existing embedded device, and since if there will be a porting for embedded devices its performance will depend on its implementation we decided to exclude the PDDL-based approach from our evaluations of memory consumption 8.7. We were expecting this second memory consumption study to give results similar to the scalability one. As expected, the locally symbolic approach resulted superior to the globally symbolic. Unexpectedly, the approach which seams to run better in an environment with very limited memory is the enumerative one. Indeed, the enumerative approach only uses hash maps to store the state matrixes and it does not require any additional support data structure such as the nodes and indexes which are required to create an OBDD. Therefore we can claim that the best approach to validate a small model with a very limited memory is the one with the simplest data structure, which in our case is the enumerative approach. According to our results the minimum required amount of memory to validate a model of the size of PhoneAdapter is at least 10MB. Notice that our implementation was not optimized for embedded devices, and therefore it may be possible to achieve better results with a dedicated implementation.

9.3 Future extensions

The research work presented in this Thesis can be extended in various directions. In this section we quickly discus the ones that in our opinion are the most promising or the more useful.

Automated model extraction. Those CAAAs in which the application logic is user defined would strongly benefit from the fault validation that we have presented in this thesis in order to prevent users from running a faulty configuration. However in this thesis we assumed the A-FSM model to be given or to be designed accurately by developers. This assumption is not valid for application in which the logic is designed at runtime. To address such applications the work presented in this thesis can be extended to include some form of runtime model extraction.

Most of the consumer market, which is at the moment the area in which this extension is needed, is divided in two main areas: Android and iOS applications. Both Android and iOS rely on SQlite [sql] for storing user data, meaning that all the user-designed states, rules and PCVs are stored in an internal database. It is therefore theoretically possible to read those databases and automatically create the A-FSM. Note that for security reasons, in both these architectures the databases are locked and only the application owning them can read them.

Web based applications. There are various similarities between mobile embedded applications and web-based applications. Indeed, most web based applications handle the user interaction according to a fixed state machine and suffer from faults similar to the ones reported in this Thesis. Similarly in a SOA (Service Oriented Architecture) composition of context-aware web services suffers from bugs related to the way context is handled on multiple web services. We did some preliminary work in this field by validating a set of known properties on a static composition of context-aware web services [CSRR09]. We successfully underlined the similarity between our CAAAs and context aware web services. However our preliminary work can be extended both by exploring more exhaustively which kind of faults affect web service compositions and both by validating dynamic web service compositions by taking in account also runtime issues such as context-aware service discovery.

Runtime failures (Hazards). Gates in logic circuits suffer from what is called *hazard*. Hazards are those phenomena in which the output of a logic gate in which more than one of its signals changes has an instability or "glitch". Such a glitch can erroneously be interpreted by the circuit as the signal to perform a certain action. Hazards happen because, when two or more inputs change simultaneously, the delay between when they change and when the gate receives their new value depends on the number of other gates on the path between the changes signals and the receiving gate. The time between when the gate acknowledges the first and the last update is called a *transitory*. Digital circuits designers have developed various techniques to prevent their output from being affected by hazards. We noticed that a similar phenomenon also happens in CAAAs when a predicate is composed by PCVs over multiple context variables and when two or more context variables change simultaneously. We did some preliminary work in detecting hazards in CAAAs [SRWE08a], and we proved that they exist and that they can be detected and prevented.

Appendix A

Hazards

In Chapter 5 we presented a set of properties whose violation leads to a certain category of faults. Respecting those properties guarantees that the logic behind the behaviour represented in the model will not be affected by faults of such categories.

In our research we came across a series of anomalies in which, even applications whose model where respecting all the behavioural properties, were failing. Such anomalies, once identified, were easily reproducibly by feeding the CAAA under test with a certain input (e.g. by recreating certain condition with TestingEmulator [SRb]). We observed that those anomalies were caused by the boolean trigger predicate of the rule in our model. We also observed that their occurrence was nondeterministic and implementation dependent.

