UNIVERSIDADE FEDERAL DO CEARÁ

CENTRO DE CIÊNCIAS E TECNOLOGIA

DEPARTAMENTO DE COMPUTAÇÃO

PROGRAMA DE MESTRADO E DOUTORADO EM CIÊNCIA DA COMPUTAÇÃO

DOUTORADO EM CIÊNCIA DA COMPUTAÇÃO

ISMAYLE DE SOUSA SANTOS

TESTDAS: TESTING METHOD FOR DYNAMICALLY ADAPTIVE SYSTEMS

FORTALEZA

2017

ISMAYLE DE SOUSA SANTOS

TESTDAS: TESTING METHOD FOR DYNAMICALLY ADAPTIVE SYSTEMS

Tese apresentada ao Curso de Doutorado em Ciência da Computação do Programa de Mestrado e Doutorado em Ciência da Computação da Universidade Federal do Ceará, como requisito parcial à obtenção do título de doutor em Ciência da Computação. Área de Concentração: Engenharia de Software

Orientadora: Prof. Dra. Rossana Maria de Castro Andrade

Co-Orientador: Prof. Dr. Pedro de Alcântara dos Santos Neto

FORTALEZA

2017

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Universitária
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

S235t Santos, Ismayle de Sousa.
TestDAS: Testing Method for Dynamically Adaptive Systems / Ismayle de Sousa Santos. – 2017.
183 f. : il. color.

Tese (doutorado) – Universidade Federal do Ceará, Centro de Ciências, Programa de Pós-Graduação em Ciência da Computação , Fortaleza, 2017.
Orientação: Profa. Dra. Rossana Maria de Castro Andrade.
Coorientação: Prof. Dr. Pedro de Alcântara dos Santos Neto.

1. Software Testing. 2. Model Checking. 3. Dynamically Adaptive System. 4. Dynamic Software Product Line. I. Título.

CDD 005

ISMAYLE DE SOUSA SANTOS

TESTDAS: TESTING METHOD FOR DYNAMICALLY ADAPTIVE SYSTEMS

Tese apresentada ao Curso de Doutorado em Ciência da Computação do Programa de Mestrado e Doutorado em Ciência da Computação da Universidade Federal do Ceará, como requisito parcial à obtenção do título de doutor em Ciência da Computação. Área de Concentração: Engenharia de Software

Aprovada em: 23 de novembro de 2017

BANCA EXAMINADORA

Prof. Dra. Rossana Maria de Castro Andrade (Orientadora)
Universidade Federal do Ceará (UFC)

Prof. Dr. Pedro de Alcântara dos Santos Neto (Co-Orientador)
Universidade Federal do Piauí (UFPI)

Prof. Dr. Guilherme Horta Travassos
Universidade Federal do Rio de Janeiro (UFRJ)

Prof. Dr. Eduardo Santana de Almeida
Universidade Federal da Bahia (UFBA)

Prof. Dr. Lincoln Souza Rocha
Universidade Federal do Ceará (UFC)

I dedicate this thesis to my parents, Ismael e Cláudia, my parents-in-law, Luiz e Conceição, and my dear wife, Paulinha.

ACKNOWLEDGEMENTS

Foremost, I would like to thank our God for all good things that have happened. It was a long journey and without God's light and peace I have not gotten that far.

I would like to thank my parents, Ismael and Claudia, and my brother, Kádson. Thank you for all your love and unconditional support. I acknowledge and thank immensely all effort and immeasurable sacrifice you have made to allow me to be here.

I would also like to thank my sweet wife, Paulinha. You are a little angel that I got as a gift. Thank you for being such a wonderful person and for your unconditional support throughout this journey.

I also thank my parents-in-law, Luiz and Conceição, who supported and encouraged me during all these years. Thank you for sharing this dream with me.

Thanks to my academic and life advisor, Prof. Rossana Andrade and Prof. Pedro de Alcântara. It was an honor to work with you. I learned a lot over the years. Thank you very much for all the encouragement, support, teachings, and opportunities. Having you by my side made all the difference.

I want to give thanks to Prof. Eduardo Santana, Guilherme Travassos, Lincoln Rocha and Rafael Capilla for generously sharing your time and knowledge.

Thanks to all my friends from the GREat lab. Thanks for the welcome, the fun times and the support in the difficult moments. You are great!!

And lastly, I thank CAPES and CNPq for the financial support during my years of research.

“In God we trust, everything else we test”

(Anonymous author)

ABSTRACT

The adaptive behavior of Dynamically Adaptive Systems (DAS), such as Dynamic Software Product Lines (DSPLs), is typically designed using adaptation rules, which are context-triggered actions responsible for features activation and deactivation at runtime. This kind of software should have a correct specification at design time and should be tested to avoid unexpected behavior such as an undesired product configuration at runtime. The use of context and the large number of configurations are challenges related to DAS verification and validation. Therefore, methods and tools supporting these activities are needed to ensure the quality of adaptive systems. The literature addresses different aspects of DAS testing, but few work deals with changes in the software features configuration, and they did not focus on the adaptation rules during the adaptive mechanism testing. Also, there is a lack of formalism to model DAS that allows to reason on the actions triggered by adaptation rules over the DAS features. The focus on the adaptation rules is important because they define the adaptation logic and, thus, they are a potential source of design faults and adaptation failures at runtime. In this thesis, a method called TestDAS is proposed to address these gaps. It involves the model checking approach to identify faults in the adaptation rules design, and the generation of tests for validating the adaptive behavior of DAS. The method is based on a model of the adaptive behavior, called Dynamic Feature Transition System (DFTS), which specifies the DAS configurations and the context changes. Moreover, the TestDAS tool is implemented to support the TestDAS use, and a library called CONTroL is developed to support the test execution. The evaluation of TestDAS is performed using: a mutant analysis to evaluate the effectiveness of the model checking approach in the identification of design faults in DAS; a controlled experiment to compare tests generated by TestDAS with tests specified based on the tester's experience; and an observational study to assess the feasibility of using the developed tools during the TestDAS activities. The results of the effectiveness evaluation show evidence that TestDAS helps in the identification of faults related to adaptation rules design. The experiment, in turn, provides evidence that TestDAS generates more tests and provides a better coverage of the DAS adaptive behavior than experience-based testing. Lastly, the observational study shows that the TestDAS tool and CONTroL can support the DAS testing and model checking.

Keywords: Software Testing. Model Checking. Dynamically Adaptive System. Dynamic Software Product Line

RESUMO

O comportamento adaptativo de Sistemas Dinamicamente Adaptáveis (DAS, acrônimo para *Dynamically Adaptive Systems*), tais como Linhas de Produto de Software Dinâmicas, é tipicamente especificado por meio de regras de adaptação, que definem ações sensíveis ao contexto responsáveis pela ativação e desativação de *features* em tempo de execução. Esse tipo de software tem que ter uma correta especificação em tempo de projeto e deve ser adequadamente testado para evitar comportamentos não esperados, como configurações de produto não desejadas em tempo de execução. Nesse cenário, o uso de informações de contexto e o grande número de configurações possíveis são desafios relacionados a verificação e validação de DAS. Dessa forma, métodos e ferramentas suportando essas atividades são necessários para garantir a qualidade de sistemas adaptativos. Na literatura, existem trabalhos abordando diferentes aspectos do teste de DAS, mas poucos deles lidam com mudanças na configuração de *features* do software, e eles não focam nas regras de adaptação durante o teste do mecanismo de adaptação. Além disso, existe uma carência de formalismos para especificar DAS que permitam raciocinar sobre as ações disparadas pelas regras de adaptação. O foco nas regras de adaptação é importante, porque elas definem a lógica de adaptação e, assim, são uma fonte potencial de faltas de *design* e de falhas que podem ocorrer em tempo de execução. Sendo assim, nesta tese é proposto um método de testes de DAS, chamado TestDAS, que envolve tanto uma abordagem de *model checking* (verificação de modelos) para identificar faltas de *design*, quanto a geração de testes para validar o comportamento adaptativo. Esse método é baseado em um modelo do comportamento adaptativo, chamado de Dynamic Feature Transition System (DFTS), que especifica as configurações do DAS e as mudanças de contexto. Adicionalmente foram implementadas a TestDAS tool, para apoiar o uso do TestDAS, e a biblioteca CONTroL, para auxiliar a execução dos testes. A avaliação do TestDAS, por sua vez, foi feita utilizando: uma análise de mutantes para avaliar a efetividade da abordagem de *model checking* na identificação de faltas no DAS; um experimento controlado para comparar os testes gerados pelo TestDAS e os testes criados com base na experiência de testadores; e uma prova de conceito para avaliar a viabilidade do uso das ferramentas desenvolvidas durante as atividades do TestDAS. Os resultados da avaliação da análise de mutantes indicam evidências de que o TestDAS ajuda na identificação de faltas de *design* presentes nas regras de adaptação. O experimento controlado fornece evidências de que o TestDAS gera mais testes e provê uma melhor cobertura do comportamento adaptativo de DAS do que o teste baseado na experiência dos testadores. Finalmente, a prova de conceito

confirma que as ferramentas desenvolvidas, TestDAS tool e CONTroL, podem auxiliar no teste e no *model checking* de sistemas dinamicamente adaptáveis.

Palavras-chave: Teste de Software. Verificação de Modelos. Sistemas Dinamicamente Adaptáveis. Linha de Produto de Software Dinâmica

LIST OF FIGURES

LIST OF TABLES

LISTA DE ABREVIATURAS E SIGLAS

C-KS	Context Kripke Structure
CAAS	Context-Aware Adaptive Software
CONTroL	*CON*text variability based software *T*esting *L*ibrary
DAS	Dynamically Adaptive System
DFTS	Dynamic Feature Transition System
DSL	Domain Specific Language
DSPL	Dynamic Software Product Line
JSON	JavaScript Object Notation
SBSE	Search-Based Software Engineering
SPL	Software Product Line
TestDAS	Testing method for Dynamic Adaptive System

CONTENTS

1 INTRODUCTION

This thesis presents a method for testing Dynamically Adaptive System (DAS). Such method involves a model checking approach and the generation of test cases focused on the adaptive behavior. Tools are also built to support the use of this method.

The current chapter introduces this thesis and is organized as follows. Section 1.1 describes the research context, whereas Section 1.2 presents the motivation of this work and the problem addressed. Section 1.3 presents the hypothesis and research questions. Next, Section 1.4 introduces the thesis goals and main contributions. After that, Section 1.5 presents the research methodology followed during this thesis work. Finally, Section 1.6 presents the organization of this thesis.

1.1 Contextualization

The concept of Software Product Line (SPL) emerged to mitigate the growing need of the software industry to maximize the software reuse aiming to obtain, among other things, better productivity. According to Northrop (2002), from the Software Engineering Institute (SEI), an SPL is "*a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way*".

Thus, the SPL engineering is about producing families of similar systems that have *common features* that are present in all SPL products and *variant features*, which do not appear in all the products of the SPL (BENAVIDES *et al.*, 2010). With the SPL strategy, the industry had obtained several benefits (NORTHROP; CLEMENTS, 2007; FOGDAL *et al.*, 2016): increased software quality, reduced cost, decreased time to market and improved productivity. However, this strategy only deals with the static variability, in which variants are set during the development cycle (CAPILLA *et al.*, 2014b). So, it cannot cope with the dynamic reconfiguration of context-aware systems that adapts at runtime such as cyber-physical systems (MUCCINI *et al.*, 2016).

Dynamic Software Product Line (DSPL) can be seen as an extension of the concept of conventional Software Product Lines (SPLs), enabling the generation of software variants at runtime (BENCOMO *et al.*, 2012b). Thus, the differential of DSPLs is to be deal with the context variability that is the variability of the environment in which the system resides (HARTMANN; TREW, 2008).

DSPLs provide a way to build software able to adapt dynamically to fluctuations in user needs and resource constraints (HALLSTEINSEN *et al.*, 2008). Thus, the DSPL engineering has been used to provide support for developing of dynamically adaptive systems, because it copes with both the system variability and the context variability (CAPILLA *et al.*, 2014b). On the other hand, since a DAS self-adapts according to the context information gathered from the surrounding environment, it can be considered a DSPL in which variabilities are bound at runtime (BENCOMO *et al.*, 2008). In this way, each DAS configuration can be considered as a product of the DAS product line.

The DSPL's adaptive behavior is typically designed using adaptation rules, which are context-triggered actions responsible for the software adaptation. Thus, to achieve the software reconfiguration, DSPLs support the activation and deactivation of system features at runtime (CAPILLA *et al.*, 2014a).

The context variability and runtime reconfiguration turns the DSPL development and testing more complex compared with traditional applications. One of the challenges is to test the adaptations provided by context variability to avoid unexpected behaviors at runtime (*e.g.*, a product reconfiguration not performed when it should be).

1.2 Motivation

Dynamic systems (e.g., autonomous systems and ubiquitous systems) exploits contextual information to adapt at runtime, according to changes in their surrounding environment. Such dynamic behavior is typically designed using adaptation rules, and many of the pervasive and software-intensive systems that use context properties are based on a DSPL approach (CAPILLA *et al.*, 2014b).

So, to avoid unexpected behavior is important to ensure the correct implementation of the adaptation rules. Software testing can be performed aiming to identify failures in the DSPL adaptations. Testing is the process of executing a program with the intent of finding errors (MYERS *et al.*, 2011). Thus, aiming to test the DSPL adaptive behavior, the tester needs to assess the adaptation rules effects while running the DAS.

There are studies concerning the test of Adaptive Systems, Software Product Lines and Dynamic Software Product Lines, as discussed in Chapter 3. These studies address different aspects of the software testing and can help in the DAS testing activity. However, usually, they cover the context space but do not focus on the adaptation rules effects. Then, they do not ensure

the coverage of some scenarios that can raise failures at runtime such as the interleaving of adaptation rules or specific sequences of triggered adaptation rules.

Therefore, there is a lack of testing method focused on the actions (*i.e.*, activation and deactivation of system features) of adaptation rules to validate the DSPL adaptations triggered by context changes.

In order to support the testing activity, the DSPL feature model, which is enriched with context information and adaptation rules, can be used as source of information to design test cases. However, to use the DSPL feature model for testing purpose, it is need to ensure its correctness to avoid false-negative and false-positive failures.

In the SPL, there are several studies dealing with the consistency checking of feature models (ZHANG *et al.*, 2011; MARINHO, 2012; MARINHO *et al.*, 2012; LESTA *et al.*, 2015). The work of Marinho *et al.* (2012), for instance, uses a formal specification, built based on First-Order Logic, to verify well-formedness (*i.e.*, the conformance with constraints of the underlying formal specification) and consistency of feature models against a set of predefined properties. Despite considering Context-Aware Software Product Lines that have adaptation rules, the verification proposed by Marinho *et al.* (2012) do not take into account the context states where the product is deployed. Furthermore, the DSPL validation requires capabilities for checking temporal properties of reconfiguration processes (LOCHAU *et al.*, 2015).

For checking temporal properties, the model checking (CLARKE JR. *et al.*, 1999) is an automated verification technique that has been used in several application domains. In a nutshell, it requires a model of the system behavior and a property, and systematically checks whether the given model satisfies this property or not.

Thus, a formalism to model the DSPL adaptive behavior is needed to reason about the effect of the adaptation rules over the DSPL product configurations. This formalism should provide answers to questions like: *Are all the product configurations in conformance with the DSPL rules?* and *Does the interleaving of adaptation rules triggered at the same time have the expected effect?*

To answer this kind of questions, the DSPL behavior model must be under the adaptation perspective, specifying both context changes and possible product configurations in response to the effects of the triggered adaptation rules. In the literature, there are studies proposing a model checking approach for SPL and self-adaptive systems (see Chapter 3), but only a few work concerning the DSPL behavior (MUSCHEVICI *et al.*, 2010; CORDY *et al.*,

2013; LOCHAU *et al.*, 2015). The latter focuses on modeling the software state (e.g. "ready") together with the required context and software configuration, not giving support to verify the effect of the adaptation rules.

Thus, there is a lack of a formalism for allowing the model checking focused on the adaptation rules and their effects over the DSPL products. Also, it is worth noting that even if the DSPL design is correct, its source code can have faults related to the adaptive behavior that can raise failures at runtime. So, the testing is important to ensure that the adaptive behavior of the products is correct, according to the DSPL design.

1.3 Hypothesis and Research Questions

In DAS, the adaptation rules specify how the system should adapt according to context changes. In particular, considering the DSPL engineering, these rules define which features should be activated and deactivated at runtime. In this case, to verify and validate the behavior defined by the adaptation rules should be a first-class concern to avoid unexpected DAS failures at runtime. Thus, this research is conducted to investigate the following research hypothesis:

Research Hypothesis: A DAS testing method using a model to specify the DAS features configuration based on adaptation rules provides better coverage of the adaptive behavior and supports the identification of faults in the adaptation rules design.

From this hypothesis, the following Research Questions (RQ) were proposed:

- **RQ1** How to model the DAS adaptive behavior to support the identification of behavioral faults in the adaptation rules? *Rationale: This question aims to find out an adequate formalism to model the dynamic adaptations over context changes allowing to use model checking to identify design faults. The identification of these faults is important to ensure that the feature model and its adaptation rules are correctly specified and, thus, they can be used to support the testing of the adaptive behavior.*
- **RQ2**: Which common behavioral properties related to the DAS adaptation mechanism should be satisfied? *Rationale: This question intends to identify behavioral properties that can be verified through model checking and whose violations are related to common faults in the specification of adaptation rules. Besides that, these properties capture the*

semantics of behavior patterns that need to be tested in DAS.

- **RQ3**: How to design test cases to achieve a better coverage of the DAS adaptive behavior? *Rationale: Even if the design is correct, the DAS source code can have faults related to the adaptive behavior that can raise failures at runtime. Thus, this question aims to find out an approach to DAS testing to assess whether its adaptive behavior occurs as expected or not.*

1.4 Research Goal and Main Contributions

Many studies reported approaches to ensure the quality of Dynamically Adaptive Systems. However, since these approaches are not focused on the actions triggered by the adaptation rules, they can miss faults and failures related to the adaptive behavior.

In this sense, the goal of this thesis is *to propose a method for testing the DAS adaptive behavior focused on the effects of the adaptation rules. This method also involves a model checking approach to support the identification of design faults in the DAS feature model*. The latter is important to ensure a correct feature model that can be used as source of information to generate tests. Thus, it is the purpose of this thesis to propose a solution to ensure the quality of the DAS adaptive behavior from the design to the implementation.

The main expected contributions of this work are summarized as follows:

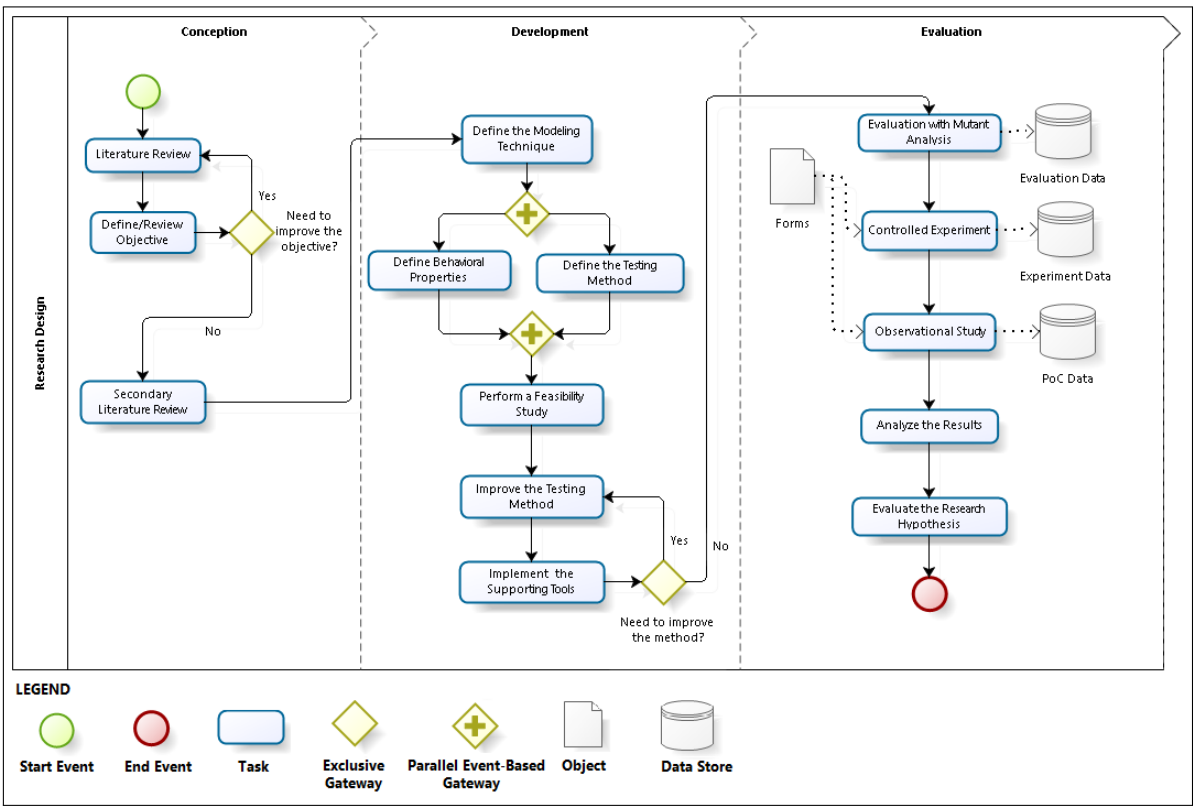
- A method for testing the DAS adaptive behavior, which generates tests based on a set of coverage criteria proposed. This method provides an approach to model checking the DAS design. Also, it should includes:
  - A model to specify the DAS adaptive behavior based on the features (de)activation and the context changes. This model is used by the method proposed to support the DAS model checking and testing;
  - A set of behavioral properties that DAS should satisfy. Such properties are proposed for helping the software engineer in the identification of faults in the DAS design;
- Supporting tools that supports the DAS model checking, as well as the generation and execution of tests following the method proposed in this thesis;

It is important to mention that is out of the scope of this thesis to address the context changes during the tests execution, propose a variability modeling technique and propose a method for testing the DAS functionality.