We named such anomalies *Adaptation Hazards*, as we found similarities with hazards in digital circuits. Hazards are both related to the formula in the the rule's trigger predicates but also to the implementation of each single CAAA. Model based validation techniques applied to a phenomenon which occurrence is implementation dependent will suffer from false positives and negatives. To prevent this from happening the A-FSM model should be extended with low level implementation and architectural details including information about the number of thread reading from sensors, used signals, monitors, semaphores and any other details involving the internal scheduling of the application. Thus we consider hazard detection a branch of this research which will lead to future works.

In the rest of this Appendix are presented some initial result that we had in hazard detection as we presented them in one of our publications [SRWE08a]. The following results assume that the implementation uses one dedicated thread pooling over each single sensor and recomputes the trigger predicate after each reading. This reactive architecture is very common among CAAA but is not the only possible.

A.1 What Hazards Are

Even if a set of rules for a CAAA satisfies the desired behavioral properties described in Chapter 5, they still may suffer from faults related to the asynchronous way in which context variables are updated. We can treat the effects of delays in asynchronous updates to context variables as *hazards*, similar to those found in sequential digital circuits. Thus, hazards in a CAAA arise not as a result of the logic of the rules, but as a result of the way physical changes to context propagate to the evaluation of rule predicates.

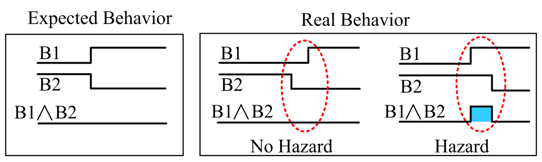

Figure A.1: A Static 0-Hazard in an AND-Gate.

In Chapter 3 we described how the layered architecture of CAAAs gives rise to four different views of the context, two of which are important for detection of context hazards—the *physical context* and the *sensed context* [SRWE08b, SRWE10]. The physical context is the context as it exists physically in the environment of a CAAA. The sensed context is the discretization of continuous physical context values that results when a context-awareness middleware periodically reads the values from sensors and stores them in the set $\mathcal{C}_{sensed}$ of sensed context variables described in Chapter 5. For instance, in a physical context in which a user is driving a car with his phone actively paired to a Bluetooth handsfree system, the pairing would be represented by a periodically sensed context value such as "BT 00:01:A6:23:FD paired".

Whenever multiple changes occur to the physical context of the CAAA, the internal representation of the sensed context will be inconsistent with the physical context, or *stale*, until *all* the relevant sensed context variables variables have been refreshed. Evaluating rule predicates on a stale sensed context exposes the system to hazards, namely to incorrect or unexpected adaptations. Such faults exist due to three related reasons: (1) the predicates of rules are re-evaluated every time a sensed context variable is updated; (2) the sensed context variables are updated asynchronously according to different refresh rates; and (3) synchronizing the updates of the sensed context variables is difficult (because typically they are updated by different sensor-specific run-time libraries) and undesirable (because of the resulting degradation in performance).

For a given predicate, the occurrence of a hazard depends on the *commutation order* of the predicate's constituent propositional context variables. We illustrate this in Figure A.1 with the simple case of an AND-gate that takes two inputs, B1 and B2, which are initially 0 and 1, respectively, thus producing an output of 0. Suppose the inputs undergo a 2-commutation to the values 1 and 0, respectively, producing an output of 0 again. If B2 commutes first, then the output of the gate does not change. However, if B1 commutes first, then the output transiently has the value 1 until B2 commutes, thereby exposing the hazard.

In the adaptation rules of a CAAA, we can identify three different patterns of faults:

- A *Hold Hazard* occurs when the rules adapt to a new state in a situation when the current state should prevail instead. This is similar to the *static 0-hazard* depicted in Figure A.1. From the user's point of view this fault produces an unwanted adaptation.

- An *Activation Hazard* occurs when the rules adapt to a new state before all relevant variables have commuted during a commutation of multiple variables, and the new state is different from what is expected.

- A *Priority Inversion Hazard* is similar to an Activation Hazard and occurs when the rule that triggers has a lower priority than the one that should have triggered. Activation Hazards and Priority Inversion Hazards are similar to a *dynamic hazard* in sequential digital circuits [Ung95], where a different output is produced before the evaluation is completed. From the user perspective the system performs an incorrect adaptation and incorrect actions.

A.2 Detecting Hazards

Static hazards in sequential digital circuits are eliminated by introducing delays (such as a double negation) into a specific signal path [Ung95]. A solution for the adaptation rules of CAAAs could be to introduce delays in the invocation of the actions of every triggered rule in order to make sure the destination state of the triggered rule holds. Unfortunately, this would apply a fixed delay even to safe commutations that do not create problems. Therefore, we instead focus on identifying which commutations of propositional context variables may lead to a hazard and then compute the smallest delay that will avoid it.

For a given state, we define a *stable assignment* as an assignment of values to the propositional context variables that satisfies none of the predicates of the active rules of the state. Correspondingly, an *unstable assignment* is an assignment that satisfies some predicate. We define a *critical path* as any sequence of commutations that starts with a stable assignment and has one or more intervening unstable assignments. If a critical path ends in a stable assignment for the given state, then we have a Hold Hazard. If the critical path ends in an unstable assignment, then we need to check for a Priority Inversion Hazard. Otherwise we have an Activation Hazard. We only consider critical paths in which each variable commutes at most once, since a critical path with multiple commutations of the same variable can be subdivided into multiple critical paths with single commutations. In addition, we assume that multiple propositional context variables associated with the same underlying sensed concrete variable are updated simultaneously whenever the sensed concrete variable is updated. We can relax this assumption by accounting for any implementation delays in the updating of the propositional context variables.

Algorithm 16 is applied in each state S and searches for hazards beginning from stable assignments, which Line 2 identifies as the set of all bit strings satisfying *no* predicates of active rules of S. Lines 4–31 explore all commutations from length two to the number of propositional context variables, with the loop variable i indicating the current length to consider. Line 5 generates the set of permutations of length i of indexes into the current bit string, indicating the different variables and their orderings to consider for commutations. For each permutation, Lines 6–30 sequentially commute the variables according to the current permutation and look for hazards under that setting. Line 15 discards a stable path that has been processed already in a previous shorter path. Lines 16–20 detect Priority Inversion Hazards when a rule R is discovered with higher priority than rules found before. Lines 21–27 detect Hold Hazards and Activation Hazards. A Hold Hazard is present when there is an adaptation (indicated by *!isEmpty(rules[])* in Line 22) between stable assignments. Line 29 reports all detected critical paths and their hazard category.

Algorithm 16 HazardDetection

Input: *M*: an instance of A-FSM.

Output: *faultsVector*: vector of detected faults.

```
1:  for each state $S$ in $M$ do
2:    stableAssignments = complementBitStrings($S$.getStateMatrix())
3:    for each bitString in stableAssignments do
4:      for i = 2 to $S$.numVar() do
5:        indexList = permutation($S$.getVars(), $i$)
6:        for each sequence in indexList do
7:          rvector = {} // selected rule in each step
8:          $u$ = bitString
9:          hazard = null
10:         for j = 0 to $i$ - 1 do
11:           $u$.flipBitAtIndex(sequence[$j$])
12:           rules[] = $S$.getSatisfiedRules($u$)
13:           $R$ = getHighestPriorityRule(rules[])
14:           if isEmpty(rules[]) && (j != $i$ - 1) then
15:             break // reached stable assignment
16:           else if ($R$ != null) && !(rvector.contains($R$)) then
17:             if $R$.isHigherPriorityRule(rvector) then
18:               hazard = "Priority Inversion"
19:             end if
20:             rvector.add($R$)
21:           else if (j == $i$ - 1) then
22:             if isEmpty(rules[]) then
23:               hazard = "Hold" // may override
24:             else if !$R \in$ rvector.prefix(R) && hazard != "Priority Inversion" then
25:               hazard = "Activation"
26:             end if
27:           end if
28:         end for
29:         faultsVector.add({hazard, $S$, bitString, sequence, rvector})
30:       end for
31:     end for
32:   end for
33: end for
34: return faultsVector
```

Algorithm 17 MinimumSafeDelays

Input: *faultsVector*: vector of detected faults.