1.5 Research Methodology

The research methodology of this thesis is defined based on objectives presented in Section 1.4. In a nutshell, this research is organized into three main phases: (i) *Conception*, in which the research problem and questions are refined based on literature review; (ii) *Development*, in which a solution is proposed to the problem addressed; and (iii) *Evaluation*, in which the developed solution is evaluated. Each phase includes a set of activities, as depicted in Figure 1.

Figure 1 – Research Methodology



Source – the author.

In the *Conception* phase, the first activity is the `Literature Review`, which comprises the search for papers related to DAS model checking and testing. It is worth noting that this search, as presented in Chapter 3, involves the DSPL domain, as well as the Context-Aware (Adaptive) Systems and Software Product Line domains. In this activity, a non-systematic review is performed using online databases like Scopus[1], Web of Science[2], IEEE Xplorer[3] and ACM DL[4]. Besides, this review involved the analyses of the secondary studies available in the literature

[1] http://scopus.com/
[2] https://www.webofknowledge.com/
[3] http://ieeexplore.ieee.org
[4] http://dl.acm.org/

that are related to the investigated topics (NETO *et al.*, 2011; ENGSTRöM; RUNESON, 2011; WEYNS *et al.*, 2012; BENDUHN *et al.*, 2015; LOPEZ-HERREJON *et al.*, 2015; MATALONGA *et al.*, 2017).

Still in the *Conception* phase, the second activity is `Define/Review Objective`, which involves the definition of the research hypothesis and questions based on the results of the literature review. The third activity in this phase is the `Secondary Literature Review`, which is performed in the form of a quasi-Systematic Literature Review (qSLR) (TRAVASSOS *et al.*, 2008) about test case design for context-aware systems testing. This secondary study is an activity of the CAcTUS project[5], and it is performed on the following databases: Scopus[6] and Web of Science[7].

In the *Development* phase, the first activity is the definition of a formalism to specify the DAS adaptive behavior (`Define the Modeling Technique` in Figure 1). Then, the activities `Define Behavioral Properties` and `Define the Testing Method` are performed at the same time. During the `Define Behavioral Properties` activity, common behavioral properties are specified based on the DSPL concepts and behavioral properties of related domains, like Context-Aware System domain. These properties define expected behavior that should be satisfied by a DAS. The other activity, in turn, is related to the definition of a method to DAS testing. This method has two main goals: (i) to support the verification of the defined properties avoiding that design faults disturb the testing activity; and (ii) to support the DAS testing based on the behavioral properties.

To assess the initial results of the proposal, the fourth activity in the *Development* phase is to `Perform a Feasibility Study` with focus on the verification using the formalism defined. In this study, the idea is to use the model proposed to check behavioral properties in a DSPL, aiming to assess the applicability of this model in the identification of design faults.

Still in the *Development* phase, the fifth activity is to `Improve the Testing Method` considering the results of the feasibility study. Then, the last activity in this phase is `Implement the Supporting Tools`, in which a tool and a library are implemented to support the use of the testing method proposed.

In the *Evaluation* phase, the first activity is `Evaluation with Mutant Analysis`. In this activity, mutant analysis (JIA; HARMAN, 2011) is used to evaluate if the method is

[5] http://lens.cos.ufrj.br/cactus/
[6] http://scopus.com/
[7] https://www.webofknowledge.com/

effective to identify design faults in a DSPL. Next, the testing method is evaluated through a controlled experiment (`Controlled Experiment` activity). The guideline of Wohlin *et al.* (2014) is used to guide the controlled experiment. The third activity is the `Observational Study` that is about a feasibility study of the supporting tools. In both the experiment and the observational study, online forms (`Forms`) are used to identify the subject's profile and collect their feedback about the performed activities. The data from all three evaluations are archived in online repositories. The last two activities of the *Evaluation* phase are `Analyze the Results` and `Evaluate the Research Hypothesis`, in which the results are used to answer the investigated research hypotheses.

1.6 Structure of the Thesis

This chapter introduced this thesis by describing the motivation and goals of this work, and the research questions that it aims to answer. Also, this chapter described the research hypothesis, and presented the research methodology.

Besides this Introduction Chapter, this thesis is organized in the following chapters:

- **Chapter 2 (Background)** outlines the mains concepts related to this thesis proposal: context variability, dynamic software product line, model checking and software testing. This chapter also describes the formalism used throughout this thesis.
- **Chapter 3 (Related Work)** compares the proposal of this thesis with studies found in the literature addressing the testing and model checking of SPL, DSPL, and dynamic systems.
- **Chapter 4 (TestDAS)** presents in details the DAS testing method proposed in this thesis, as well as the tool implemented.
- **Chapter 5 (Evaluation)** describes the assessments of the method proposed through experiments focused on the model checking and the test criteria proposed.
- **Chapter 6 (Conclusions)** summarizes the achieved contributions and discusses some future research directions.

This thesis also present seven appendices with the following subjects: Appendix A summarizes the Promela grammar; Appendix B presents the codes used in the feasibility study; Appendix C describes the class diagram for the context-aware Feature models used as input to the tool implemented; Appendix D presents the list of generated mutants; Appendix E outlines the experiment instrumentation; Appendix F describes the instrumentation of the observational study; and Appendix G presents other published papers during the thesis work period.

2 BACKGROUND

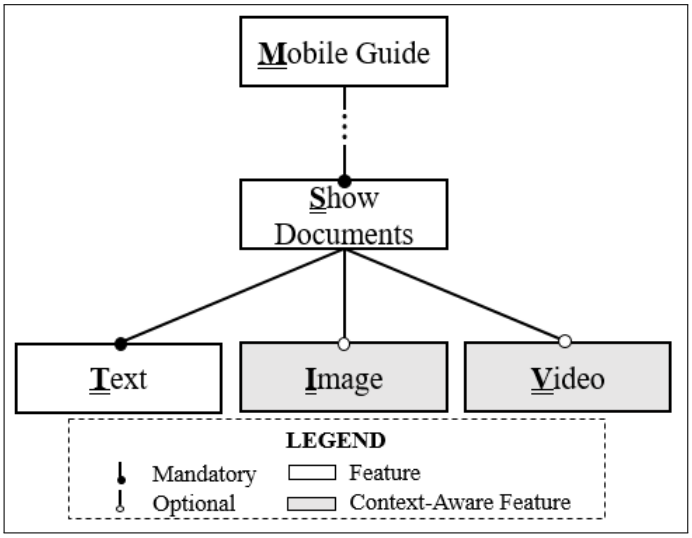
This chapter describes the background of this thesis. It presents an overview of context-awareness in DAS, Software Verification and Software Testing.

The organization of this chapter is as follows. Section 2.1 introduces the running example that is used throughout this thesis. Section 2.2 defines what is context, and introduces variability modeling techniques and the DSPL paradigm. Section 2.3 presents the main concepts related to software verification, focusing on the model checking technique. Section 2.4 introduces software testing, the testing types, the main test case design techniques, and the challenges related to DAS testing. Finally, Section 2.5 concludes this chapter.

2.1 Running Example

This section presents the DSPL used as running example along this thesis. The Mobile Visit Guides is a dynamic SPL for the domain of mobile and context-aware visit guides. This DSPL is a work product of the MobiLine Project (MARINHO *et al.*, 2013). Figure 2 shows a small part of the feature model of the Mobile Visit Guides DSPL. The complete model is available at the MobiLine project site[1].

Figure 2 – A small part of the Mobile Visit Guides DSPL feature model



Source – adapted from Marinho *et al.* (2013).

The feature model depicted in Figure 2 has five features: *Mobile Guide*, *Show Documents*, *Text*, *Image*, and *Video*. In this model, there is one variation point, which is an

[1] http://mobiline.great.ufc.br/

or-group that can be bound at runtime. This variation point concerns how the information, about the place where the visitor is, can be displayed. All products of the Mobile Visit Guides DSPL can display textual information (i.e., all have the feature *Text*) and, optionally, they can display images and videos when the features *Image* and *Video* are respectively present in the product.

The features *Image* and *Video* can be activated (*status on*) and deactivated (*status off*) during the product execution according to both the battery charge level and the power source connection (context information). In this thesis, these features are called "context-aware features" because their state depends on the current context. For instance, given that the smartphone is not connected to a power source; if the battery level is low, then only textual information is available. Otherwise, if the battery level is normal, then the visitor can only access textual information and images; and if the battery charge is full, all files (texts, images, and videos) are available. Table 1 summarizes all adaptation rules for the variation point *Show Documents*.

Table 1 – Adaptation Rules of the Running Example

Rule ID	Context Condition	Image	Video
AR01	isBtLow $\wedge \neg$ hasPwSrc	off	off
AR02	isBtLow $\wedge$ hasPwSrc	on	off
AR03	isBtNormal $\wedge \neg$ hasPwSrc	on	off
AR04	isBtNormal $\wedge$ hasPwSrc	on	on
AR05	isBtFull	on	on

Source – the author.

2.2 Context Awareness in DAS

There are several definitions of context (BROWN *et al.*, 1997; DEY, 2001; MOSTEFAOUI *et al.*, 2004; VIANA *et al.*, 2011). One of the most used is the definition of context as any information that can be used to characterize the situation of an entity (person, place, or object) (DEY, 2001). Santos *et al.* (2017) evolved this definition with the purpose of enhancing the importance of the relationship between actors and computers:

> Context is any piece of information that may be used to characterize the situation of an entity (logical and physical objects present in the systems environment) and its relations that are relevant for the actor-computer interaction. (SANTOS *et al.*, 2017, p. 3)

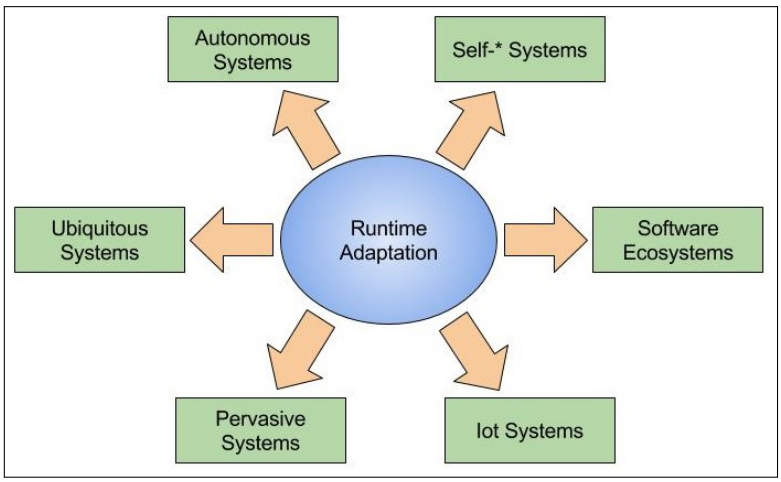
Thus, given the running example described in Section 2.1, the context information used by the system to adapt itself are the battery charge level and whether the device is connected

or not to a power source.

Several application domains have been used contextual properties to drive dynamic runtime reconfiguration with little or no human intervention (CAPILLA *et al.*, 2014b). This reconfiguration, realized by changes in the features of the system, is referred to as *runtime adaptation* (BASHARI *et al.*, 2017). Hence, a Dynamically Adaptive System (DAS) is a software system with enabled runtime adaptation (ALVES *et al.*, 2009).

Figure 3 depicts the main application domains that require a context-aware runtime adaptation. Self-* systems include, for example, self-healing and self-adaptive systems, which can modify their behavior and/or structure in response to their goals and their perception of the environment (LEMOS *et al.*, 2013). Concerning autonomous systems, they should be aware of its internal state and current external conditions, to detect changing circumstances and adapt accordingly (DOBSON *et al.*, 2010). Ubiquitous systems and Pervasive Systems are related to computational services available transparently, anytime and anywhere to the people in such a way that computer is no longer visible (SPÍNOLA; TRAVASSOS, 2012; ROCHA *et al.*, 2011). A software ecosystem can be defined as a collection of software projects composing a common technological platform in which a set of actors interacts (MANIKAS; HANSEN, 2013). The systems in the Internet of Things (IoT), in turn, allow people and things to be connected anytime, anyplace, with anything and anyone (VERMESAN *et al.*, 2011).

Figure 3 – Systems demanding runtime adaptation



Source – the author.

With the increasing of the complexity in Dynamically Adaptive Systems, the notion of “context variability” was introduced by Hartmann and Trew (HARTMANN; TREW, 2008) for modeling the context variants. Thus, context variability concerns the context-aware software

variability (MENS *et al.*, 2016). In this way, The Dynamic Software Product line paradigm has been identified as a promising approach for developing DAS, because it copes with both the system variability and the context variability. In fact, a Dynamically Adaptive System itself is seen as a DSPL [2] (BENCOMO *et al.*, 2008).

The following subsections present more details about the context variability modeling (Subsection 2.2.1) and the DSPL paradigm (Subsection 2.2.2). Besides that, Subsection 2.2.3 discusses the Context Kripke Structure (C-KS) (ROCHA; ANDRADE, 2012a), which is the context variation model used in this thesis.

2.2.1 Context Variability Modelling

The variability of a software system can be specified with the notion of features. Features are the attributes of a system that directly affect end-users (KANG *et al.*, 1990). The context variability, in turn, is used to model those context features intended to be activated or deactivated at runtime (CAPILLA *et al.*, 2014b).

A context feature is any feature that represents, uses, or manages data or knowledge coming from the surrounding context (MENS *et al.*, 2016). On the other hand, non-context features are chosen based on a static selection of features (MENS *et al.*, 2016). Thus, the context features represent the variants of context information relevant to the system, while the non-context features model the software functionality.

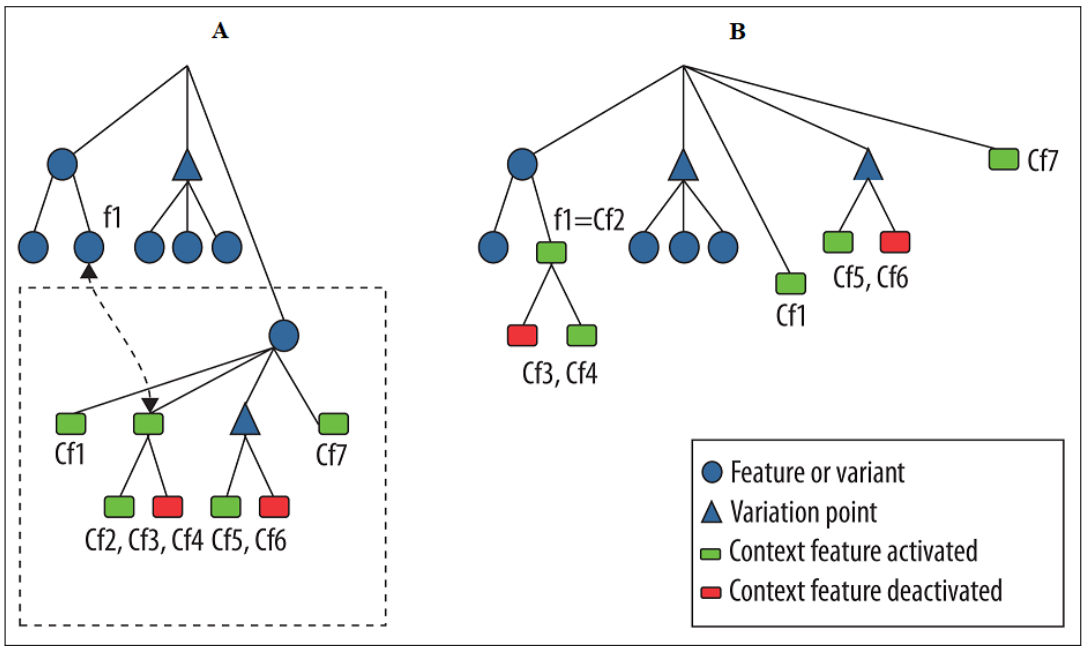
The variability modeling of systems with context-aware runtime adaptation involves modeling both context and non-context features, as well as the dependencies among them. The model language widely used to manage the system variability is the feature model (KANG *et al.*, 1990). This model represents through a hierarchical structure a set of features and the relationship among them. Currently, there are several types of feature model, varying from basic feature models to extended versions with extra information represented by feature attributes (BENAVIDES *et al.*, 2010).

Based on the feature model, there are two main approaches, which are depicted in Figure 4, to model the context. In the strategy **A** (left side of Figure 4), the feature model includes a branch in which the engineer models context features separately from non-context features (CAPILLA *et al.*, 2014b). In the strategy **B** (right side of Figure 4), the context and non-context features are modeled under the same model (CAPILLA *et al.*, 2014b). In both

[2] In this thesis, DAS and DSPL refer to a software system with context-aware adaptations at runtime

modeling strategies, the context variability model is linked to the system feature model. This link between context features and system features facilitates relating the different context conditions to an appropriate feature configuration (BASHARI *et al.*, 2017).

Figure 4 – Strategies (A and B) for Context Variability Modeling



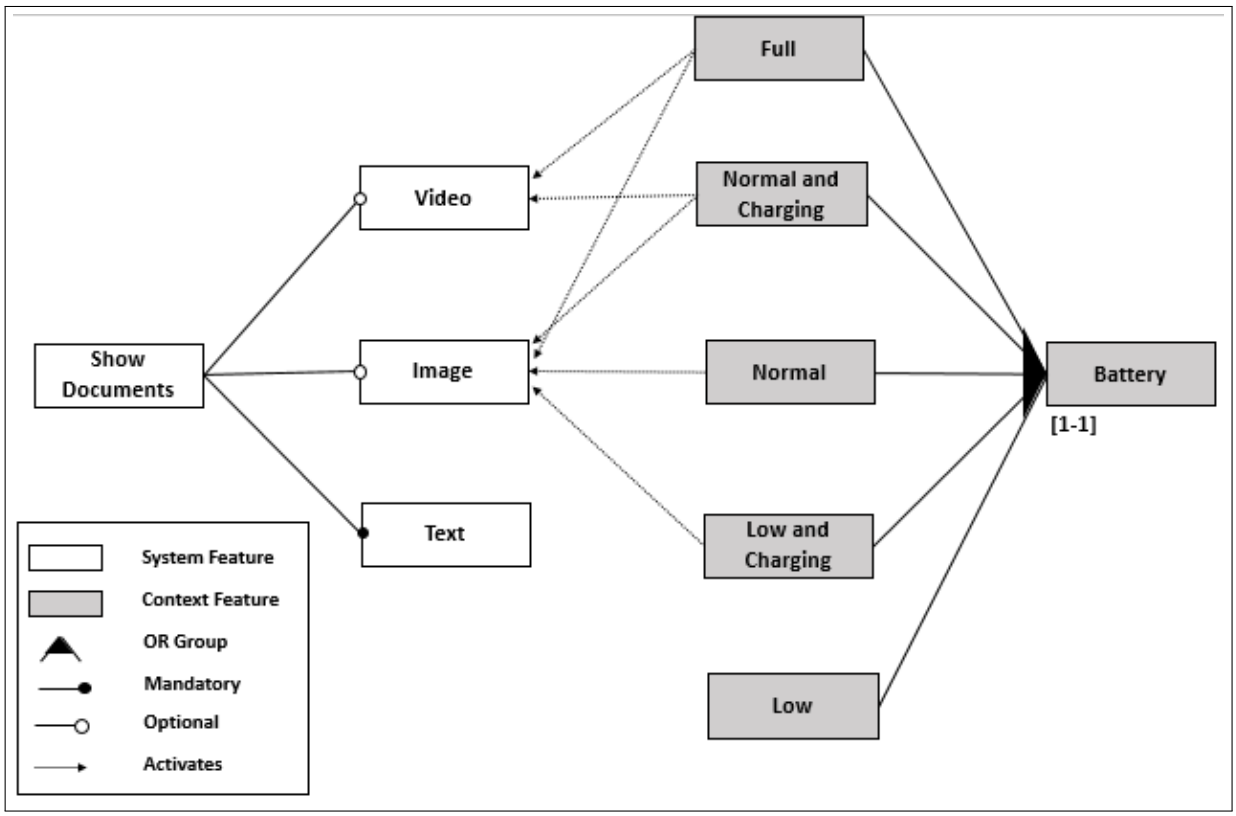
Source – adapted from Capilla *et al.* (2014b).