Output: *delaysVector*: {state, bitString, subPath, delay}.

```
1:  delaysVector = {}
2:  for each fault in faultsVector do
3:      path = fault.getCriticalPath()
4:      for $i$ = 0 to path.size() - 1 do
5:          subPath[] = path.getSubPath(0, $i$)
6:          delay = 0
7:          for each Variable v in path do
8:              ContextVariable cv = v.getContext()
9:              if !(cv $\in$ subPath.getContexts()) then
10:                 t = cv.getRefreshRate()
11:                 delay = max(delay, t)
12:             end if
13:         end for
14:         if delay $>$ delaysVector.get(subPath) then
15:             delaysVector.add({fault.getState(), fault.getBitString(), subPath, delay})
16:         end if
17:     end for
18: end for
19: return delaysVector
```

A.2.1 Fixing a Hazard

For a detected critical path, a simple solution to prevent its associated hazard would be to introduce a delay until all the variables of the critical path have been updated by the underlying middleware. Underestimating this delay may not eliminate the hazard, while overestimating it may make the application inefficient. In any case, in the typical situation, different sensed concrete variables have different refresh rates, and so additional commutations of a variable that already commuted may occur before any introduced delay has elapsed.

To address this problem, for each unstable assignment reachable from a given initial stable assignment in a given state, we can calculate the *minimum safe delay*, defined as the smallest interval of time starting from the time at which the first variable of the assignment commuted, after which the assignment is hazard-free. If during the unsafe period another variable commutes, then a new minimum safe delay must be recomputed for the resulting assignment. For assignments not affected by hazards this delay is zero. For other assignments, it is the maximum over all hazards from which the assignment must be protected. Algorithm 17 generates the minimum safe delays for a given set of hazards, such as those reported by Algorithm 16. For each critical *path*, Lines 3–5 select a *subPath* of the current length

i. Lines 7–13 extract the set of context variables corresponding to context that may commute in *path* but that have not commuted in *subPath*. The slowest refresh rate of the remaining context is stored in Lines 10–11. Since the same *subPath* can be obtained from multiple *path*s, Lines 14–16 store only the slowest. Engineers can use Algorithm 17's output to force waits that prevents hazards.

Bibliography

[ABM09] Aws Albarghouthi, Jorge A. Baier, and Sheila A. Mcilraith. On the use of planning technology for verification, 2009.

[AD94] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, April 1994.

[Aic03] Bernhard K. Aichernig. Contract-based testing. In *Formal Methods at the Crossroads From Panacea to Foundational Support*, volume 2757, pages 34–48. Springer -Verlag, 2003.

[Aki71] Fumio Akiyama. An example of software system debugging. In *World Computer Congress*, pages 353–359, 1971.

[All09] Open Handset Alliance. Android. `http://www.android.com/`, 2009.

[AMNT08] Marco Autili, Leonardo Mostarda, Alfredo Navarra, and Massimo Tivoli. Synthesis of decentralized and concurrent adaptors for correctly assembling distributed component-based systems. *Journal of Systems and Software*, 81(12):2210–2236, 2008.

[App09] Apple, Inc. Iphone. `http://developer.apple.com/iphone/`, 2009.

[Bar97] Valerie Barr. Applications of rule-base coverage measures to expert system evaluation. In *Proc. National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference*, pages 411–416, July 1997.

[BC04] G. Biegel and V. Cahill. A framework for developing mobile, context-aware applications. In *Proc. Second IEEE Annual Conference on Pervasive Computing and Communications*, pages 361–365, March 2004.

[Bin07] David Binkley. Source code analysis: A road map. In *2007 Future of Software Engineering*, FOSE '07, pages 104–119, Washington, DC, USA, 2007. IEEE Computer Society.

[Bry86] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transaction on Computers*, 35(8):677–691, 1986.

[Bry92] Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[BWM07] Stefan Bruning, Stephan Weissleder, and Miroslaw Malek. A fault taxonomy for service-oriented architecture. In *Proceedings of the 10th IEEE High Assurance Systems Engineering Symposium*, HASE '07, pages 367–368, Washington, DC, USA, 2007. IEEE Computer Society.

[CBS$^+$09] K. Chan, Judith Bishop, Johan Steyn, Luciano Baresi, and Sam Guinea. A fault taxonomy for web service composition. In Elisabetta Di Nitto and Matei Ripeanu, editors, *Service-Oriented Computing - ICSOC 2007 Workshops*, volume 4907 of *Lecture Notes in Computer Science*, pages 363–375. Springer Berlin / Heidelberg, 2009.

[CEM03a] L. Capra, W. Emmerich, and C. Mascolo. Carisma: Context-aware reflective middleware system for mobile applications. *IEEE Transactions on Software Engineering*, 29(10):929–945, October 2003.

[CEM03b] Licia Capra, Wolfgang Emmerich, and Cecilia Mascolo. Carisma: Context-aware reflective middleware system for mobile applications. *IEEE Transactions on Software Engineering*, 29(10):929–945, 2003.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programmming Languages and Systems*, 8(2):244–263, 1986.

[CFL03] Tarak Chaari and Augusto Celentano Frdrique Laforest. The secas project: Adaptation in context-aware pervasive information system. *International Journal of Pervasive Computing and Communications*, 3, 2003.

[CGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[CJ03] Mihai Christodorescu and Somesh Jha. Static analysis of executables to detect malicious patterns. In *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*, pages 12–12, Berkeley, CA, USA, 2003. USENIX Association.

[Cla97] Edmund Clarke. Model checking. In S. Ramesh and G Sivakumar, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 1346 of *Lecture Notes in Computer Science*, pages 54–56. Springer Berlin / Heidelberg, 1997. 10.1007/BFb0058022.

[Coh04] Haim Cohen. Buddy. http://buddy.sourceforge.net/manual/main.html, 2004.

[CSRR09] Javier Cubo, Michele Sama, Franco Raimondi, and David S. Rosenblum. A model to design and verify context-aware adaptive service composition. In *IEEE SCC*, pages 184–191, 2009.

[CwHW04] Yixin Chen, Chih wei Hsu, and Benjamin W. Wah. Sgplan: Subgoal partitioning and resolution in planning. In *Proc. of ICAPS04*, pages 30–32, 2004.

[eco] Track your mobile carbon footprint, reduce and offset it, inspire others to do the same. http://www.ecorio.org/. Last accessed October 1st 2009.

[Ede08] Stefan Edelkamp. Limits and possibilities of BDD for model checking software. In *German Conference on Artificial Intelligence*, pages 46–53, Kaiserslautern, 2008.

[EJN06] Stefan Edelkamp, Shahid Jabbar, and Mohammed Nazih. Large-scale optimal PDDL3 planning with MIPS-XXL. In *In 5th International Planning Competition Booklet (IPC06)*, 2006.

[ENDK02] Abdeslam En-Nouaary, Rachida Dssouli, and Ferhat Khendek. Timed Wp-method: Testing real-time systems. *IEEE Transactions on Software Engineering*, 28(11):1023–1038, November 2002.

[FC04] P. Fahy and S. Clarke. CASS—Middleware for mobile context-aware applications. In *Proc. MobiSys Workshop on Context Awareness*, pages 304–308, June 2004.

[Flo06] Jacqueline Floch. Theory of adaptation. Deliverable D2.2, MADAM Project, available from http://www.ist-music.eu/MUSIC/madam-project/madam-deliverables/techreportreference.2007-04-13.0451108510, 2006. Last accessed 7 March 2008.

[GL97] A. Gerevini and D. Long. Plan constraints and preferences in PDDL3: The language of the fifth international planning competition. *University of Brescia*, 1997. Technical Report.

[GPZ04] Tao Gu, Hung Keng Pung, and Da Qing Zhang. A middleware for building context-aware mobile services. In *Proc. IEEE Vehicular Technology Conference*, pages 2656– 2660, May 2004.

[Gup98] Uma G. Gupta. Automatic tools for testing expert systems. *Communications of the ACM*, 5:179–184, May 1998.