It is worth noting that each approach has advantages and drawbacks. For instance, the strategy **A** is more reusable when the model contains several context features, while the strategy **B** simplifies the model and reduces the number of dependencies among context and non-context features (CAPILLA *et al.*, 2014b).

Figure 5 presents the context-aware feature model (SALLER *et al.*, 2013) for the running example introduced in Section 2.1. This model follows the strategy **A** and, thus, it has two branches: one for the system features (`Video`, `Image`, and `Text`) and another for the context features (`Full`, `NormalAndCharging`, `Normal`, `LowAndCharging`, and `Low`).

Thus, the feature model of Figure 5 specifies that based on the currently active context features, related to the Battery context, the system is reconfigured by the activation of a set of system features. For instance, if the context feature `Normal` is active (*i.e.*, the current battery charge level is at *Normal* state), then just the system features `Image` and `Text` will be actives. Note that the feature `Text` is always active since it is a mandatory feature.

Figure 5 – Example of Context-Aware Feature Model



Source – the author.

2.2.2 Context Variability Management with the DSPL Engineering

According to Northrop (2002) a Software Product Line (SPL) "is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way". Therefore, the SPL strategy allows a systematic and planned reuse, providing benefits like increased quality and best time-to-market (ALMEIDA *et al.*, 2007)

Several organizations have successfully applied the SPL strategy (SEI, Last Access in Nov. 2017, 2017). However, in domains like ubiquitous computing and robotic, the software has become more complex with variation in both requirements and resource constraints (HALL-STEINSEN *et al.*, 2008). In this scenario, the SPL strategy can not handle the changes in the environment at runtime, because once the product is generated from the SPL, it could no longer be changed.

To address this limitation, emerged the Dynamic Software Product Lines (DSPL) that allow the generation of software variants at runtime (BENCOMO *et al.*, 2012b). In this way, the DSPL paradigm develops software able to adapt to changes in requirements and resources

(HALLSTEINSEN *et al.*, 2008). To allow this dynamic behavior, a DSPL usually has the following properties (CAPILLA *et al.*, 2014a):

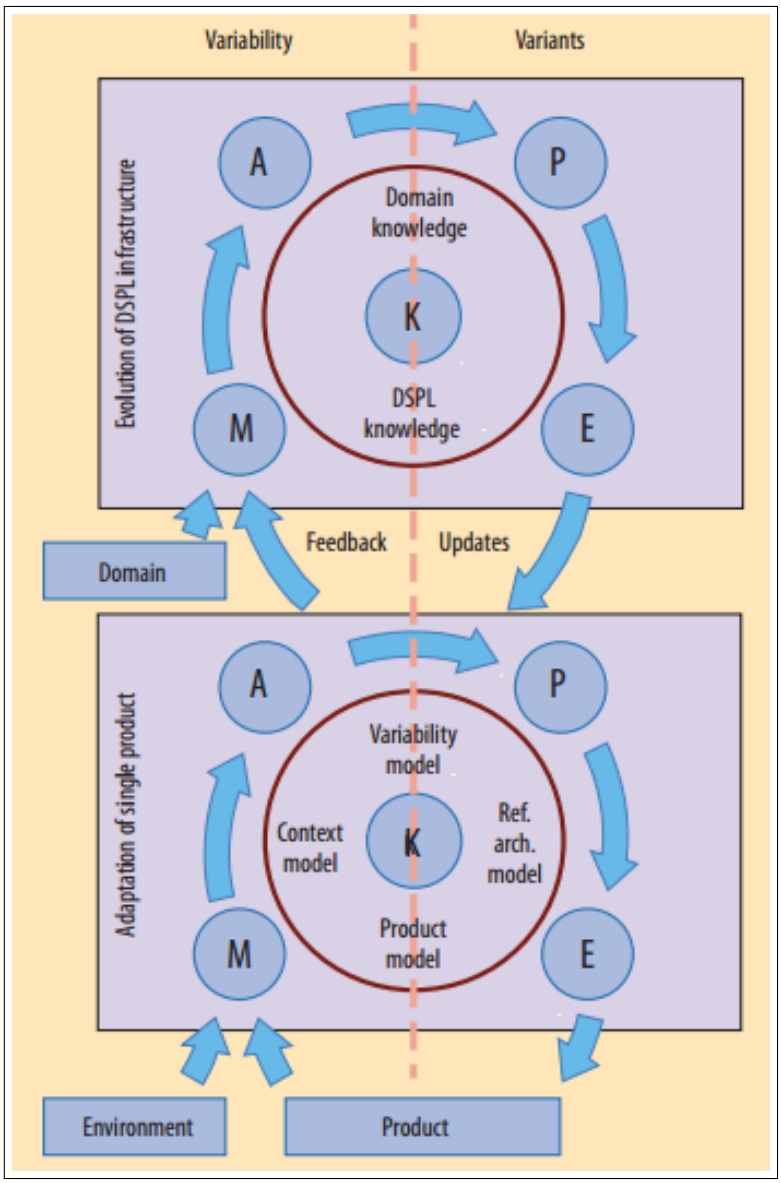
- **P1**: *Runtime Variability Support*. A DSPL must support the activation and deactivation of features and changes in the structural variability that can be managed at runtime;
- **P2**: *Multiple and dynamic binding*. In a DSPL, features can be bound several times and at different binding times (e.g., from deployment to runtime). The binding time is the time at which one decides to include or exclude a feature from a product (CHAKRAVARTHY *et al.*, 2008); and
- **P3**: *Context-Awareness*. DSPLs must handle context-aware properties that are used as input data to change the values of system variants dynamically and/or to select new system options depending on the conditions of the environment.

According to Bencomo *et al.* (2012a), the DSPL conceptual model is based on the SPL paradigm and the MAPE-K loop for autonomic computing, as depicted in Figure 6. In the upper part of this figure is presented the DSPL Domain Engineering (DE), whereas the lower part depicts the DSPL Application Engineering (AE). The Domain engineering produces the product line infrastructure consisting of the common architecture, the reusable artifacts, and the decision model. Such infrastructure is evolved by the MAPE-K loop driven by feedback both from the deployed product and the application domain evolution. The Application Engineering builds products for particular contexts represented by the context model.

The MAPE-K model comprises the main tasks of the feedback control loop of self-adaptive systems (BENCOMO *et al.*, 2012a; BASHARI *et al.*, 2017): (i) *Monitoring*, for detecting events which may require adaptation; (ii) *Analysis*, for analyzing the impact of a change in the requirements or constraints on the product; (iii) *Planning*, for deriving a suitable adaptation to cope with the new situation; and (iv) *Executing*, for carrying out the adaptation. The K denotes the *Knowledge*, which is usually represented by models which are used in the first four tasks.

Therefore, DSPLs, also known as self-adaptive SPL (CORDY *et al.*, 2013), have been considered an emergent paradigm to manage the variability at runtime and anytime (CAPILLA *et al.*, 2014a; BENCOMO *et al.*, 2012b). In this way, there are several studies describing the application of DSPL in different domains, like robotic (BRUGALI *et al.*, 2015), wind farm (MURGUZUR *et al.*, 2014) and cloud computing (BARESI; QUINTON, 2015).

Figure 6 – Classical SPL model with MAPE-K model



Source – Bencomo *et al.* (2012a).

2.2.3 Context Variation as Kripke Structure

Context-awareness is the system ability to observe the context changes and adapt its structure and behavior accordingly. In (self-)adaptive systems, the context characterizes situations in which the system must adapt to maintain compliance with its functional and performance specifications.

The context varies dynamically during the system execution. A **context state** is a snapshot of a system context in an instant of time t of its execution. Thus, the context variation can be seen as a sequence of context states over a period. The Context Kripke Structure

(Definition 2.2.1), proposed by Rocha and Andrade (2012), models this variation in a formal manner. The C-KS is a state-based formalism that models the system context variation as a transition graph in which nodes are context states and edges are transitions that represent changes in the system context.

Definition 2.2.1 (Context Kripke Structure) *A Context Kripke Structure (C-KS) is a 5-tuple* $\langle S, I, C, L, \rightarrow \rangle$ *where S is a finite set of context states,* $I \subseteq S$ *is a set of initial context states, C is a set of atomic context propositions,* $L : S \rightarrow 2^C$ *is a label function that maps each context state to a set of atomic context propositions that are true in that state, and* $\rightarrow \subseteq S \times S$ *is a transition relation.*

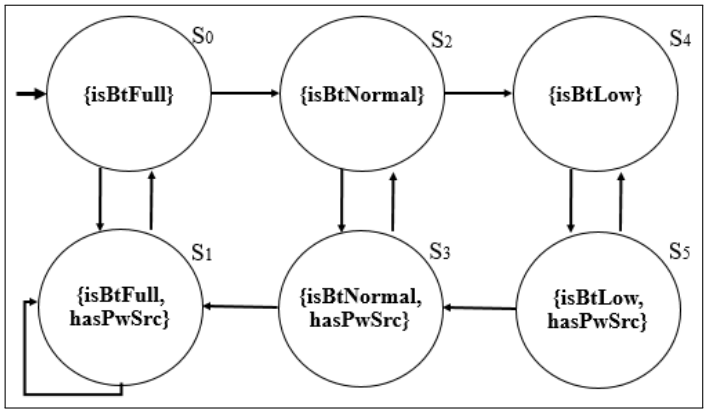
In a C-KS, the system context variation is viewed as a sequence of state transitions (e.g., $s_0 \rightarrow s_1 \rightarrow s_2 \ldots$). It is worth noting that C-KS defines context states and transitions without providing any form of causality. It means that C-KS does not explain why the system is in a specific state, or why it moves to another one. It collects possible values of different context variables that can occur in the system executions. Let $V = \{v_1, v_2, \ldots, v_n\}$ be a set of context variables that ranges over a finite set D (*domain of interpretation*). A valuation for V is a function that associates values in D to each context variable $v_i \in V$. In this sense, a context state is given by assigning values for all variables in V (i.e., $s : V \rightarrow D$).

An atomic context proposition $c \in C$ is a piece of context information defined as a logic proposition. A context proposition typically takes a form $v \circ d$, where $v \in V$, $d \in D$, and $\circ$ denotes a relational operator ($==$, $\leq$, $\geq$, $\neq$). The context propositions can be combined using classical propositional logic operators to describes high-level context information. Every context state is labeled by an element of 2^C (i.e., the powerset of C) that contains all context propositions that are true in this state (i.e., $L(s) \in 2^C$). So, given a valuation, a propositional formula that is true for this valuation can be written. For example, let s be a valid context state and $L(s) = \{c_1, c_2, c_3\}$ a label of s, the propositional formula $c_1 \wedge c_2 \wedge c_3$ that is true in this state can be derived.

Given the running example (see Section 2.1), let $V = \{\texttt{pwConnection}, \texttt{btStatus}\}$ be a set of context variables, and $D(\texttt{pwConnection}) = \{0, 1\}$ and $D(\texttt{btStatus}) = \{1, 2, 3, 4\}$ its respective domain of interpretation. Figure 7 presents the C-KS representing the context variation for the Mobile Guide DSPL products. The set of context states is $S = \{s_0, s_1, s_2, s_3, s_4, s_5\}$, the set of initial states is $I = \{s_0\}$, the $\rightarrow$ is given by $\{(s_0, s_1)$, (s_1, s_1), (s_1, s_0), (s_0, s_2),

(s_2,s_3), (s_3,s_2), (s_3,s_1), (s_2,s_4), (s_4,s_5), (s_5,s_4), $(s_5,s_3)\}$, the set of atomic context proposition is $C = \{\texttt{isBtLow}, \texttt{isBtNormal}, \texttt{isBtFull}, \texttt{hasPwSrc}\}$, where $\texttt{isBtLow} \equiv \texttt{btStatus} == 1$, $\texttt{isBtNormal} \equiv 2 \leq \texttt{btStatus} \geq 3$, $\texttt{isBtFull} \equiv \texttt{btStatus} == 4$, and $\texttt{hasPwSrc} \equiv \texttt{pwConnection} == 1$. The context states labels are $L(s_0) = \{\texttt{isBtFull}\}$, $L(s_1) = \{\texttt{isBtFull}, \texttt{hasPwSrc}\}$, $L(s_2) = \{\texttt{isBtNormal}\}$, $L(s_3) = \{\texttt{isBtNormal}, \texttt{hasPwSrc}\}$, $L(s_4) = \{\texttt{isBtLow}\}$, and $L(s_5) = \{\texttt{isBtLow}, \texttt{hasPwScr}\}$.

Figure 7 – C-KS for the Mobile Guide DSPL.



Source – the author.

Thus, the C-KS transitions in Figure 7 define the relationships among the context states. For instance, given a context state with a battery charge level full and without a power source (`isBtFull`); once the battery charge level is a variable affected by physical law (battery level usually decreasing), the model of Figure 7 specifies two possibilities to the next state: (1) the power source is connected (`isBtFull`, `hasPwSrc`); or (2) the battery charge level changes from full to normal (`isBtNormal`).

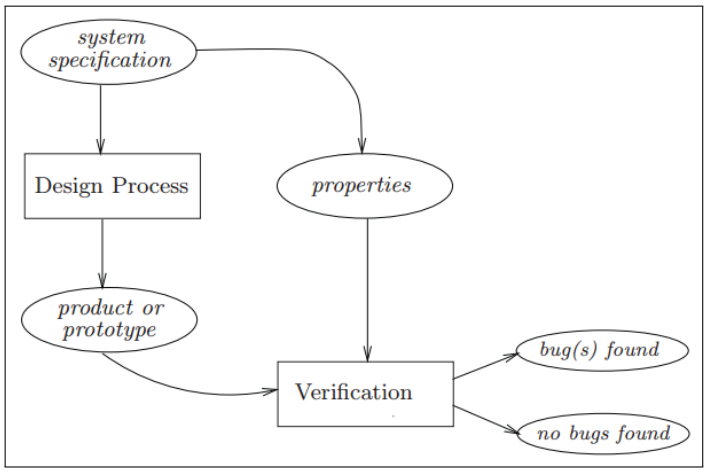
2.3 Software Verification

Verification is the process of evaluating a system to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase (IEEE, 2012). Thus, it is an important activity to assess the software during its life cycle.

Figure 8 depicts an overview of the Software Verification process. The basis for the verification is the system specification, from which are derived the properties that should be validated. For instance, a property could be never to reach a deadlock scenario, in which no process can be made. Thus, the system verification is used to establish that the design or product

under consideration possesses the defined properties (BAIER; KATOEN, 2008). A *bug* (defect) is found when the system does not fulfill one of the properties verified. Otherwise, the system is considered to be "correct".

Figure 8 – Verification Process Overview.



Source – Baier e Katoen (2008)

There are different techniques for software verification. For instance, one can use Inspection, which is an examination of the item against applicable documentation to confirm compliance with requirements (IEEE, 2012). Another technique is Model Checking, which is well-known for automatic verification of system designs (DIMOVSKI *et al.*, 2016; CORDY *et al.*, 2012). The latter is a model-based verification technique, since it is based on models describing the possible system behavior in a mathematically precise and unambiguous manner (BAIER; KATOEN, 2008).

Subsection 2.3.1 presents more details about the model checking technique, which is used by the testing method proposed in this thesis to automatically identify faults in the DAS adaptive behavior model. Next, Subsection 2.3.2 presents an approach to compute a feature model from a propositional formula, which is used by the method proposed to define behavioral properties that should be verified in DAS.

2.3.1 Model Checking

Model checking is a formal method employed in the automatic verification of finite state concurrent systems (CLARKE JR. *et al.*, 1999). Formal methods, in a nutshell, can be considered as "the applied mathematics for modeling and analyzing Information and

Communication Technology systems" (BAIER; KATOEN, 2008, p. 7). In the model checking approach, the system behavior is modeled using some formalism based on states and transitions (e.g., Kripke Structure (BIERE *et al.*, 1999)) and the system properties are specified using temporal logic (e.g., Linear Temporal Logic (BAIER; KATOEN, 2008)). The intended behavioral property verification is given by an exhaustive enumeration (implicit or explicit) of all reachable system states derived from the system model.

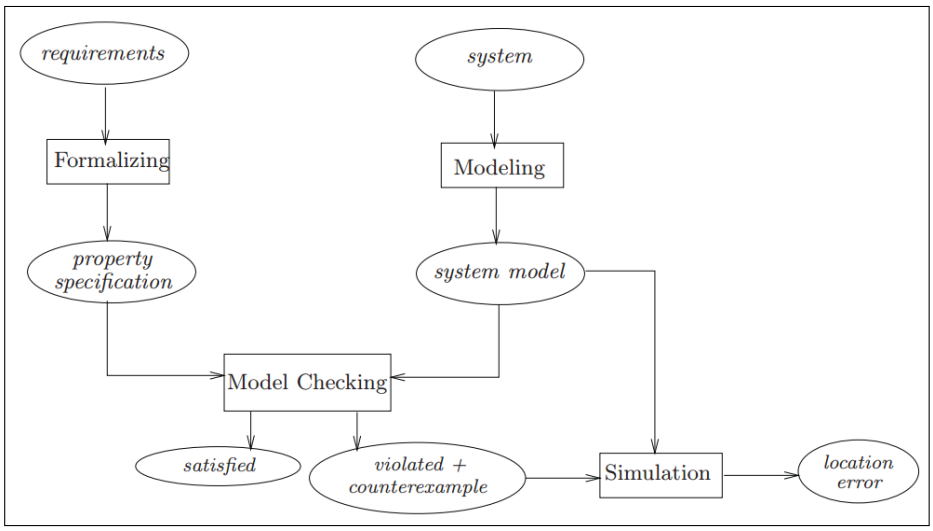
The model checking process can be divided into three main activities (CLARKE JR. *et al.*, 1999): (i) modeling – where the system model is built using a proper notation/language provided by the underlying model checker tool; (ii) specification – where the system properties are defined using a temporal logic supported by the model checker; and (iii) verification – where the properties are automatically checked against the system model by the underlying model checker tool.

Figure 9 presents an overview of the model checking approach. The first steps are the system modeling describing how it behaves and the definition of the properties that should be validated. During the verification, the model checker tool examines the system states to check whether they satisfy the desired property. If the property is not violated by the model, the model checker indicates that the property is satisfied. Otherwise, if a state is encountered that violates the property under consideration, the model checker provides a counterexample. This counterexample describes an execution path that leads from the initial system state to a state that violates the property being verified (BAIER; KATOEN, 2008). Then, by using a simulator the user can simulate the violating scenario to locate the error source.

In mathematical logic, temporal logic is a formalism based on a system of rules and symbolism for representing, and reasoning about, the notion of time (CLARKE JR. *et al.*, 1999). This kind of logic has temporal operators that allow expressing the notion of past and future. Typically, the temporal logic formulae are interpreted over Kripke Structures. For instance, given a Kripke Structure $\mathscr{K}$ and a temporal logic formula φ, a general formulation of a model checking problem consists in verifying if φ is satisfied ($\models$) in the $\mathscr{K}$ structure (i.e., $\mathscr{K} \models \varphi$).

This thesis uses Linear Temporal Logic (BAIER; KATOEN, 2008) to express behavioral properties over the DAS adaptive behavior model. This temporal logic was chosen due to its increasing popularity (BARTOCCI; LIó, 2016; OUAKNINE; WORRELL, 2008). Also, it is the logic supported by the SPIN, which is a well-know model checker tool (BAIER; KATOEN, 2008) and that is used in this work during the DAS model checking, as presented in Chapter 4.

Figure 9 – Model Checking Process Overview.



Source – Baier e Katoen (2008)

LTL is a linear-time temporal logic that makes it possible express properties over system states. The LTL formulae are built on top of atomic propositions using propositional operators ($\neg$, $\wedge$, $\vee$, $\rightarrow$, and $\leftrightarrow$) and temporal operators ($\bigcirc$ - next, $\Diamond$ - eventually, and $\Box$ - always). The LTL formulae are interpreted over execution paths of Kripke Structure. Let ϕ be an LTL formula, the intuition for the meaning of the LTL temporal operators is given in Table 2. More details about LTL can be found in (CLARKE JR. *et al.*, 1999; BAIER; KATOEN, 2008).

Table 2 – LTL Temporal Operators Meaning.

Operator	Meaning
$\bigcirc\phi$	"ϕ is true in the next state of the path."
$\Diamond\phi$	"eventually, ϕ is true in some state in the path."
$\Box\phi$	"always, ϕ is true in all states in the path."

Source – the author.

Also, it is worth noting that the model checking has several benefits described as follows (BAIER; KATOEN, 2008): (i) it is a general verification approach that is applicable to a wide range of applications; (ii) it supports partial verification since it allows to check properties individually; (iii) it is not vulnerable to the likelihood that an error is exposed; (iv) it provides diagnostic information in case of a property violation to support the identification of the defect source; and (v) it does not need a high degree of user interaction nor a high degree of expertise.

2.3.2 Feature Model as Propositional Formula

This section describes the Czarnecki and Wasowski's approach to map feature diagrams into propositional logic formulas (CZARNECKI; WASOWSKI, 2007). To better understand this approach, it is important to know some basic concepts about SPL feature models.

A feature model represents all products of SPL in terms of features. In the feature model, the features are presented hierarchically, and the basic rules are (BENAVIDES *et al.*, 2010): (i) the root feature is included in all products; and (ii) a child feature can only be included in a product if its parent feature was included.

Besides that, usually a feature model allows the following relationships among features (BENAVIDES *et al.*, 2010): (i) *Mandatory*, a child feature has a mandatory relationship with its parent when the child is included in all products in which its parent feature appears; (ii) *Optional*, a child feature has an optional relationship with its parent when the child can be optionally included in all products in which its parent feature appears; (iii) *Alternative (Xor)*, a set of child features has an alternative relationship with its parent when only one child feature can be selected when its parent feature appears; (iv) *Or*, a set of child features has an or-relationship with its parent when one or more of them can be included in the products in which its parent feature appears.

A feature model can also contain cross-tree constraints specified by *require* and *exclude* relationship between features (BENAVIDES *et al.*, 2010). If a feature *A exclude* a feature *B*, then both features cannot be included in the same product. If a feature *A requires* a feature *B*, then there is a dependence relationship where if the feature *A* is included in a product, then the feature *B* would also be included.

Czarnecki and Wasowski (2007) discuss the semantics of feature diagrams and their relation to logic. They show that is possible to describe all configurations of a feature model by a propositional formula defined over a set of Boolean variables (atomic propositions), where each variable corresponds to a feature. Aiming to do that, Czarnecki and Wasowski (2007) propose to represent a feature model as a conjunction of (i) implications from all subnodes to their parents, (ii) additional implications from parents to all their mandatory features, (iii) implications from parents to groups, and (iv) any additional constraints (e.g., require relationship) represented as propositional formula. Table 3 summarizes the logic formulae correspondents to the relationships among a parent feature p and its subfeatures $f_1, \ldots, f_k$. For simplicity, henceforth, this thesis uses the notation (i) $\texttt{fpf}(FM)$ to refer to propositional formula of a feature model FM; and (ii)

$\texttt{afp}(FM) = \{f_0, \ldots, f_n\}$ to refer to a set of all atomic feature propositions, which represent the features of FM.

Table 3 – Feature Relationships and their corresponding logic formula

Feature Model Relation Type	Logic Representation
child-parent	$\bigwedge_{i=1}^{k}(f_i \rightarrow p)$
mandatory	$\bigwedge_{i=1}^{k}(p \rightarrow f_i)$
or-group	$p \rightarrow (\bigvee_{i=1}^{k} f_i)$
xor-group	$p \rightarrow (\underline{\bigvee}_{i=1}^{k} f_i)$

Source – adapted from Czarnecki e Wasowski (2007).

Following the Czarnecki and Wasowski's approach (CZARNECKI; WASOWSKI, 2007), the $\texttt{fpf}(MobileGuideFM)$ from the running example (see Section 2.1) is given by a *conjunction* of formulas related to the relation child-parent (2.1) and the mandatory features (2.2):

$$((\ldots \rightarrow MobileGuide) \wedge (Video \rightarrow ShowDocuments) \wedge (Image \rightarrow ShowDocuments) \wedge (Text \rightarrow ShowDocuments)) \wedge \quad (2.1)$$

$$((MobileGuide \rightarrow \ldots) \wedge (ShowDocuments \rightarrow Text)) \quad (2.2)$$

Therefore, any assignment of Boolean values to all features ($\texttt{afp}(MobileGuideFM) = \{MobileGuide, \ldots, ShowDocuments, Text, Image, Video\}$) that makes the propositional formula satisfied represents a valid product configuration of the Mobile Guide feature model, otherwise, it represents an invalid configuration. For instance, if the feature *Show Documents* is included (ShowDocuments = $\top$) and the feature *Text* is not (Text = $\bot$), the configuration resulting is invalid, because these assignments make the formula $\texttt{fpf}(MobileGuideFM)$ unsatisfied since $(ShowDocuments \rightarrow Text) = \bot$.

2.4 Software Testing

Software Testing can be defined as the "dynamic verification of a program's behavior on a finite set of test cases, suitably selected from the usually infinite executions domain, against the expected behavior" (BOURQUE; FAIRLEY, 2014). This definition is interesting because it highlights that exhaustive testing is unlikely. Some rationales for this are: (i) the domain of possible inputs of a program is too large; (ii) it may not be feasible to simulate all possible system environment conditions; and (iii) the high cost of this activity, which could exceed 50 percent of the total cost of the software development (MYERS *et al.*, 2011).

Tests can be conducted in stages or levels (BOURQUE; FAIRLEY, 2014): unit, integration, and systems. The unit testing verifies the functioning in isolation of software elements that are separately testable, while integration testing validates the interactions among software components. At the level of system, the focus of this thesis, the testing activity is concerned with the behavior of an entire system.

Concerning the testing objective, the tests can be classified into different types. The functional testing, for instance, verifies if the software is developed according to the functional requirements established (MYERS *et al.*, 2011). This kind of testing is the focus of this thesis work, since it deals with the testing of the DAS adaptations. There are several other testing types such as performance and usability testing. The performance testing is specifically geared to verify that the software is in accordance with the specified performance requirements, such as capacity and response time (BOURQUE; FAIRLEY, 2014). The usability testing determines how well the final user can interact with the system (MYERS *et al.*, 2011). More details about the testing types can be found in Bourque e Fairley (2014).

Other concepts important in software testing are error, fault, defect, and failure. An error is a human action that produces an incorrect result (IEEE, 2010). A fault is a manifestation of an error in software (IEEE, 2010). Defect is an imperfection or deficiency in a work product where that work product does not meet its requirements or specifications and needs to be either repaired or replaced (IEEE, 2010). A failure, in turn, is an event in which a system or system component does not perform a required function within specified limits (IEEE, 2010). Thus, the testing can reveal failures, which may be caused by a fault (a subtype of defect).

Subsection 2.4.1 presents the main test case design techniques according to the ISO/IEC 29199-4 (IEEE, 2015). Next, Subsection 2.4.2 summarizes the challenges faced during the DAS testing.

2.4.1 Test Case Design

The ISO/IEC 29119-4 (IEEE, 2015) describes a series of techniques that have wide acceptance in the software testing industry. It also classifies them for three types of tests based on the source of information used to design test cases: (i) specification-based testing ("black-box testing"), where requirements, specifications or user needs are used as the main source of information to design test cases; (ii) structure-based testing ("white-box testing"), where the test item structure (*e.g.*, source code) is used as information source; and (iii) experience-based

testing, where the tester's knowledge and experience are used as primary information source. It is worth noting that these classes of test design techniques are complementary (IEEE, 2015).

The use of a test design techniques is important because it helps to specify test cases with a higher probability of finding defects. According to the ISO/IEC/IEEE 29119-4 (IEEE, 2015), a test case design technique provides guidance on the derivation of:

- A *test condition* that is a testable aspect of a test item, such as a function, transaction, feature, quality attribute or structural element identified as a basis for testing;
- *Test coverage items* that are attributes of each test condition that can be covered during testing. From a single test condition may be extracted one or more test coverage items; and
- A *test case* that is a set of preconditions, inputs and expected results, developed to determine whether or not the covered part of the test item has been implemented correctly. It has to be derived to exercise the test coverage items.

For instance, the Boundary-Value Analysis is a test case technique in which test cases are chosen on near the boundaries of the input domain of variables (BOURQUE; FAIRLEY, 2014). In this technique, the test conditions are the boundaries, while the test coverage items are the values on the boundary and an incremental distance outside the boundary (IEEE, 2015). Given a system that receives as input values from 1 to 10; these two values are the test conditions, while the test conditions items are the set {0,1,10,11}.

Based on the test cases and the test coverage items, it is possible to measure the test coverage through the Formula 2.3. In this case, `N` is the number of test coverage items covered by executed test cases, while `T` is the total number of test coverage items identified. For example, to measure the coverage of the Boundary Value Analysis (IEEE, 2015), `N` is the number of distinct boundary values covered, and `T` is the total number of boundary values.

$$Coverage = \frac{\mathtt{N}}{\mathtt{T}} * 100 \tag{2.3}$$

It is worth to highlight that the peculiarities of the application under test impact the test case designing activity. In context-aware software systems, the context information is a new kind of input affecting the application behavior (WANG *et al.*, 2007) and, therefore, should be taken into account to design a test case.

2.4.2 Challenges for DAS testing

The main characteristic of Dynamically Adaptive Software is that it can adapt at runtime according to the context information (GUEDES *et al.*, 2015). Both the use of context information and the reconfiguration of the software while running bring several challenges to the software testing activity.

As depicted in Section 1.5, aiming to investigate the testing for context-aware applications, it was performed a quasi-Systematic Literature Review (qSLR) about the test case design for this kind of system. From the initial set of 833 primary studies, this qSLR identified just 17 studies regard the design of test cases for context-aware systems. From these 17 studies, it was possible to identify testing challenges, as well as test design techniques and existing test criteria (discussed in Section 3.3) for context-aware systems. The details of this secondary study are described in (SANTOS *et al.*, 2017).

In addition, other systematic reviews have presented the challenges of testing Context-Aware Systems (MATALONGA *et al.*, 2017) and Adaptive Systems (SIQUEIRA *et al.*, 2016). Based on these studies and the performed secondary study (SANTOS *et al.*, 2017), the main challenges for DAS testing are depicted as follows:

- *To deal with the exponential growth of system configurations that should be tested.* Since the adaptive systems change over time and the number of configurations is huge (MATALONGA *et al.*, 2017; SANTOS *et al.*, 2017), one challenge is scope appropriately a test suite (SIQUEIRA *et al.*, 2016);
- *To design test cases for dealing with the uncertainty of contextual data.* Examples of uncertainties are unexpected power drain and physical damages into the sensors (SANTOS *et al.*, 2017). Adaptive systems should adapt correctly even in unpredictable situations. In this case, a major difficulty is to define the test cases to cover unforeseen configurations and that have never been tested in advance (SIQUEIRA *et al.*, 2016);
- *To identify incorrect configurations defined at runtime*. The dynamicity of adaptive systems may lead it to unpredictable configurations. Thus, the challenge is to dynamically define test cases to avoid incorrect settings of the system at runtime (SIQUEIRA *et al.*, 2016);
- *To anticipate all the relevant context changes and when they could impact the behavior of adaptive systems*. The environment may change continuously (SANTOS *et al.*, 2017; MATALONGA *et al.*, 2017) and then it can affect the system behavior at any time. So, this challenge concerns the need of anticipating context changes and building a test set that

properly encompasses all relevant context variables with representative values (SIQUEIRA *et al.*, 2016);

- *To automatically generate test cases for a changing environment*. Since adaptive systems change their structure continuously at runtime, it is hard to generate automatically test cases for them (SIQUEIRA *et al.*, 2016). Besides, it is unlikely the definition of a test oracle for all possible combination of values that can stimulate the adaptation (MATALONGA *et al.*, 2017).
- *To deal with the variation of context during the testing*. A context variation should be allowed unrestraint when executing the test item (MATALONGA *et al.*, 2017). Thus, this challenge is about to design test cases, allowing changes in the expected output according to the current context (SANTOS *et al.*, 2017).

2.5 Conclusion

This chapter introduced the running example that is used throughout this work. Next, it described the main concepts related to the topics involved in this thesis work, which are Context Awareness, Software Verification, and Software Testing.

With regards to Context Awareness, this chapter first introduced the definition of context, highlighting its importance for different application domains. After that, it was presented how to model context variability and how DSPLs use the context information to generate software variants at runtime. Also, a context variation model that specifies the behavior of the context states was presented since it is used by the testing method proposed in this thesis.

The Software Verification was addressed by introducing the verification process and an overview of the model checking technique. Furthermore, it was depicted how the feature model can be represented by a propositional logic formula. The latter is important to support the use of the model checking with DSPL/SPL feature models.

This chapter also introduced the main concepts of software testing, as well as the testing levels, types of tests and some test case design techniques. Besides that, it was presented a set of challenges related to the testing of adaptive systems. Among them, there are the combinatory explosion of the system's context and the uncertainty of contextual data.

The next chapter presents the related work to this thesis. Several studies concerning the DAS testing and model checking were identified, and based on a comparison of them, Chapter 3 describes the gaps in the literature that motivated this thesis work.

3 RELATED WORK

This chapter brings details about the literature review conducted by searching for studies concerning model checking or testing in DSPLs and Context-Aware Adaptive Software (CAAS).

The following sections present the studies found in the literature. Section 3.1 presents the studies related to model checking for CAAS. Section 3.2 discusses the studies concerning the model checking in the DSPL domain. Section 3.3 presents the related work to CAAS testing. Section 3.4 describes the studies related to DSPL testing. Section 3.5 presents a discussion of these papers. Finally, Section 3.6 concludes this chapter.

3.1 Model Checking for Context Aware Adaptive Software

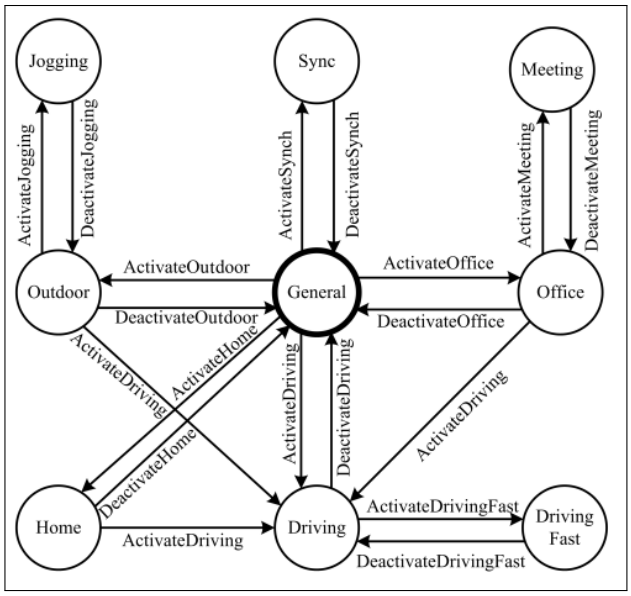
In the literature, there are several studies (SAMA *et al.*, 2008; SAMA *et al.*, 2010; LIU *et al.*, 2013; XU *et al.*, 2012; XU *et al.*, 2013; DJOUDI *et al.*, 2016; ARCAINI *et al.*, 2017) proposing approaches for model checking of Context Aware Adaptive Software.

Sama *et al.* (2008)(2010) propose a finite-state model of adaptive behavior, called Adaptation Finite-State Machine (A-FSM), which supports the detection of faults that introduces unwanted adaptations or unexpected states. Figure 10 presents the A-FSM of the Phone Adapter (SAMA *et al.*, 2010), which is an application that uses contextual information to adapt the phone's configuration profile to one of the nine predefined profiles. In this case, each A-FSM state represents an execution state (e.g., *Home, Driving*) of the CAAS, while the transitions represent the satisfaction of the adaptation rules predicates. For instance, one rule of the Phone Adapter states that if the phone is in the *Outdoor* profile and the formula "`GPS.isValid() AND GPS.speed()>5`" is true, then the profile should be changed to *Jogging*. Thus, this rule is active in the state *Outdoor*, and its execution is represented by the transition (label with *ActivateJogging*) from this state to the state Jogging that indicates the satisfaction of the formula before mentioned.

To support the faults identification, Sama *et al.* (2008) define a set of adaptation faults patterns: (i) *Determinism*: For each state in the A-FSM and each possible assignment of values to context variables, there is at most one rule that can be triggered; (ii) *State Liveness*: For each state in the A-FSM, if the state contains any active rules, then at least one of the active rules has a satisfiable predicate; (iii) *Rule Liveness*: For each state in the A-FSM and each one of

its active rules, there is at least one assignment of values to propositional context variables that satisfies the predicate of the rule; (iv) *Stability*: The state of an A-FSM is not dependent on the length of time a propositional context variable holds its value; and (v) *Reachability*: It should be possible to reach every state from the initial state. Moreover, other three fault patterns related to the asynchronous updating of context information are presented, and for each fault pattern the authors describe algorithms to check it by analyzing the A-FSM.

Figure 10 – Example of Adaptation Finite-State Machine



Source – Sama *et al.* (2010).

Liu *et al.* (2013) propose an approach and a tool, called AFChecker, to improve the precision of the A-FSM fault detection. In their approach, two models (a domain model and an environment model) are derived through deterministic and probabilistic constraints. To identify deterministic constraints, the authors use the concept of propositional atom $p(x)$ as a function that produces a truth value (i.e., true or false) by evaluating its context variable x (e.g., *slowDriving*(GPS.speed) = true if GPS.speed < 50km/h). Thus, the deterministic constraints are inferred by analyzing propositional atoms in pairs to derive internal correlations. For instance, if two atoms $p(x)$ and $p'(y)$ cannot both be true at same time, AFChecker infers the constraint $\neg p(x) \vee \neg p'(y)$.

The probabilistic constraints, in turn, are extracted from static analyses of the dynamically collected environmental information. In this analysis, their approach infers the correlation's probability and organize all probabilistic constraints in a weighted directed graph where: (i) each propositional atom $p(x)$ is mapped to two vertices, one vertex representing the positive truth

value assignment of $p(x)$ and other vertex representing the negative truth value assignment of $p(x)$; and (ii) the edges are associated with a likely correlation between the vertices. For example, an edge starting from the vertex where the atom $p(x)$ is true and ending at the vertex where the $p(y)$ is true, indicates that if $p(x)$ is evaluated to be true, then $p(y)$ is likely to be evaluated to be true. The likely correlation is the edge weight. Therefore, based on the inferred constraints, the Liu *et al.*'s approach prunes false positives (i.e., faults that cannot happen in practice) using deterministic constraints and ranking any remaining faults using probabilistic constraints.