[Hau95] S. Hauck. Asynchronous design methodologies: An overview. *Proceedings of the IEEE*, 83:69–93, January 1995.

[HH91] William C. Hetzel and Bill Hetzel. *The Complete Guide to Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 2nd edition, 1991.

[HJL96] Constance L. Heitmeyer, Ralph D. Jeffords, and Bruce G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, 1996.

[HL96] Mats P.E. Heimdahl and Nancy G. Leveson. Completeness and consistency in hierarchical state-based requirements. *IEEE Transactions on Software Engineering*, 22(6):363–377, June 1996.

[HNTC99] Teruo Higashino, Akio Nakata, Kenichi Taniguchi, and Ana R. Cavalli. Generating test cases for a timed I/O automaton model. In *Proc. International Workshop on Testing Communicating Systems: Method and Applications*, pages 197–214, September 1999.

[Hol03] Gerard Holzmann. *Spin model checker, the: primer and reference manual*. Addison-Wesley Professional, first edition, 2003.

[HP00] Klaus Havelund and Thomas Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2:366–381, 2000. 10.1007/s100090050043.

[HPB$^+$09] M. A. Hanson, H. C. Powell, A. T. Barth, K. Ringgenberg, B. H. Calhoun, J. H. Aylor, and J. Lach. Body area sensor networks: Challenges and opportunities. *Computer*, 42(1):58–65, 2009.

[ICA] International conference on automated planning and scheduling (ICAPS). http://www.icaps-conference.org/, last accessed: August 2009.

[Ind09] Individual bugzilla.org contributors. Bugzilla. `http://www.bugzilla.org/`, 2009.

[Jav00] Java Community Process. Jsr-000082 java apis for bluetooth. `http://jcp.org/en/jsr/detail?id=82`, 2000.

[JV99] Rune M. Jensen and Manuela M. Veloso. Obdd-based universal planning: Specifying and solving planning problems for synchronized agents in non-deterministic domains. In *in Non-Deterministic Domains, Lecture Notes in Computer Science*, pages 213–248, 1999.

[Kar53] M. Karnaugh. The Map Method for Synthesis of Combinational Logic Circuits. *Trans. AIEE. pt. I*, 72(9):593–599, 1953.

[Kin76] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19:385–394, July 1976.

[KKKM06] Sanna Kallio, Juha Kela, Panu Korpipää, and Jani Mäntyjärvi. User independent gesture interaction for small handheld devices. *International Journal of Pattern Recognition and Artificial Intelligence*, 20(4):505–524, 2006.

[KNP02] M. Kwiatkowska, G. Norman, and D. Parker. PRISM: Probabilistic symbolic model checker. In T. Field, P. Harrison, J. Bradley, and U. Harder, editors, *Proc. 12th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS'02)*, volume 2324 of *LNCS*, pages 200–204. Springer, 2002.

[KT04] Moez Krichen and Stavros Tripakis. Black-box conformance testing for real-time systems. In *Proc. 11th International SPIN Workshop*, pages 109–126, April 2004.

[Lak03] Harri Lakkala. *Context Exchange Protocol Specification*. Nokia, 2003. Version 1.0.

[LC06] Yu Lei and Richard H. Carver. Reachability testing of concurrent programs. *IEEE Transactions on Software Engineering*, 32:382–403, 2006.

[LCT06] Heng Lu, W. K. Chan, and T. H. Tse. Testing context-aware middleware-centric programs: a data flow approach and an rfid-based experimentation. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 242–252, New York, NY, USA, 2006. ACM.

[LCT08] Heng Lu, W. K. Chan, and T. H. Tse. Testing pervasive software in the presence of context inconsistency resolution services. In *ICSE '08: Proc. 30th International Conference on Software Engineering*, pages 61–70, New York, NY, USA, 2008. ACM.

[LHS03] Brad Long, Daniel Hoffman, and Paul Strooper. Tool support for testing concurrent java components. *IEEE Transactions on Software Engineering*, 29(6):555–566, 2003.