Xu *et al.* (2012)(2013) state that the A-FSM (SAMA *et al.*, 2008) suffers expressiveness due to the use of a propositional logic based language. Thus, they propose an Adaptation Model (AM) that offers increased expressive power to model complex rules, which are adaptation rules based on the first-order logic that contains universal and existential quantifiers. An AM is a finite-state machine that contains a set of states `S` and a set of transitions that are the adaptation rules. In this model, a context variable can be mapped to either a single context value or a set of context values (for complex rules).

Furthermore, Xu *et al.* (2013) take into account variable dependency and physical constraints that enforce specific relationships among contexts (*e.g.*, two context variables that cannot both take true assignment at the same time) to avoid false positive (*i.e.*, unreal faults). To support the faults identification, these authors propose an algorithm based on the AM model to detect two types of fault in context-aware adaptive applications: *non-determinism fault* and *instability fault*. The *non-determinism fault* violates the property that for each state in an AM and each possible value assignment to context variables at that state, there is at most one active rule that can be triggered. The *instability fault* violates the property that an AM state is not dependent on the context update rate or rule execution speed.

Djoudi *et al.* (2016) combine model-driven techniques with formal methods to define a framework for context-aware systems specification and verification. The model-driven technique is used to identify context-aware systems concepts, their interrelation and specify the corresponding component-based meta-model. Formal methods, in turn, are used for formal specification and verification using the model checking technique. To support the association of formal semantic to the identified concepts of CAAS, the authors proposed a domain specific language called CTXs-Maude (for ConTeXt-aware Systems using Maude). Thus, this language allows to specify context entities and the adaptive behavior.

With regards to the verification process, Djoudi *et al.* (2016) use Maude LTL model

checker (EKER *et al.*, 2004) to ensure system safety and consistency by verifying the context-aware system. Therefore, the software engineer can specify properties to be checked against the model created. In their paper, a set of properties, expressing safety and liveness requirements, are verified using Maude model checker. The authors also developed the CTXstool that enables software engineer to graphically model, specify, and verify context-aware systems.

Arcaini *et al.* (2017) present a framework for formal modeling and analyzing CAAS with focus on distributed self-adaptive systems, which have a decentralized adaptation control. They define a formalism called *self-adaptive Abstract State Machines (ASM)* to specify the decentralized adaptation control by using MAPE-K loops. A self-adaptive ASM consists of a set of running agents divided into *managing* ones to control and perform the adaptation logic, and *managed* ones to perform the functional logic. Thus, the authors formalize a MAPE-K control loop in terms of actions of distributed *managing* agents.

In the verification process, the framework of Arcaini *et al.* (2017) supports the verification of system-independent properties (metaproperties) and requirement verification properties. In both cases, they use the model checker AsmetaSMV (ARCAINI *et al.*, 2010). The metaproperties checked are: (i) *knowledge locations are not in conflict*; (ii) *all rules involved in MAPE-K loops are executed*; and (iii) the *knowledge is minimal* (*e.g.*, it does not contain locations that are unnecessary). Therefore, these properties verify that the knowledge has been updated correctly and that the subsequent steps of the MAPE-K loop have been triggered correctly.

Therefore, it is possible to note that the system models used by the existing approaches are specified by meaning of a formalism based on states and transitions. Some of these models are focused on a specific type of adaptive system. For instance, the State Machine Model (LOCHAU *et al.*, 2015) supports the checking of properties over the DSPL staged reconfiguration processes with complex binding time constraints. On the other hand, the Abstract State Machine (ARCAINI *et al.*, 2017) is intended for distributed self-adaptive systems. Also, some proposals (SAMA *et al.*, 2008; SAMA *et al.*, 2010; XU *et al.*, 2012; XU *et al.*, 2013; LIU *et al.*, 2013) implement tools and propose models that only allow to check a predefined set of properties. This way, they are limited regarding the properties that can be verified, since they do not allow the user to check specific systems requirements.

3.2 Model Checking for Dynamic Software Product Lines

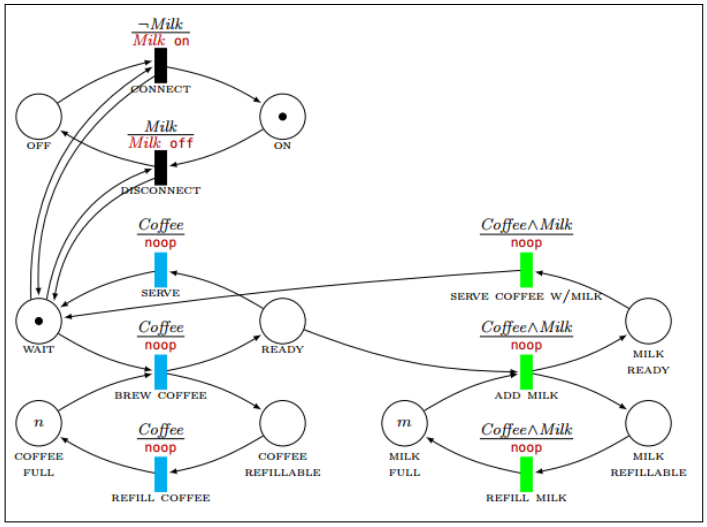
There are several papers that propose behavioral models for supporting the verification in the SPL domain (MUSCHEVICI *et al.*, 2010; CLASSEN *et al.*, 2010; CLASSEN *et al.*, 2013; CORDY *et al.*, 2012; VARSHOSAZ; KHOSRAVI, 2013; DIMOVSKI *et al.*, 2016). In a nutshell, these studies model the possible SPL products, annotating the transitions with information about the required features. So, the models proposed for SPL consider that a given configuration is chosen and fixed through the whole execution of the system (CORDY *et al.*, 2013).

In the DSPL scenario, however, there are different product configurations that are triggered according to context changes. Therefore, behavioral models for SPL fail to check properties of the DSPL adaptive behavior. To address this gap, some authors have proposed specific models to DSPLs. In the following paragraphs, the studies related to model checking in the DSPL domain (MUSCHEVICI *et al.*, 2010; CORDY *et al.*, 2013; MUSCHEVICI *et al.*, 2015; LOCHAU *et al.*, 2015) are discussed.

Muschevici *et al.* (2010) propose Dynamic Feature Petri Nets (DFPN), or Dynamic Feature Nets (DFN) for short (MUSCHEVICI *et al.*, 2015), extending the Feature Petri Nets (FPN) to capture the dynamic reconfiguration of products. In this approach, beyond the application conditions from the FPN, they associate to each transition an update expression that describes how the feature selection evolves after the transition. Thus, if the feature is dropped, this action globally disables all transitions whose application condition depends on the dropped feature. Figure 11 depicts a DFN for a coffee machine DSPL. In this model, the update expression ‘‘`Milk off`’’ and the application condition ‘‘`Milk`’’ are associated to the *disconnect* transition. Then, by firing *disconnect*, the feature Milk is dropped and the transitions whose application condition depends on the Milk feature are disabled, that is, *add milk*, *refill milk* and *serve coffee w/milk*.

Muschevici *et al.* (2010) also propose an analysis method for DSPL modeled as Dynamic Feature Nets by representing all possible traces in a relaxed variable reachability graph. In this graph, each node have a state of the DFN and a feature selection associated with it. For checking DFNs, one can use a traditional model checker as SPIN (HOLZMANN, 2003) or mCRL2 (GROOTE *et al.*, 2007). Furthermore, according to the authors, this graph can be seen as a Featured Transition System (FTS) (CLASSEN *et al.*, 2010), and then, it can also be checked using dedicated FTS model checkers.

Figure 11 – Example of Dynamic Feature Net

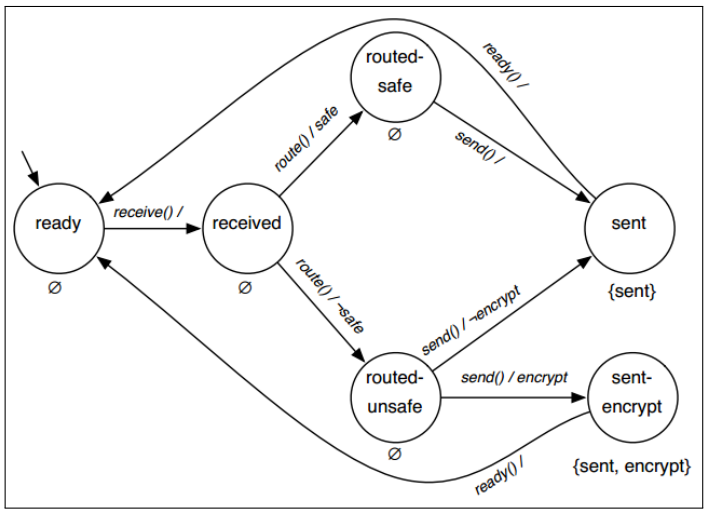


Source – Muschevici *et al.* (2015).

Cordy *et al.* (2013) propose Adaptive Featured Transition Systems (A-FTS), an extension of FTS (CLASSEN *et al.*, 2010) for modelling DSPLs. A-FTS model the evolution of both the environment and the adaptive system. In this approach, the capability of the software to execute a transition depends on both its features (system features) and those of the environment (environment features). An environment feature is a Boolean characteristic of the environment that may change over time and that the software has the ability to perceive. In A-FTS, a state is called *macrostate* and refers to the state of the system itself, its configuration and the environment state. Figure 12 presents an A-FTS for an adaptive routing protocol. This system has one adaptable feature called *encryption*, and one environment feature called *safe* that may or may not be enabled. The macrostate (*ready*; $\emptyset$; {*safe*}), for instance, means that the system is in the ready state, has not the feature encryption enabled and executes in a safe environment. The next state depends on the action executed (*e.g.*, *receive()*), how the environment evolves and how the system decides to reconfigure itself.

Additionally, Cordy *et al.* (2013) formally define the concepts of *environment strategy* and *reconfiguration strategy*, which determine how the context evolves and how the systems reacts, respectively. It is important to highlight that according to Cordy *et al.* (2013) the A-FTS is "just a fundamental model, which is used by the tools but is difficult to manage by humans". The authors also defined and implemented a model checking technique that allows to verify the transition systems against temporal properties. For supporting A-FSM checking, the authors also propose a property language, called AdaCTL, to describe requirements on adaptive systems.

Figure 12 – Example of Adaptive Featured Transition System



Source – Cordy *et al.* (2013).

Lochau *et al.* (2015) present an approach for modeling and verifying validity properties of staged reconfiguration processes, which imposes a prioritization among configuration decisions, with complex binding time constraints. For this, the authors extend feature models with binding time information to capture the semantics of staged reconfiguration processes, and present a semantic representation of DSPL staged reconfiguration behaviors. This representation is the basis of a state machine model and it integrates binding times constraints.

Then, Lochau *et al.* (2015) propose to use both the constraint-solving and model-checking approaches for the DSPL validation. For supporting model checking, the representation created is further translated into Promela to serve as input for the model checker SPIN (HOLZMANN, 2003). In this way, their proposal can automatically verify validity properties with focus in the binding time constraints on a DSPL specification. The following properties are checked by the proposal of Lochau *et al.* (2015): (i) *Proper initialization*, which is related to the notion of feature model satisfiability to staged configurations; (ii) *Reachability*, in which every state of the reconfiguration automaton is supposed to be reachable during the life cycle of a DSPL variant; (iii) *Progress*, which enhances the notion of core features to staged reconfigurations; and (iv) *Liveness*, which requires that from every configuration reachable, every other state within the reconfiguration automaton always remains eventually reachable from the current state.

Therefore, it is possible to note that some of the models identified for DSPL mode checking focus only on the execution states (“ready”, “wait”), such as the DFN (MUSCHEVICI *et al.*, 2010; MUSCHEVICI *et al.*, 2015). Other models also specify the environment states, like the A-FTS (CORDY *et al.*, 2013) that uses macrostates to specify the system state, features

configuration and environment state. Moreover, despite the importance of adaptation fault patterns to support the software engineer in the identification of design faults, most of the studies related to DSPL model checking (CORDY *et al.*, 2013; MUSCHEVICI *et al.*, 2010; MUSCHEVICI *et al.*, 2015) do not discuss about the identification of adaptation fault patterns.

3.3 Testing Context-Aware Adaptive Systems

As presented in Chapter 2, the coverage of tests cases is measured by testing coverage criteria. Thus, the use of a test cases design technique and test coverage criteria is important to achieve a good test suite.

In the literature, there is work (GRIEBE; GRUHN, 2014; AMALFITANO *et al.*, 2013) related to Context-Aware Adaptive Systems testing that use the coverage criteria applied commonly to the traditional (i.e., non-context-aware) application testing. For instance, Griebe and Gruhn (2014) propose a model-based approach to improve the context-aware mobile application testing. In their approach, first the system models (i.e. UML Activity Diagrams) are enriched with context information. Then, these models are transformed into Petri Nets. From the Petri Nets representation, a system testing model is generated and it is used the all-transition-coverage criterion (IEEE, 2015) to generate the tests. Amalfitano *et al.* (2013), in turn, present approaches based on the definition of reusable event patterns for the manual and automatic generation of test cases for mobile application testing. These authors measure the resulting code coverage in terms of lines of code (LOCs) and methods.

Other studies (WANG *et al.*, 2007; WANG; CHAN, 2009; WANG *et al.*, 2014; MUNOZ, 2010; MICSKEI *et al.*, 2012; YU *et al.*, 2014) deal with the context information in the test cases design by proposing new test coverage criteria. Also, some studies (RODRIGUES *et al.*, 2016; QIN *et al.*, 2016) discuss testing strategies to cope with context information, but without defining test coverage criteria. The following paragraphs presents these studies.

Wang *et al.* (2007) propose a white-box testing approach to improve the test suite of a context-aware application. This approach has the following steps: 1) it identifies key program points (context-aware program points), represented with a control flow graph, where context information can effectively affect the application's behavior; 2) it generates potential variants for each existing test case exploring the execution of different context sequences based on the generated control flow graph; and 3) it attempts to dynamically direct the application execution towards the generated context sequence. The outputs are Drivers (sequence of nodes

that are be used to drive the test execution) and the enhancement of an existing test suite for a context-aware application. To guide the tests generation, the authors also propose three test criteria to expose all types of contexts under execution (WANG *et al.*, 2007): Context-Adequacy, Switch-to-Context-Adequacy, and Switch-With-Preempted-Capp-Adequacy.

Wang and Chan (2009) define a metric named *context diversity* to capture the context changes inherent in a context stream. For example, the *context diversity* for the sequences "meeting room, present report" and "home, watch football" is two, after summing up the changes in the location (where "meeting room" is different from "home") and activity (where "present report" is different from "watch football"). Based on this measure, Wang *et al.* (2014) show how to select test cases with higher, lower, and more evenly distributed *context diversity* for constructing test suites that are adequate with respect to the data-flow testing criteria (IEEE, 2015).

Munoz (2010) presents a combinatorial testing technique named Multidimensional Covering Arrays (MDCA) for self-adaptive systems that make decisions based on past conditions. This technique selects the combinations (combinatorial selection) of the environmental property values that constitute each environmental condition, and the interactions among environmental conditions that represent the temporality. Regarding the temporality, it is related to the (intra-variation) interactions among the values of a single reasoning variable (i. e., the possible transitions of context values over time). For instance, given a variable *Speed* with two possible values "slow" and "fast", the set of 2-transitions is: *{(slow,slow); (slow,fast); (fast,fast); (fast,slow)}*.

Micskei *et al.* (2012) present an approach that uses context modeling and scenarios, in the form of extended UML 2 Sequence Diagrams, to capture the context and requirements of the system, and automatically to generate test data and test oracle. The testing is carried out in an interactive simulator environment and concentrated on the system-level behavior. Micskei *et al.* (2012) suggest the following coverage measures: (i) Context related coverage metrics, which measure what part of the context model has been covered during testing and what combinations of initial context fragments from different requirements were covered; (ii) Scenario related metrics that measure coverage on the scenarios (e.g., whether all scenarios have been triggered); and (iii) Robustness related metrics, which measure the thoroughness of the generation of extreme contexts by considering the coverage of violated constraints and potential boundary values from the context model.

Yu *et al.* (2014) propose a model-based testing approach that uses Bigraphical Reactive Systems to model the behavior of context-aware applications. A bigraph consists of two graphs: a place graph that captures notions of locality or containment, and a link hypergraph that models connectivity or associations. In their proposal, the bigraphical model-based testing uses the middleware and environment models as input in order to generate test cases. The authors also propose pattern-flow testing criteria that are similar to the data-flow testing criteria (IEEE, 2015), and define all-uses and all-def coverage, but in terms of bigraphical structures defined and used in the reaction rules. In another paper (YU *et al.*, 2016), the authors propose a backward-derivation testing approach to trace back the event sequences from a fault or undesired state by reversing the relevant reaction rules.

Rodrigues *et al.* (2016) propose an approach, called CATS Design, to design functional test cases for context-aware software systems. The goal of the CATS Design is to enable the development of test suites, including test cases which consider the variation of context during the test design and execution. This approach is based on the concept of Thresholds that is a disturbance capable of changing the system identity (RODRIGUES *et al.*, 2016). The main activities of this approach are: (i) *Context Variables Identification*, which intend to find the context variables in the requirements and by tacit knowledge; (ii) *Thresholds Identification*, where the tester should use the requirements documentation together with the collected context variables to reproduce the system in a conceptual model and an analytical model, and, then, to identify the thresholds; (iii) *Test Suite Generation*, where the tester uses the created models to specify test cases.

Qin *et al.* (2016) propose an approach called SIT (Sample-based Interactive Testing) for testing self-adaptive applications. This approach involves: (i) an interactive application model, which captures the characteristics of interactions between a CAAS and its environment; and (ii) a test generation technique, which uses adaptive sampling and measures the similarity of execution traces to systematically explore the CAAS's input space. Therefore, the SIT explores the input space through systematic sampling and returns a set of sequences of value assignments to the CAAS' input parameters to exercise different application behaviors. It is worth noting that in order to apply the sampling technique, the target of the authors are self-adaptive applications whose input parameters take real numbers as values from sensors.

Therefore, it is possible to observe that some studies (AMALFITANO *et al.*, 2013; GRIEBE; GRUHN, 2014; QIN *et al.*, 2016) do not define test coverage criteria based on

the context covered. On the other hand, some studies (WANG *et al.*, 2007; MUNOZ, 2010; MICSKEI *et al.*, 2012; WANG; CHAN, 2009; WANG *et al.*, 2014; YU *et al.*, 2014; YU *et al.*, 2016; RODRIGUES *et al.*, 2016) handle the context-awareness by defining test coverage criteria based on context information.

3.4 Testing Dynamic Software Product Lines

Aiming to support the testing of Software Product Lines, there are several studies in the literature. Most of these work (JOHANSEN *et al.*, 2012; HASLINGER *et al.*, 2013; KOWAL *et al.*, 2013; LOPEZ-HERREJON *et al.*, 2013; LOPEZ-HERREJON *et al.*, 2014; LAMANCHA *et al.*, 2015) propose solutions based on the combinatorial testing and focus on finding the set of products that should be used to represent the SPL in the testing scenario.

Johansen *et al.* (2012) propose a specialized algorithm, called ICPL, for generating covering arrays from large feature models. Haslinger *et al.* (2013) apply a set of rules exploring the feature model knowledge to reduce the test combinations that have to be covered. Kowal *et al.* (2013) propose an approach to reduce the combinatorial test set by explicitly modeling information about shared resources and communication in feature models, and, then, indicating the more likely feature interactions. Lopez-Herrejon *et al.* (2013) propose an algorithm for solving the multi-objective problem of minimizing the number of test products and maximizing the pairwise coverage. In another work (LOPEZ-HERREJON *et al.*, 2014), they study the application to SPL pairwise testing of four classical multi-objective evolutionary algorithms. Lamancha *et al.* (2015) present a greedy algorithm, called PROW (Pairwise with constRaints, Order and Weight) that handles constraints and prioritization for pairwise coverage.

In the DSPL domain, however, just a few work (CAFEO *et al.*, 2011; PüSCHEL *et al.*, 2012; HäNSEL; GIESE, 2017) were found. A possible reason to the lower number of papers focused explicitly on DSPL testing is the fact that the testing approaches of adaptive systems can be applied for testing DSPLs, since both have as main characteristic the context-aware adaptive behavior. The related work to DSPL testing are presented in the following paragraphs.