[lif] Life360: Live confidently. http://www.life360.com/. Last accessed October 1st 2009.

[McM93] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[NO07] Robert Nilsson and Jeff Offutt. Automated testing of timeliness: A case study. In *Proc. International Workshop on Automation of Software Test*, page 11, May 2007.

[NS03] Brian Nielsen and Arne Skou. Automated test generation from timed automata. *International Journal on Software Tools for Technology Transfer*, 5(1):59–77, November 2003.

[Ope08a] Open Handset Alliance. android.location.locationmanager. `http://code.google.com/android/reference/android/location/LocationManager.html`, 2008.

[Ope08b] Open Handset Alliance. android.net.wifi.wifimanager. `http://code.google.com/android/reference/android/net/wifi/WifiManager.html`, 2008.

[Ope09] Open Moko community. Open moko. `http://wiki.openmoko.org/wiki/Main_Page`, 2009.

[Pal09] Palm. Palmos. `http://developer.palm.com/`, 2009.

[pow] Power Manager application. http://www.xphonesoftware.com/pm.html, last accessed: August 2009.

[Pro06a] Java Community Process. Jsr 179: Location api for j2me. `http://jcp.org/en/jsr/detail?id=179`, 2006. Version 1.0.

[Pro06b] Java Community Process. Jsr 256: Mobile sensor api. http://jcp.org/aboutJava/communityprocess/final/jsr256/index.html, 2006. Version 1.0.

[RC03] Anand Ranganathan and Roy H. Campbell. A middleware for context-aware agents in ubiquitous computing environments. In *Proc. ACM/IFIP/USENIX International Middleware Conference*, pages 143–161, June 2003.

[Res09] Research In Motion, Ltd. Blackberry. http://na.blackberry.com/eng/developers/, 2009.

[RMP97] Gruia-Catalin Roman, Peter J. McCann, and Jerome Y. Plun. Mobile UNITY: Reasoning and specification in mobile computing. *ACM Transactions on Software Engineering and Methodology*, 6(3):250–282, July 1997.

[RN03] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition edition, 2003.

[ROPT05] Mika Raento, Antti Oulasvirta, Renaud Petit, and Hannu Toivonen. Contextphone: A prototyping platform for context-aware mobile applications. *IEEE Pervasive Computing*, 4(2):51–59, 2005.

[RS04] Ari Koivisto Riku Suomela, EEro Rasanen. *MUPE Context Programming Manual*. Nokia, 2004. Version 1.10.

[Sam] M. Sama. CAAA verifier. http://code.google.com/p/caaaverification/.

[SDA99] Daniel Salber, Anind K. Dey, and Gregory D. Abowd. The context toolkit: Aiding the development of context-enabled applications. In *CHI '99: Proc. SIGCHI Conference on Human Factors in Computing Systems*, pages 434–441, New York, NY, USA, 1999. ACM Press.

[Sen08] Koushik Sen. Race directed random testing of concurrent programs. *SIGPLAN Not.*, 43:11–21, June 2008.

[SER+10] Michele Sama, Sebastian Elbaum, Franco Raimondi, David S. Rosenblum, and Zhimin Wang. Context-aware adaptive applications: Fault patterns and their automated identification. *IEEE Transactions on Software Engineering*, 36:644–661, 2010.

[sof] Softrace. http://www.softrace.net/. Last accessed October 1st 2009.

[SPF+06] Michele Sama, Vincenzo Pacella, Elisabetta Farella, Luca Benini, and Bruno Riccó. 3did: A low-power, low-cost hand motion capture device. In *DATE '06: Proc. Conference on Design, Automation and Test in Europe*, pages 136–141, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[sql] SQLite. http://www.sqlite.org/, last accessed: January 2010.

[SRa] M. Sama and D.S. Rosenblum. ContextNotifier. http://code.google.com/p/contextnotifier/. Last accessed 7 March 2008.

[SRb] M. Sama and D.S. Rosenblum. Testingemulator. http://code.google.com/p/testingemulator/. Last accessed 7 March 2008.