Cafeo *et al.* (2011) address the runtime testing for DSPLs by presenting an approach for inferring test results during the software execution. In their proposal, first, a representative subset of configurations should be tested in details before the DSPL is deployed. Next, once the DSPL has been deployed, the existing test results are used to approximate the test results for new untested configurations. For this purpose, their approach determines which tested configuration

is most similar to the untested one, and then, it infers the quality of the untested configuration based on the test results previously conducted for the similar one. This similarity among the DSPL configurations is based on the configurations structure defined in call graphs.

Püschel *et al.* (2012) present an approach applying Model Based Testing (MBT) and Dynamic Feature Petri Nets (DFPN) (MUSCHEVICI *et al.*, 2010) to define a test model from which an extensive test suite can be derived. This test model is based on DFPNs enriched with a parallel branch derived from the information given in the context rules. By combining the test model, context rules, and the feature model, a generator derives one test suite for each application configuration. In order to generate the test suite, the approach from Püschel *et al.* (2012) generates a complete simulation of the combined DFPN, consisting of the test model and the context branch to derive all possible traces (*i.e.*, sequences of DFPN's states), each one corresponding to a specific test case.

Hänsel e Giese (2017) propose an approach based on online and offline testing for DSPLs. They propose to make use of monitoring results from multiple instances of systems derived from a DSPL at runtime. Then, these observations and applied configurations are employed to estimate an up-to-date operational profile. After that, additional offline tests are incrementally run according to the estimated profile to ensure that the most relevant parts of the environment observation and configuration space are properly covered by systematic tests. Then, online testing is done at runtime to ensure that these observations and configurations are also properly covered by random tests. It is worth noting that to execute the tests, their proposal requires a facility called *test center* that also has the purpose of periodically collect the system status based on the current configuration and environment observation.

Therefore, it is possible to note that the found studies (CAFEO *et al.*, 2011; PüSCHEL *et al.*, 2012; HäNSEL; GIESE, 2017) focused on the DSPL testing do not define test coverage criteria based on the context covered. Also, two of the identified papers (CAFEO *et al.*, 2011; HäNSEL; GIESE, 2017) propose solutions to address the DSPL testing at runtime.

3.5 Discussion

In the literature, there are several studies dealing with the model checking of Dynamically Adaptive Systems. Some of them (SAMA *et al.*, 2010; LIU *et al.*, 2013; XU *et al.*, 2013; DJOUDI *et al.*, 2016; ARCAINI *et al.*, 2017) deal with Context-Aware Adaptive Software developed without the Software Product Line Engineering. Other studies (CORDY *et al.*, 2013;

MUSCHEVICI *et al.*, 2010; MUSCHEVICI *et al.*, 2015; LOCHAU *et al.*, 2015) focus on DAS developed using the DSPL Engineering.

As discussed in Section 2.3, the model checking technique is based on three activities: system modeling, properties specification, and properties verification by using a model checker tool. In this way, Table 4 summarizes the found papers according to the following criteria: (i) *System Model* - The formalism used to model the system behavior; (ii) *Tool* - The model checker tool used to verify the behavioral properties; (iii) *Fault Patterns* - Whether the paper present properties to check adaptation fault patterns, which means that any DAS satisfy these properties; and (iv) *User-Defined* - Whether the paper proposal allows to check properties defined by the software engineer to verify adaptation goals related to the system requirements.

Table 4 – Related Work to DAS Model Checking.

Reference	System Model	Tool	Properties Checked	
			Fault Patterns	User-Defined
(SAMA *et al.*, 2008) (SAMA *et al.*, 2010)	Adaptation Finite State Machine	Prototype	Yes	No
(XU *et al.*, 2012) (XU *et al.*, 2013)	Adaptation Model	Prototype	Yes	No
(LIU *et al.*, 2013)	Adaptation Finite State Machine	AFChecker	Yes	No
(CORDY *et al.*, 2013)	Adaptive Featured Transition Systems	Prototype	No	Yes
(MUSCHEVICI *et al.*, 2010) (MUSCHEVICI *et al.*, 2015)	Dynamic Feature Nets	SPIN or mCRL2	No	Yes
(LOCHAU *et al.*, 2015)	State machine model	SPIN	Yes	Yes
(DJOUDI *et al.*, 2016)	Defined by component modules based on CTXs-Maude grammar	Maude model checker	No	Yes
(ARCAINI *et al.*, 2017)	Self-adaptive Abstract State Machine	AsmetaSMV	Yes	Yes

Source – the author.

Most of the models identified for the DAS behavior focus only on the execution states (“ready”, “wait”), such as the A-FSM (SAMA *et al.*, 2010) and the DFN (MUSCHEVICI *et al.*, 2015). Some of them also model the environment states, like the A-FTS (CORDY *et al.*, 2013) that uses macrostates to specify the system state, features configuration and environment state. However, none of the identified formalisms supports the modeling of the active system features and context features according to the adaptation rules that can be triggered. This modeling is important because it can support the reasoning over the effect of the adaptation rules (i.e.,

activation/deactivation of system features) according to context changes.

Only three proposals (MUSCHEVICI *et al.*, 2010; MUSCHEVICI *et al.*, 2015; LOCHAU *et al.*, 2015; DJOUDI *et al.*, 2016) support the use of model checkers already known by practitioners. By using an existing model checker that is already commonly used by practitioners, their proposals can benefit from the present (and future) optimizations of this checker. Also, only the model proposed by Lochau *et al.* (2015) supports the use of a known model checker, and the checking of properties related to fault patterns, as well as user-defined properties. Their proposal, however, is focused on staged DSPL reconfiguration processes with complex binding time constraints.

Therefore, there is a lack of a formalism that represents the effects of adaptation rules in the DAS configuration, allowing the checking of properties related to the adaptation rules and the (de)activation of features. Furthermore, there is a lack of DAS model checking approaches that allow using existing model checkers, and the verification of properties defined by the user and related to fault patterns.

With regards to the DAS testing, there are several studies in the literature addressing different aspects of the Context Aware Adaptive Software testing. Table 5 present these papers according to the following criteria: (i) Test Technique - How the test cases are generated; (ii) Test Type - Whether the test strategy is black-box (i.e., based on the requirements) or white-box (i.e., based on the source code); and (iii) *Context-Based Coverage* - Whether the proposal uses or defines test criteria that taken into account the context to measure the tests coverage.

Five of the found studies use model-based testing. The models used by these techniques to specify the context-aware behavior with testing purpose are: (i) Bigraphical Reaction System model from Yu *et al.* (2014)(2016); (ii) UML Activity Diagrams enriched with context information from Griebe and Gruhn (2014); (iii) extended UML 2 Sequence Diagrams from Micskei *et al.* (2012); (iv) Dynamic Feature Petris Nets from Püschel *et al.* (2012); and (v) Conceptual model and analytical model from Rodrigues *et al.* (2016).

Regarding the test criteria used, the found papers apply different strategies. Some studies (WANG *et al.*, 2007; MUNOZ, 2010; MICSKEI *et al.*, 2012; WANG; CHAN, 2009; WANG *et al.*, 2014; YU *et al.*, 2014; YU *et al.*, 2016; RODRIGUES *et al.*, 2016) define test coverage criteria based on context information. For instance, Wang and Chan (2009)(2014) propose the metric *context diversity* to measure how many changes in contextual values of individual test cases. On the other hand, other studies (CAFEO *et al.*, 2011; PüSCHEL *et al.*,

2012; AMALFITANO *et al.*, 2013; GRIEBE; GRUHN, 2014; QIN *et al.*, 2016; HäNSEL; GIESE, 2017) do not define test coverage criteria based on the context covered. Cafeo *et al.* (2011), for example, infer the test results of an untested configuration based on similar configurations. Also, Griebe and Gruhn (2014) and Amalfitano *et al.* (2013) use data-flow criteria (IEEE, 2015) and coverage of lines of codes, which are applied commonly to traditional application testing (ELBERZHAGER *et al.*, 2012; SHAHID *et al.*, 2011).

Table 5 – Related Work to DAS Testing.

Reference	Test Technique	Test Type	Context-Based Coverage
(WANG *et al.*, 2007)	Context-aware program points	White-Box	Yes
(MUNOZ, 2010)	Combinatorial Testing	Black-box	Yes
(CAFEO *et al.*, 2011)	Inferring Test Results	White-Box	No
(PüSCHEL *et al.*, 2012)	DFPN-based testing	Black-Box	No
(MICSKEI *et al.*, 2012)	Context models and extended UML Sequence Diagrams	Black-box	Yes
(AMALFITANO *et al.*, 2013)	Reusable event patterns	Black-box	No
(GRIEBE; GRUHN, 2014)	UML Activity Diagrams enriched with context	Black-box	No
(WANG; CHAN, 2009) (WANG *et al.*, 2014)	Context diversity measure	Black-Box	Yes
(YU *et al.*, 2014) (YU *et al.*, 2016)	Bigraphical Reactive Systems	Black-Box	Yes
(RODRIGUES *et al.*, 2016)	CATS Design	Black-box	Yes
(QIN *et al.*, 2016)	Sampling-based interactive testing	White-Box	No
(HäNSEL; GIESE, 2017)	Online and Offline Testing	Black-Box	No

Source – the author.

Thus, most of the testing approaches deal with black-box testing, whose advantage is that they do not depend on the DAS source code. However, only some of them (MUNOZ, 2010; MICSKEI *et al.*, 2012; WANG; CHAN, 2009; WANG *et al.*, 2014; YU *et al.*, 2014; YU *et al.*, 2016) also proposed context-based testing coverage criteria. In particular, even measuring the context coverage, these criteria do not ensure the coverage of the adaptation rules effects. For example, they do not ensure the coverage of the DAS configurations resulting from the adaptation rules interleaving or the (de)activation of the DAS features by the adaptation rules. Therefore, there is a lack of black-box approaches to guide the DAS testing that take into account the coverage of the adaptation rules actions.

3.6 Conclusion

In this chapter, the related work to model checking and testing Dynamically Adaptive Software were presented. In particular, this chapter discussed 11 studies related to DAS model checking and 14 one related to DAS testing.

With regards to the DAS model checking, in the literature there are different proposals of formalism allowing the verification of properties over the adaptive behavior. Some of them also depict adaptation fault patterns that can help the software engineer to identify faults in the DAS adaptive behavior. Furthermore, the existing work usually propose proprietary algorithms to verify a fixed set of properties, and just a few work present a formalism that can be used with model checkers tools widely used by practitioners.

Concerning the DAS testing, most of the proposals deals with the black-box testing allowing to test the DAS without analyzing its source code. There are studies extending UML Activity Diagram or Sequence Diagrams with context information, and other proposing new test models. Some of the existing proposals use testing coverage criteria applied to traditional application testing. Thus, these approaches do not take into account the context itself in the test coverage measure. On the other hand, there are studies that define new testing coverage criteria that use the context coverage. Nevertheless, there is a lack of verification and validation approaches that focus on the DAS adaptation rules. Since the existing approaches are not focused on these rules and the (de)activation of features, they do not ensure the coverage of scenarios that could raise failures at runtime, such as the interleaving of adaptation rules or specific sequences of triggered adaptation rules.

The next chapter presents the proposal of this thesis for supporting the testing of dynamically adaptive software through the generation of test sequences based on the context and adaptation rules. These sequences are created from a model of the DAS adaptive behavior, which not only supports the tests generation, but also allows to verify behavioral properties on the DAS design.

4 TESTDAS AND SUPPORTING TOOLS

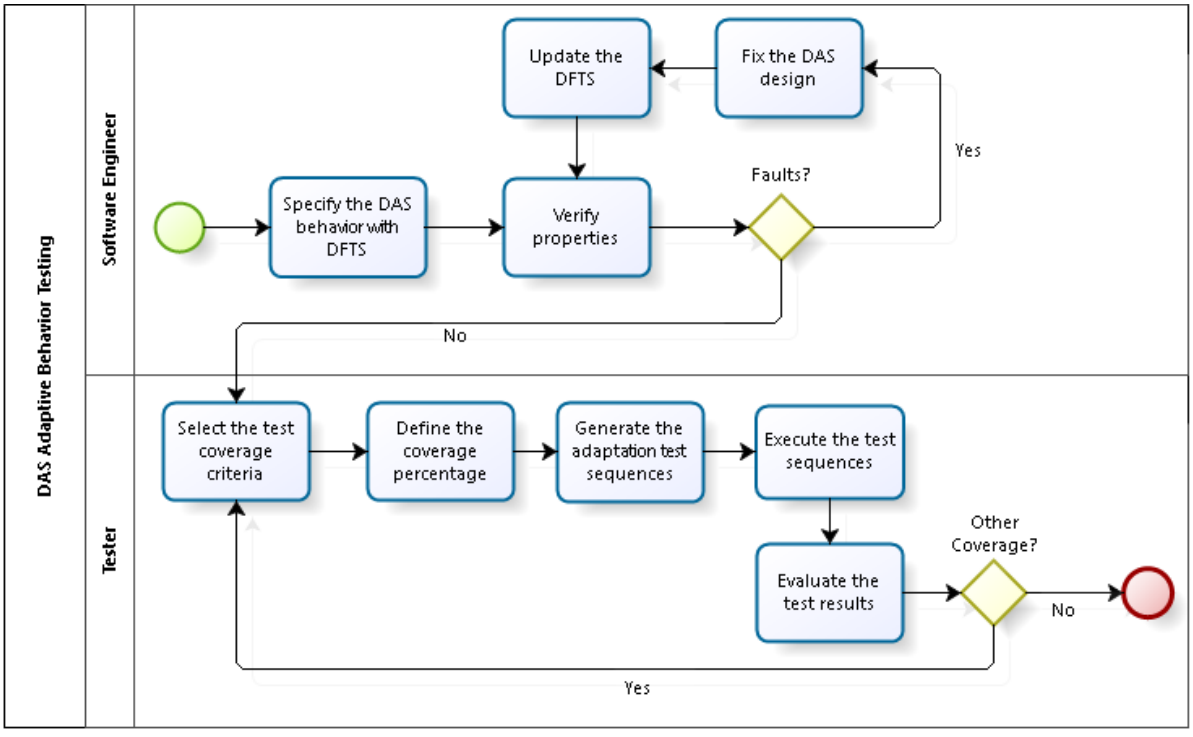
This chapter introduces a method, called Testing method for Dynamic Adaptive System (TestDAS), for supporting the testing of the DAS adaptive behavior. In a nutshell, this method receives as input the DAS feature model with adaptation rules, and a context variation model. Then, it provides an approach to DAS model checking and generates a set of tests for validating the DAS adaptive behavior.

More details about TestDAS are presented in the following sections that are organized as follows. Section 4.1 presents the overview of the TestDAS activities. Section 4.2 describes the formalism proposed to specify the DAS adaptive behavior. Section 4.3 presents the DAS model checking approach. Section 4.4 describes the TestDAS activities concerning the generation of test cases. Section 4.5 presents the tools implemented to support the TestDAS use. Finally, Section 4.6 concludes this chapter.

4.1 TestDAS Overview

Figure 13 presents the activities of the TestDAS method and the roles involved (i.e., Tester and Software Engineer) in performing these activities.

Figure 13 – TestDAS Overview



Source – the author.

The role of the Software Engineer in TestDAS is to verify whether the DAS design is correct (*i.e.*, it does not contain design faults). The correctness of the DAS design (i.e., its feature model and adaptation rules) is important, because it is used in TestDAS to guide the tests generation.

In the first activity of TestDAS, the Software Engineer should specify the DAS adaptive behavior using the Dynamic Feature Transition System (DFTS), which is the formalism proposed in this thesis to support the validation of the DAS adaptive behavior. The DFTS models the changes of the DAS configurations according to context changes and the triggered adaptation rules. Thus, such model describes the snapshots of context and system features in a given instant of time. This model is presented in Section 4.2.

In the scope of this work[1], the DFTS is created using Promela, which is the language of the SPIN[2] model checker tool. In this way, it is possible to use SPIN (HOLZMANN, 2003), which is the most well-known LTL model checker (BAIER; KATOEN, 2008), to verify behavioral properties. Details about how to create a DAS behavior model using DFTS and Promela are depicted in Section 4.3.

After that, the Software Engineer should specify and verify behavioral properties against the DFTS using the SPIN tool. To support this activity, a set of five general properties are presented in Section 4.3. These properties can be used to help in the identification of design faults in any DAS, since they are related to adaptation fault patterns. Besides that, the Software Engineer can specify and check domain-specific properties from the application requirements.

If any of the properties checked are violated, then the DAS has design faults in the adaptive behavior specification (*e.g*, adaptation rules with conflicting actions). Therefore, the Software Engineer should fix these faults in the DAS design. It is worth noting that this thesis considers the DAS design as its feature model, context model, and adaptation rules. After the "Fix the DAS design" activity, the Software Engineer should update the DFTS created and verify again the properties.

Once the DAS design is correct with regards the checked properties, the tests generation activities can be started. At this moment, the behavioral properties checked during the DAS design verification are used to guide the tests generation.

Then, by using TestDAS, the Tester generates the tests to achieve the required test

[1] Since DFTS is a formalism to specify the DAS behavior, it can be defined using other languages and, thus, it is possible to use other model checker tools

[2] http://spinroot.com/spin/whatispin.html

coverage. To do this, the first task is to select the test coverage criteria that define the required coverage. Section 4.4 proposes a set of five test coverage criteria that are defined based on behavioral properties presented in Section 4.3.2. The choice of the test criteria is based on the needs and the application under testing.

The Tester also defines the percentage of the required coverage (vary from 0 to 100%) for each criterion chosen. This percentage can be used to prioritize some criterion or to avoid a huge mass of tests that can require an unfeasible execution time.

After that, the Tester must generate test cases to satisfy the selected test criteria. These test cases are created in TestDAS as "adaptation test sequences" and they explore different adaptation sequences of DAS. Some algorithms for the test sequences generation are presented in Section 4.4.

Next, the tests are executed and the results are analyzed to evaluate whether the tests coverage is enough or more tests are needed. If no more tests are required, then the testing of the DAS adaptive behavior is concluded; otherwise, new test sequences can be generated by selecting other test coverage criteria.

A supporting tool, called TestDAS tool, is implemented to automatize the application of the TestDAS method to generate the test sequences. Also, a library called *CON*text variability based software *T*esting *L*ibrary (CONTroL) is implemented to support the test sequences execution. Details about these tools are depicted in Section 4.5.

4.2 Modeling the DAS Adaptive Behavior

In this section, the model proposed in this work to capture the adaptive behavior of a Dynamically Adaptive System is presented. The main idea of this model is to describe the possible configurations of the system and context. For this purpose, this model uses as input the DAS feature model with its adaptation rules, and a context variation model, which specifies the behavior of the context over time. The latter is a Context Kripke Structure (C-KS), presented in Section 2.2.3, which presents the context behavior through context states and their transitions.

One important characteristic of DAS is that one or more adaptation rules can be triggered at runtime in response to the current context state. Thus, this section starts by discussing, in Subsection 4.2.1, the interleaving nature of adaptation rules and how to formalize it. After that, the Dynamic Feature Transition System (DFTS) is introduced in Subsection 4.2.2 as the proposed formalism to model the DAS adaptive behavior.

4.2.1 Adaptation Interleaving and Effect

The adaptation rules are context-triggered actions responsible for the activation and deactivation of features in the DAS feature model. Thus, this work defines an adaptation rule as presented in the Definition 4.2.1.

Definition 4.2.1 (Adaptation Rule) *Let* $\texttt{afp}(FM) = \{f_0, ..., f_n\}$ *be the set of features of the feature model FM. An adaptation rule is a tuple* $\langle \omega, \alpha(F) \rangle$ *where* ω *is a context guard condition and* $\alpha \in \{\texttt{act}, \texttt{deact}\}$ *is an atomic action that simultaneously changes the value of features in* $F \subseteq \texttt{afp}(FM)$*. If* $\alpha = \texttt{act}$ *then the features in F are activated (i.e., assigns the truth value true). Otherwise, if* $\alpha = \texttt{deact}$ *then the features in F are deactivated (i.e., assigns the truth value false).*

For instance, given the running example (see Section 2.1), the $\texttt{afp}(MobileGuideFM)$ is the set $\{MobileGuide, ..., ShowDocuments, Text, Image, Video\}$. In this case, the rule AR01 from Table 1 can be represented by the tuple $\langle \texttt{isBtLow} \wedge \neg \texttt{hasPwSrc}, deact(Image, Video) \rangle$, which means that both features *Image* and *Video* are deactivated when the battery charge level is low (`isBtLow`) and there is not a power source ($\neg$`hasPwSrc`).

Besides that, depending on the current context at runtime, more than one adaptation rule can be triggered simultaneously. In this work, the "ready to run" rules are called **active adaptation rules**. Thus, in the parallel execution of active adaptation rules, actions of independent adaptation rule are merged, or interleaved, with actions from other rules. Hence, adaptation rules concurrency is represented by the non-deterministic choice between actions of simultaneously acting adaptation rules. The notion of adaptation interleaving caused by the parallel execution of a set of active adaptation rules is formally given in Definition 4.2.2.

Definition 4.2.2 (Adaptation Interleaving) *Let R be a set of active adaptation rules of a DSPL in a given context state, and* $A = \{\alpha(F) \mid \langle \omega, \alpha(F) \rangle \in R\}$ *be the set of all actions from the active adaptation rules. The interleaving of all active adaptation rules is given by the function* $\texttt{ari}(R) = \{\alpha_1(F_1)\alpha_2(F_2)\ldots\ \alpha_{|R|}(F_{|R|}) \mid \alpha_i(F_i) \in A \wedge \alpha_i(F_i) \neq \alpha_j(F_j), i \neq j\}$*, which generates all sequences of actions whose size is* $|R|$*.*

In Definition 4.2.2, the sequences of actions derived from the function $\texttt{ari}(R)$ correspond to all possible parallel executions of a given set R of active adaptation rules. For instance, given the running example (see Section 2.1), if there is an additional rule AR06, which

specifies that without an earphone connected, the feature *Video* should be deactivated. Then, when the context is battery charge level full and earphone disconnected, both adaptation rules AR05 and AR06 are triggered at the same time. Therefore, the result of the function $\texttt{ari}(R)$ is the set $\{act(Image,Video), deact(Video); deact(Video), act(Image,Video)\}$, which describes the possible adaptation sequences.

Another important aspect in the DAS reconfiguration process is the effect of the adaptation rules. Definition 4.2.3 introduces the concept of Adaptation Effect that refers to the impact of a set R of active adaptation rules over a product configuration E. The adaptation effect is the set of all product configurations resulting from the execution of each sequence of actions derived from the interleaving of all rules in R. For instance, given the running example (see Section 2.1), if the initial configuration E has both *Image* and *Video* activated, and only the rule AR01 is active, then $\texttt{effect}(R,E) = \{MobileGuide, ShowDocuments, Text\}$, which is the configuration resulting from the adaptation actions (i.e., $\{deact(Video), deact(Image)\}$) of the rule AR01.

Definition 4.2.3 (Adaptation Effect) *Given a set R of active adaptation rules and a product configuration E, the adaptation effect is given by the function* $\texttt{effect}(R,E) = \{\Phi(\beta^{(1)},E) \mid \beta^{(1)} = \alpha_1(F_1) \wedge \beta \in \texttt{ari}(R)\}$*, which generates a set of all possible features configuration derived from the adaptation rules interleaving.*

$$\Phi(\alpha_i(F_i),\mathscr{E}) = \begin{cases} \alpha_i = \texttt{act}, i = |R| & \mathscr{E} \cup F_i \\ \alpha_i = \texttt{deact}, i = |R| & \mathscr{E} \setminus F_i \\ \alpha_i = \texttt{act}, i < |R| & \Phi(\alpha_{i+1}(F_{i+1}), \mathscr{E} \cup F_i) \\ \alpha_i = \texttt{deact}, i < |R| & \Phi(\alpha_{i+1}(F_{i+1}), \mathscr{E} \setminus F_i) \end{cases}$$

It is worth noting that the function `effect` does not specify the impact, in reason of the cross-tree constraints, of the inclusion/exclusion of a context-aware feature over the non context-aware features. For instance, lets suppose that, in the running example, the triggered rule in a given instant of time is the AR05 represented by the tuple $\langle \texttt{isBtFull}, \texttt{act}(Video, Image)\rangle$. If there is, in the Mobile Visit Guides, a feature named *Sound* and the cross-tree rule "*Video requires Sound*", then the activation of the feature *Video* at runtime enforce the activation of the feature *Sound*. In this scenario, given an initial configuration E, the function `effect` of the Definition 4.2.3 returns $E \cup \{Video, Image\}$. However, the expected result considering the cross-tree constraint should be $E \cup \{Video, Image, Sound\}$.

Therefore, the function `effect` is generic, because of the focus only on the adaptive behavior guided by the context-aware features, not describing, for example, the activation of non-context-aware features from cross-tree rules. By focusing only on the context-aware features, this function separates the concern of configurations triggered by the context changes from the changes enforced by the cross-tree rules. On the other hand, the function `effect` cannot capture the actual behavior of the DSPL being analyzed. For achieving this, the *stakeholders* can include policies of feature inclusion/exclusion in the function `effect` according to the DAS analyzed.

4.2.2 Dynamic Feature Transition System

This section introduces the Dynamic Feature Transition System (DFTS), which is a formalism to specify the DAS adaptive behavior focused on the system configuration status and the context changes. This model is derived from a Context Kripke Structure (C-KS) (ROCHA; ANDRADE, 2012a) and a DAS feature model with its adaptation rules. As presented in Section 2.2.3, the main concern of the C-KS is the modeling of the context variation (ROCHA; ANDRADE, 2012b) by specifying the relationship among context information (e.g., it can indicate two context situations that cannot be true at the same time). Therefore, the advantage of using C-KS as input to create a DFTS is twofold: (i) it makes possible to reduce the number of possible paths (context states) to explore; and (ii) it allows the verification in more realist scenarios, avoiding the identification of false positives faults.

Definition 4.2.4 presents the DFTS concept. A DFTS has two atomic proposition types: context propositions (P_C) and feature propositions (P_F). The propositions in P_C come from a C-KS, and they represent the context. The propositions in P_F represent all features in the feature model FM of the DSPL.

Definition 4.2.4 (DFTS) *Given a Context Kripke Structure* $CKS = \langle S, I, C, L, \rightarrow \rangle$ *and a DSPL with a feature model FM, a set R of adaptation rules, and a set E of initial product configurations, a Dynamic Feature Transition System (DFTS) is given by a tuple* $\langle S', I', P, L', \rightarrow' \rangle$ *where:*

- S' *is the set of configuration states*
- $I' \subseteq S'$ *is the set of initial configuration states defined by the rule*

$$\frac{e \in E \quad s \in I \quad A = \{\langle \omega, \alpha(F) \rangle \in R \mid L(s) \models \omega\}}{CS = \{F' \cup L(s) \mid F' \in effect(A, e)\} \quad I' = I' \cup CS}$$

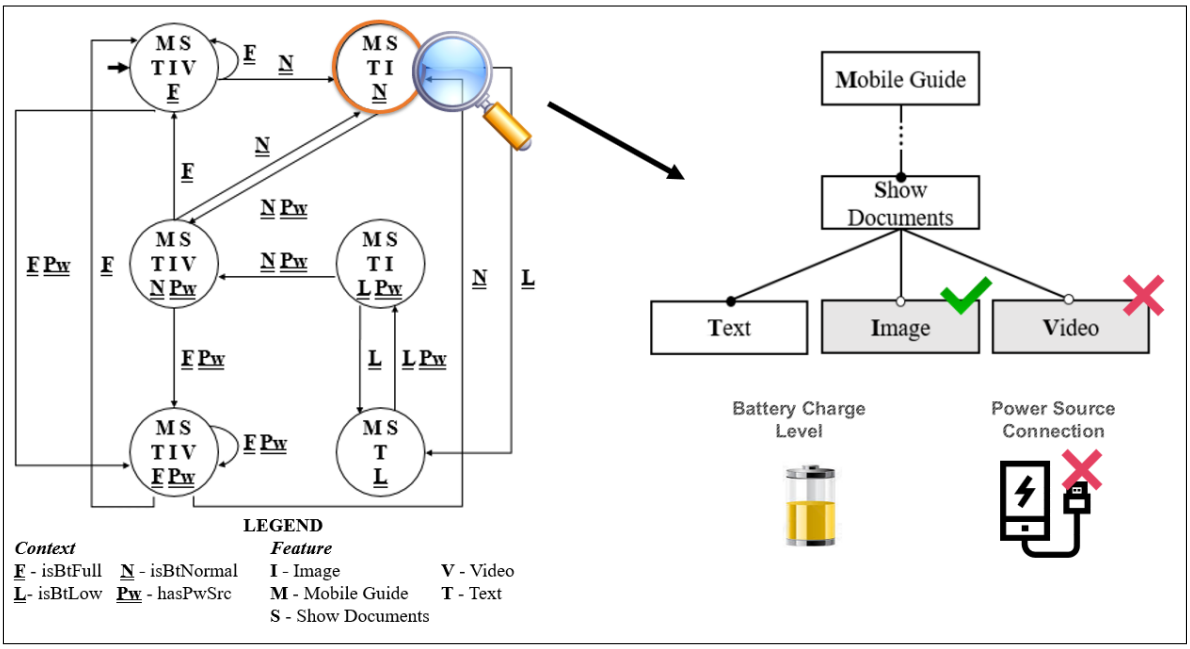
- $P = P_C \uplus P_F$ *is the set of atomic propositions that is partitioned into context and feature propositions with* $P_C = C$ *and* $P_F = \texttt{afp}(FM)$

- *L' is the labeling function such that $L' : S' \rightarrow' 2^P$. For sake of simplicity, this work also uses the notation L'_{P_F} to refer to the L' function over the feature propositions, and L'_{P_C} to refer to the L' function over the context propositions;*
- *$\rightarrow' \subseteq (S' \times P_c \times S')$ is the transition relation defined by a set of transitions rules T, where T = $\{\langle s_0, cs, \omega : \alpha(F), s_1\rangle \,|\, S_0, S_1 \in S'$, $cs \in C$, $\langle \omega, \alpha(F)\rangle \in R, L'(s_1) = L(cs) \cup effect(\alpha(F), L'_{P_F}(s_0)\}$. Therefore, the DFTS transitions are labeled with the set of context propositions $CP \in P_C$, where the truth value true of the context propositions in CP satisfy the propositional formula derived from the context state cs (see Section 2.2.3).*

It is worth noting that, in a DSPL, a feature can be activated by a user intervention. This intervention can be treated as a context proposition, and, therefore, DFTS can also cope with the (de)activation of features by user interventions.

Given the Mobile Guide DSPL (see Section 2.1) and its initial configuration, where only the features "Video" (V) and "Image" (I) are not activated, Figure 14 presents the corresponding DFTS.

Figure 14 – DFTS of the running example



Source – the author.

In the initial state of this DFTS, both features "Video" and "Image" are activated since the initial context (`isBtFull`, as depicted in Figure 7) triggers their activation (see Table 1). From this initial state, there is an arrow labeled with the context `isBtNormal` (label *N*) that

connects this state with another state where the feature “Video” is deactivated. Thus, this arrow represents the adaptation in which the context `isBtNormal` triggers the rule AR03 (Table 1) that deactivates the feature “Video”.

In summary, in DFTS, the states depict the active context and system features, while the arrows represent the adaptation (reconfigurations) among these states. In this way, DFTS reflects the effects of the adaptation rules over the DAS features. The DFTS in Figure 14 specifies the behavior of the Mobile Guide DSPL through the reconfigurations among six different *system states*, where the current state is defined by the current context.

4.3 DAS Model Checking Approach

As presented in Figure 13, some activities of the TestDAS method concern the model checking of the DAS design to avoid faults that can disturb the test results. A model checking approach involves three main activities (CLARKE JR. *et al.*, 1999): system modeling, properties specification, and verification. TestDAS defines these activities as follows:

- *Modeling*. The software engineer models the DAS adaptive behavior by using the DFTS (see Section 4.2). This modeling is made with the Promela language to allow the model checking with the SPIN checker tool;
- *Specification*. The software engineer specifies properties over the adaptation mechanism logic to be checked against the DFTS; and
- *Verification*. By using the SPIN model checker tool, the software engineer must check the defined properties over the DFTS to identify faults in the DAS adaptive behavior.

Subsection 4.3.1 presents how to specify the DAS adaptive behavior using the Promela language and the DFTS formalism. After that, Subsection 4.3.2 introduces five properties that can be used to identify design faults in any DAS, since they are independent of the application domain. Then, Subsection 4.3.3 describes a feasibility study, in which the proposed model checking approach is used to identify faults in two dynamic adaptive systems.

4.3.1 Mapping DFTS into Promela Code

The mapping from the DFTS concepts to a Promela code has twofold advantages: (i) by using the Promela, which is the specification language of the SPIN, the software engineer can specify other properties to be checked beyond the five properties defined in Section 4.3.2; and

(ii) the verification takes the advantages provided by SPIN, which has been widely accepted by the community and already apply partial order reduction techniques to optimize the verification process (HOLZMANN, 2003).

Promela programs consist of *processes* specifying behavior, and *message channels* and *variables* defining the environment in which the processes run (HOLZMANN, 2003). Therefore, to specify the DSPL behavior with Promela, the features and context propositions should be defined as *variables*, the messages that invoke the adaptation actions (feature activation and deactivation) should be specified with *message channels*, and the environment changes and adaptation rules should be defined as *processes*. Thus, the process of mapping DFTS into Promela is defined as follows:

- **Step 1**: Define the DFTS feature propositions (set P_F) as Boolean variables;
- **Step 2**: Define the DFTS context propositions (set P_C) as variables whose type depends on the kind of context information. For instance, a context information could be a Boolean variable or an Integer variable, in which the values can define different context status;
- **Step 3**: For each context-aware feature, define a process to be the actuator in charge of its adaptation. In other words, it is the process that will activate or deactivate the features by changing the values of their corresponding Boolean variables;
- **Step 4**: Define a process to manage the context changes. This process represents the context variation model, *i. e.*, the Context-Kripke Structure presented in Section 2.2.3;
- **Step 5**: Define a process to trigger the adaptation rules after context changes. This process has the role of *controller* since it monitors the current context and triggers the adaptation rules;
- **Step 6**: Define a process for each adaptation rule that triggers its adaptation actions if the context guard condition of the rule is satisfied. Since each adaptation rule is modeled by a different process, this modeling allows the interleaving among the adaptation rules; and
- **Step 7**: Define a channel with the messages `ctxChanged`, `adapted` and `done` to be used as flags to keep the communication among the process synchronously. These messages indicate when the context changes in the environment, the adaptation process is finished and the verification of the context guard conditions of the adaptation rule is done, respectively. Besides that, two more messages should be specified for each feature: one to require its activation and another to require its deactivation.

Code 1 presents part of the Promela code created to the DFTS representing the

adaptive behavior of the Mobile Visit Guides described in Section 2.1. The main elements of the Promela grammar are described in Appendix A.

Code 1 – Part of the Promela code for the running example

```
1
2  active proctype VideoActuator() {
3   do
4    :: atomic{ buss?videoOn -> video = true }
5    :: atomic{ buss?videoOff -> video = false }
6   od
7  }
8  active proctype ContextManager() {
9   battery = 3; hasPwSrc = false;
10  buss!ctxChanged;buss?adapted
11  do
12   :: (battery == 1 && hasPwSrc == false) ->
13   hasPwSrc = true;buss!ctxChanged;buss?adapted
14   :: (battery == 1 && hasPwSrc == true) ->
15   battery = battery + 1;buss!ctxChanged;buss?adapted
16   :: (battery == 2 && hasPwSrc == false) ->
17   battery = battery - 1;buss!ctxChanged;buss?adapted
18   :: (battery == 2 && hasPwSrc == true) ->
19   battery = battery + 1;buss!ctxChanged;buss?adapted
20   :: (battery == 3 && hasPwSrc == false) ->
21   battery = battery - 1;buss!ctxChanged;buss?adapted
22   :: (battery == 3 && hasPwSrc == true) ->
23   hasPwSrc =false;buss!ctxChanged;buss?adapted;break
24  od
25 }
26 active proctype Controller() {
27  do
28   :: buss?ctxChanged -> run AR01();run AR02();
29      run AR03();run AR04();run AR05();
30      numA = 0;
31      do
32       :: (numA != 5) -> buss?done; numA = numA + 1;
33       :: else -> break
34      od
35      buss!adapted
36  od
37 }
38 proctype AR01() {
39  if
40   :: (battery == 1 && hasPwSrc == false) ->
41   buss!imageOff;buss!videoOff;buss!done
```

```
42    :: else -> buss!done
43  fi
44 }
```

In Code 1, the features *Image* and *Video* are represented by two Boolean variables named “image” and “video”, respectively. Thus, if the value of these variable is true, it means that the corresponding feature is activated. The context proposition `battery` is specified as an Integer variable to refer to the different levels of battery charge (1 = `isBtLow`, 2 = `isBtNormal` and 3 = `isBtFull`). Besides that, the context proposition `hasPwSrc` is modeled through a Boolean variable indicating whether there is or not a connection to a power source. It is worth noting that in Promela, the value of a variable or the status of a message channel can only be viewed and changed by the processes, which are defined in a *proctype* declaration.

The process *VideoActuator* (line 2) represents the actuator in charge of the adaptation of the feature *Video*. If it receives the message `videoOn`, for instance, it activates the feature *Video* by changing the value of the variable *video* to true (line 4). The keyword `atomic` in the Promela language indicates that the statements sequence should be executed as one indivisible unit, *i.e.*, non-interleaved with other processes. Thus, the use of this keyword in lines 4-5 ensures that the adaptation effect will be instantaneous.

Furthermore, in order to keep only atomic reconfigurations, it was used the synchronous message exchange among the involved processes. To this end, the message channel, named “*buss*”, is defined with size zero creating a rendezvous port that can pass single byte messages, but cannot store messages. For example, the statement *buss!videoOff* (line 41) sends the message *videoOff* to the channel *buss*. Only after this message is retrieved by a process (*buss?videoOff* in line 5), the channel *buss* is available again to pass another message.

The process *ContextManager* has a loop defined by the repetition construct **do-od** to monitor and simulate the context changes based on the Context Kripke Structure of the Mobile Guide (see Section 2.2.3). When the context changes, this process sends the message `ctxChanged`, which is received by the process *Controller*. Once the process *Controller* receives this message, it invokes all the adaptation rules processes (e.g., run AR01(), line 28) and waits for the end of their executions that is signalized by the message `done`. After all adaptation rule processes have checked their context guard condition, the `Controller` sends the message `adapted` (line 35), which indicates that no more adaptations are required, to the process *ContextManager*. After that, the process *ContextManager* backs to monitor the context status and simulates new

context changes.

The process *AR01* (line 38) implements the rule AR01 from Table 1, by requiring the deactivation of both features *Image* and *Video* when the rule's context guard (e.g., `battery` $==$ 1 && `hasPwSrc` $== false$) is satisfied. Thus, if the AR01 condition guard is true (line 40), it triggers the adaptation actions and sends the message `done`. Otherwise, it just sends the message `done` to indicate that its evaluation was finished.

4.3.2 DAS Behavioral Properties

In the model checking, the behavioral properties specify conditions that the system model should satisfy. The violation of these properties reveals then faults in the system design. In the TestDAS, the identification of faults through the model checking is driven over the DFTS, which models the DAS adaptive behavior. In this case, although each DAS can have its properties to identify domain-specific faults, this kind of system share common adaptation faults patterns.

For instance, given that more than one adaptation rule of the DAS can be triggered at the same time, it is important to check the behavior of the system after the adaptation rule interleaving since the effect of one adaptation rule can cancel the effect of another rule. Besides that, the activation and deactivation of DAS features at runtime should always generate valid configurations, which are those that satisfy the rules of the DAS feature model.

Moreover, the adaptation rules that are never triggered should be identified, because this can indicate, for example, an error in the definition of the rules context guard conditions. It is also important to verify if the context-aware features are activated and deactivated. A context-aware feature that is never activated can indicate, for example, the lack of some rule to activate it at runtime. On the other hand, a context-aware feature that is never deactivated can indicate that it should be mandatory or that it is missing some rule to deactivate it.