[SR07] M. Sama and D.S. Rosenblum. Contextnotifier & testingemulator: A toolkit for developing adaptive, context-aware applications. Demo session on MobiSys 2007, June 2007.

[SRWE08a] Michele Sama, David S. Rosenblum, Zhimin Wang, and Sebastian Elbaum. Model-based fault detection in context-aware adaptive applications. In *SIGSOFT '08/FSE-16: Proc. 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 261–271, New York, NY, USA, 2008. ACM.

[SRWE08b] Michele Sama, David S. Rosenblum, Zhimin Wang, and Sebastian Elbaum. Multi-layer faults in the architectures of mobile, context-aware adaptive applications: A position paper (Short Paper). In *SAM '08: Proc. 1st International Workshop on Software Architectures and Mobility*, pages 47–49, New York, NY, USA, 2008. ACM.

[SRWE10] Michele Sama, David S. Rosenblum, Zhimin Wang, and Sebastian Elbaum. Multi-layer faults in the architectures of mobile, context-aware adaptive applications. *J. Syst. Softw.*, 83:906–914, June 2010.

[SSF$^+$03] Daniel Siewiorek, Asim Smailagic, Junichi Furukawa, Neema Moraveji, Kathryn Reiger, and Jeremy Shaffer. Sensay: A context-aware mobile phone. In *Seventh IEEE International Symposium on Wearable Computers*, pages 248–249. IEEE Computer Society, 2003.

[Sun09] Sun Microsystems, Inc. *Java Mobile Edition*, 2009.

[Thi05] M. Thielscher. Reasoning robots: The art and science of programming robotic agents. *Journal of Applied Logic*, 33, 2005.

[tim] Timeriffic (automatically updates setting for Android). `http://code.google.com/p/autosettings/`.

[Two09] Two forty four a.m. LLC. Locale. `http://www.twofortyfouram.com/`, 2009.

[Ung95] Stephen H. Unger. Hazards, critical races, and metastability. *IEEE Transactions on Computers*, 44(6):754–768, June 1995.

[wer] The mobile app for nightlife. http://wertago.com/. Last accessed October 1st 2009.

[WER07a] Zhimin Wang, Sebastian Elbaum, and David S. Rosenblum. Automated generation of context-aware tests. In *Proc. International Conference on Software Engineering*, pages 406–415, May 2007.

[WER07b] Zhimin Wang, Sebastian Elbaum, and David S. Rosenblum. Automated generation of context-aware tests. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 406–415, Washington, DC, USA, 2007. IEEE Computer Society.

[WER07c] Zhimin Wang, Sebastian Elbaum, and David S. Rosenblum. Automated generation of context-aware tests. In *ICSE '07: Proc. 29th International Conference on Software Engineering*, pages 406–415, Washington, DC, USA, 2007. IEEE Computer Society.

[WGSD07] Gary Wassermann, Carl Gould, Zhendong Su, and Premkumar Devanbu. Static checking of dynamically generated queries in database applications. *ACM Transactions on Software Engineering and Methodology*, 16(4):article 14, 2007.

[Wha07] John Whaley. JavaBDD. http://javabdd.sourceforge.net/, 2007.

[XC05] Chang Xu and S. C. Cheung. Inconsistency detection and resolution for context-aware middleware support. In *Proc. Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 336–345, September 2005.

[XCC06] Chang Xu, S. C. Cheung, and W. K. Chan. Incremental consistency checking for pervasive context. In *Proc. International Conference on Software Engineering*, pages 292–301, May 2006.

[ZC06] Ji Zhang and Betty H. C. Cheng. Model-based development of dynamically adaptive software. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 371–380, New York, NY, USA, 2006. ACM.

[ZLS$^+$08] Piero Zappi, Clemens Lombriser, Thomas Stiefmeier, Elisabetta Farella, Daniel Roggen, Luca Benini, and Gerhard Trster. Activity recognition from on-body sensors: Accuracy-power trade-off by dynamic sensor selection. In Roberto Verdone, editor, *EWSN*, volume 4913 of *Lecture Notes in Computer Science*, pages 17–33. Springer, 2008.