Thus, to help software engineers in the DAS model checking process, this work defines a set of behavioral properties using Linear Temporal Logic (LTL) (BAIER; KATOEN, 2008) that should be satisfied. These properties capture the semantics of behavior patterns that need to be verified in any DAS design. They were defined based on the DSPL characteristics (CAPILLA *et al.*, 2014a), anomalies detected in SPLs (BENAVIDES *et al.*, 2010) and proprieties checked in context-aware applications (SAMA *et al.*, 2010). The properties defined in this work are described as follows:

Property 01: Configuration Correctness. This property specifies that the system

configuration in each state of the Dynamic Feature Transition System should be in conformance with the DAS feature model. The violation of this property shows invalid product configurations. This fault is called **invalid configuration fault**. Given a DSPL feature model $\mathscr{FM}$ and the function `fpf` presented in Section 3.1, Formula 4.1 specifies this property by stating that the propositional formula of the feature model should always be satisfied.

$$\Box \texttt{fpf}(\mathscr{FM}) \tag{4.1}$$

In the running example, presented in Section 2.1, if the adaptation rule AR01 was designed to deactivate the feature *Text*, then this would violated the `fpf`(Mobile Guide) described in Section 2.3.2. This *invalid configuration fault* would happens because *Text* is a mandatory feature and, thus, a system configuration with the feature *Text* deactivated is not a valid configuration.

Property 02: Rule Liveness. This property, defined based on the Sama *et al.*'s work (SAMA *et al.*, 2010), specifies that for each adaptation rule, should exist at least one state in the DFTS where the assignment of values to the atomic context propositions satisfies the context guard condition of the rule. The violation of this property shows adaptation rules that never are triggered. Thus, if the DFTS violates this property, then the DAS contains a **dead predicate fault**. Given a DFTS $\mathscr{D}$ and its adaptation rules defined by a tuple $\langle \omega, \alpha(F) \rangle$, Formula 4.2 specifies this property by stating that eventually the context guard condition of each adaptation rule is satisfied.

$$\forall \langle \omega, \alpha(F) \rangle \in \mathscr{D}, \quad \Diamond \omega \tag{4.2}$$

In the running example, if by mistake the context `isBtFull` was related with the battery charge level greater than 100%, then this context state would never occur. As another example, if the context `isBtFull` was battery charge levels equals to 100%, and in the real environment (specified by the Context Kripke Structure) this percentage is never achieved, then this context state is never true. In both cases, the rule AR05, which has `isBtFull` as context condition, will never be triggered and as a consequence, its adaptation actions will never be performed, generating a *dead predicate fault*.

Property 03: Interleaving Correctness. This property specifies that for each adaptation rule, its effect should be performed on the DAS, even in the situations where exists

an interleaving of adaptation rules. The violation of this property shows adaptation rules whose effect is not performed because of the rules interleaving process. This fault is called **nondeterministic interleaving fault**. Given a DFTS $\mathscr{D}$ and its adaptation rules defined by a tuple $\langle \omega, \alpha(F) \rangle$, where $\alpha(F) = \{f_1, \ldots, f_k\}$ indicates the set of activated features, Formula 4.3 specifies this property. This formula states that always when the rule context guard condition is true, there is a DFTS state with this context and the features activated by the adaptation actions of the rule triggered.

$$\forall \langle \omega, \alpha(F) \rangle \in \mathscr{D}, \quad \Box(\omega \rightarrow \Diamond(\omega \wedge \alpha(F))) \tag{4.3}$$

In the running example (Section 2.1), if the context guard condition of the adaptation rule AR03 was defined as `isBtNormal` $\wedge$ `hasPwSrc`, then both rules AR03 and AR04 would have the same context guard condition and, thus, they would be triggered at the same time. Since they affect the same features, the order of the execution of these rules impacts the final configuration. If first it is performed AR03 and then AR04, the feature *Video* will be activated, but, otherwise, the feature *Video* will be deactivated. This case is an example of *nondeterministic interleaving fault*.

Property 04: Feature Liveness. This property specifies that each context-aware feature should be active in some state of the DFTS. The violation of this property indicates a *functional overhead* when features included in the product are not used. This fault is called **dead feature fault**. Given a DSPL feature model $\mathscr{FM}$ and its set $\mathscr{FC}$ of context-aware features, Formula 4.4 specifies this property by stating that for each context-aware feature, there is a state in which it is activated (i.e., has true value).

$$\forall f \in \texttt{afp}(\mathscr{FM}) \wedge f \in \mathscr{FC}, \quad \Diamond f \tag{4.4}$$

In the running example, if both adaptation rules AR04 and AR05 by mistake did not activate the feature *Video*, then this feature would never be active. This situation represents a *dead feature fault* since the feature *Video* will never be enabled to use.

Property 05: Variation Liveness. This property specifies that each context-aware feature should be activated in some state of the DFTS and deactivate in some other state. The violation of this property shows a feature that does not adapt. This fault is called **false variable**

feature fault. Given a DSPL feature model $\mathscr{FM}$ and its set $\mathscr{FC}$ of context-aware features, Formula 4.5 specifies this property by stating that for each context-aware feature, there is a state in which it is deactivated (i.e., has false value).

$$\forall f \in \texttt{afp}(\mathscr{FM}) \wedge f \in \mathscr{FC}, \quad \Diamond !f \tag{4.5}$$

For instance, suppose that the effect of the rule AR01 in the running example is to activate both features *Image* and *Video*. In that way, the feature *Image* will never be deactivated, and then this feature will not be adaptive, resulting in a *false variable feature fault*.

The aforementioned properties aim to help in the identification of common faults of the DAS adaptation mechanism. In particular, the set of properties presented can help the software engineer to answer questions like: *Are all the product configurations in conformance with the DAS rules? Are there for each adaptation rule a context situation where this rule is triggered? Is there some context-aware feature that is never activated? Is there some context-aware feature that is never deactivated? Does the interleaving of adaptation rules triggered at the same time have the expected effect?*

4.3.3 Feasibility Study

In order to investigate the feasibility of the model checking approach proposed in this thesis, it was performed a study with two DSPLs: (i) Mobile Visit Guides DSPL (MARINHO *et al.*, 2013) presented in Section 2.1; and (ii) Car DSPL (MAURO *et al.*, 2016), which is an SPL based on an industrial scenario and related to the features of a car.

The goal of this study was to answer the following question: "*Do the DFTS and the defined properties support the identification of design faults in DAS feature models?*"

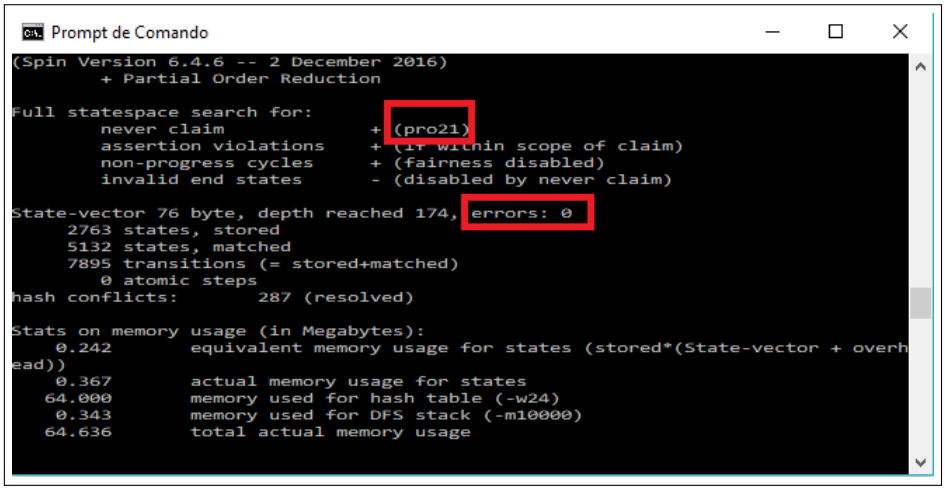
The first task was to define the Context Kripke Structure that models the context variation of the DSPLs used in the study. After that, the Promela codes with the DFTS corresponding to the DSPLs of the study were written using the mapping proposed in Section 4.3.1. Lastly, the SPIN (HOLZMANN, 2003) model checker tool was used to check the properties presented in Section 4.3.2.

In order to run the SPIN, it was used its Windows PC executable version[3]. Figure 15 shows an example of the execution of the SPIN to check the property referenced as *pro21* on the

[3] Download available at http://spinroot.com/spin/Src/index.html

Mobile Guide DSPL. This property refers to the “Rule Liveness” of the rule AR01 (see Table1) and its checking did not identify errors in the Mobile Guide design. This means that there is at least one state in the DFTS analyzed where the rule AR01 is triggered.

Figure 15 – Execution of the SPIN with command prompt in Windows



Source – the author.

In the following subsections, the results obtained during the checking of the Mobile Guide and Car DSPL are presented. After that, Subsection 4.3.3.3 presents a discussion about the results of this feasibility study.

4.3.3.1 Mobile Guide DSPL

In the Promela code for the Mobile Visit Guide DSPL were specified five processes to the adaptation rules, and two more processes to the actuators on the features *Image* and *Video*. Regarding the properties, they were specified by using the Linear Temporal Logic (see Section 4.3). In total, 15 properties were specified: (i) One for *Configuration Correctness*; (ii) Five to *Rule Correctness*, one to each adaptation rule in Table 1; (iii) Five to *Interleaving Correctness*, one to each adaptation rule; (iv) Two to *Feature Liveness*, corresponding to the features *Image* and *Video*; and (v) Two to *Variation Liveness*, related to the features *Image* and *Video*;

As initial configuration, the following features were specified as activated: *Mobile Guide*, *ShowDocuments* and *Text*. Besides that, the initial battery charge level was full (`isBtFull`) and the smartphone was not connected to a power source (`hasPwSrc` = `false`). This initial context state was defined by the C-KS depicted in Section 2.2.1. Furthermore, to verify if the model checking approach can identify faults in the Mobile Guide design, a fault

was deliberately inserted by changing the type of features *Image* and *Video* to a XOR-group. In this way, at least one of these features should be active at runtime, but the adaptation rule AR01 deactivates both features and this generates a fault. The complete Promela code of the DFTS of the Mobile Guide is available at Appendix B.

As result of the properties verification, the SPIN returned that the Mobile Visit Guides violates the property *Configuration Correctness*. This happened because the DFTS had states where none of the features *Video* and *Image* were activated, and this violates the feature model rules. More specifically, this violates the constraint $ShowDocuments \rightarrow (Image \vee Video)$ that determines that in each product state where "ShowDocuments" is activated, at least one of the features "Image" or "Video" should be activated. Therefore, the inserted fault was identified successfully. One solution to this fault is to remove the Or-group and to change the type of both features *Image* and *Video* to optional, as depicted in the feature model of Figure 2. Another solution is to use a cardinality-based feature model, as the feature model presented by Marinho *et al.* (2013).

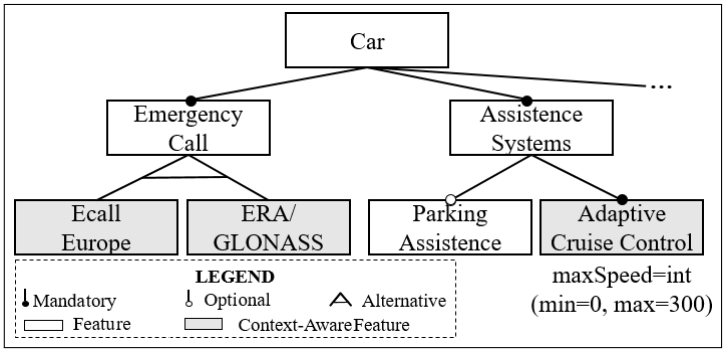
With regards to the other properties, none violation was found. Then, all the model adaptation rules are triggered in at least one state (*Rule Liveness*) and their effects are performed (*Interleaving Correctness*), and all context-aware features are activated at some state (*Feature Liveness*) and deactivated at some state (*Variation Liveness*).

It is worth noting that, as expected, it was not observed an interleaving of the adaptation rules in this DSPL, since their rules do not have a common context guard condition.

4.3.3.2 Car DSPL

The Car DSPL (MAURO *et al.*, 2016) has 16 features, five cross-tree rules and five adaptation rules. Figure 16 presents part of the Car feature model. This model has three context-aware features. Two of them (*Ecall Europe* and *ERA/GLONASS*) are activated according to the current location context. If the car is in Europe, the feature *Ecall Europe* is activated. If, however, the car is in Russia, the activated feature is *ERA/GLONASS*. The third feature (*Adaptive Cruise Control*) has an attribute value, named *maxSpeed*, in function of both road and location context. This attribute defines the maximum speed settable for cruise control. There are three adaptation rules in this case: (i) if the road is icy, then the *maxSpeed* should be lower or equal to 100; (ii) if the road is wet, then the *maxSpeed* should be lower or equal to 160; and (iii) if the location is Russia, then the *maxSpeed* should be lower or equal to 110.

Figure 16 – Part of the Car DSPL



Source – adapted from Mauro *et al.* (2016).

For the Car DSPL, the Promela code has five processes corresponding to the adaptation rules, and three more processes for the actuators on the features *Ecall Europe*, *ERA/-GLONASS* and on the feature attribute *maxSpeed*. As an initial configuration, it was specified the configuration described in the Mauro *et al.*'s work (MAURO *et al.*, 2016), where the feature *Ecall Europe* is activated and the initial value to the *maxSpeed* is 200. The complete Promela code of the DFTS of the Car DSPL is available at Appendix B.

As a result of the properties verification, it was possible to identify a violation on the property *Interleaving Correctness*. The explanation for this violation is that the interleaving among the adaptation rules produces a *system state* where the effects of all active adaptation rules are not performed. For instance, if there is icy on the road (road = icy) and the car is in the Russia (location = Russia), two adaptation rules are triggered: (i) (road = icy) $\rightarrow$ (maxSpeed <= 100) ; and (ii) (location = Russia) $\rightarrow$ (maxSpeed <= 110). In this case, the result of *maxSpeed* depends on the execution order of the adaptation rules. Therefore, there is a design fault, because the interleaving violates one of the actions from the triggered adaptation rules. It is worth noting that this is a fault present in the Car feature model presented in the paper of Mauro *et al.* (2016). One possible solution would be extending the context guard to make it more specific, for example, changing the last rule to (location = Russia $\wedge$ road = dry) $\rightarrow$ (maxSpeed <= 110).

4.3.3.3 Discussion

The goal of the feasibility study was to investigate if the DFTS (see Section 4.2.4) and the behavioral properties (see Section 4.3.2) proposed can be used to detect design faults in the DAS adaptive behavior. Thus, the focus of this study was not to measure the effectiveness of the model checking approach. This analysis is presented in Chapter 5.

During the feasibility study, it was identified a fault regarding the *Configuration Correctness* in the Mobile Guide DSPL and a fault regarding the *Interleaving Correctness* in the Car DSPL. Regarding the properties not violated, a manual verification endorsed that they were satisfied in the DSPLs used in the study. There is some threats to the validity regarding the feature models used, the number and type of fault inserted in the Mobiline, and the modeling using the Promela. However, the results of the study indicated that the proposed model checking approach can identify design faults in the DAS adaptive behavior.

It is important to mention that in this feasibility study, it was used only the SPIN model checker to verify the DSPLs adaptive behavior. However, other model checker tools could be used. For example, by using PRISM[4], the software engineer could relate probabilities to the DFTS transitions.

4.4 Testing the DAS Adaptive Behavior

In TestDAS, the final output is a set of test cases to validate if the DAS adaptation mechanism is working correctly. As presented in Chapter 2, a test case is defined by a set of preconditions, inputs and expected results. For testing the adaptive behavior, the test cases shall be derived from the adaptation rules and the possible context states. Thus, first, in this section, the concept of adaptation test case for TestDAS is introduced in Definition 4.4.1.

Definition 4.4.1 (Adaptation Test Case (ATC)) *An adaptation test case is defined by the initial configuration of the adaptive system S_1 that is defined by its active features, a context state C and the expected system configuration S_2, which should satisfy the adaptation rules triggered by the context state C. An ATC aims to determine whether or not the covered adaptive behavior has been implemented correctly in the dynamically adaptive system*

Then, this section presents how adaptation test cases can be generated to validate the DAS adaptive behavior. The key idea is to use the properties specified in the Section 4.3.2 for the DAS model checking as test coverage criteria to guide the DAS testing. For the latter, TestDAS generates sequences of ATCs that have the potential to reveals failures among the system re-configurations.

[4] http://www.prismmodelchecker.org/

Subsection 4.4.1 introduces the concept of test sequence based on the DFTS structure. Next, Subsection 4.4.2 presents a set of test coverage criteria to guide the DAS test cases design. Finally, Subsection 4.4.3 discusses the interactions among the test coverage criteria.

4.4.1 Adaptation Test Sequence

In a DFTS $\mathscr{D} = \langle \mathscr{S}', \mathscr{I}', \mathscr{P}, \mathscr{L}', \rightarrow' \rangle$, a *transition* is a tuple (s_1, Ctx, s_2), where $s_1, s_2 \in S'$ and $Ctx \in P$. Semantically, this transition means that given the current system state s_1, if the context Ctx becomes true, the expected system state is s_2. If the set of active features in s_2 is different from the set of active features in s_1 ($L'_{P_F}(s_2) \neq L'_{P_F}(s_1)$), then this transition triggers an adaptation. Otherwise, it shows that the context Ctx has no effect over the current product configuration. In both cases, a transition from DFTS can be seen as an adaptation test case to validate if the context Ctx makes the product changes from s_1 to s_2 properly. In this case, the steps involved to test a transition (s_1, Ctx, s_2) could be: 1) Set up the DAS configuration according to the active features in the state s_1; 2) Change the context to Ctx; 3) Verify if the DAS configuration is changed to the system state s_2.

However, such single transition test case will not detect faults that are only detectable through exercising sequences of adaptations. For instance, failures in the intern representation of the DAS configuration after a re-configuration (PUSCHEL *et al.*, 2014). Thus, adaptation test sequences are more suitable to find out different kinds of failures in the DAS adaptive behavior. The test sequences also have other advantages: (i) they can save testing resources (e.g., time to set up the tests preconditions) since the tests are performed in sequence; and (ii) they can be used to test potential failures scenarios, for example, the context changing before the effect of an adaptation takes place.

Definition 4.4.2 introduces the concept of test sequence used by TestDAS.

Definition 4.4.2 (Adaptation Test Sequence (ATS)) *Let $\mathscr{D} = \langle \mathscr{S}', \mathscr{I}', \mathscr{P}, \mathscr{L}', \rightarrow' \rangle$ be a DFTS. An n-test sequence is a finite sequence of n system state transitions $s_0 \xrightarrow{ctx_1, \omega_1 : \alpha_1(F_1)} s_1 \xrightarrow{ctx_2, \omega_2 : \alpha_2(F_2)} s_2 \ldots \xrightarrow{ctx_n, \omega_n : \alpha_n(F_n)} s_n$, where $s_i \in S'$ and $ctx_i \models \omega_i$. Testing such sequence means to assess that the active features in the state s_{i+1} are according to the action $\alpha(i)$, triggered by the context ctx_i. When n=1 (1-test sequence), the sequence is the testing of a single DFTS transition.*

Table 6 shows an example of test sequence from the Mobile Guide (see Section 2.1). This is a 6-test sequence that has six adaptation test cases and covers the different context states

defined in the C-KS of this DSPL (see Figure 7). It is worth noting that as ATCs should be executed as a sequence, the expected configuration of a test case is the initial configuration of the next adaptation test case. For instance, the system configuration with the set of features {Mobile Guide, Show Documents, Text, Image, Video} is the expected result of the first ATC and the initial configuration (i.e., precondition) for the second ATC.

Table 6 – Example of Adaptation Test Sequence.

Parameter	Value
Initial Configuration	{Mobile Guide, Show Documents, Text, Image, Video}
New Context	$isBtFull$
Expected Configuration	{Mobile Guide, Show Documents, Text, Image, Video}
New Context	$isBtNormal$
Expected Configuration	{Mobile Guide, Show Documents, Text, Image}
New Context	$isBtLow$
Expected Configuration	{Mobile Guide, Show Documents, Text}
New Context	$isBtLow, hasPwSrc$
Expected Configuration	{Mobile Guide, Show Documents, Text, Image}
New Context	$isBtNormal, hasPwSrc$
Expected Configuration	{Mobile Guide, Show Documents, Text, Image, Video}
New Context	$isBtFull, hasPwSrc$
Final Configuration	{Mobile Guide, Show Documents, Text, Image, Video}

Source – the author.

By using the concept of test sequence, the tester can explore the DAS in different perspectives to achieve a given testing goal. For example, one can look for context states specific faults, to test the DAS with the more likely adaptation sequences or to test sequences where specific use cases should be executable. In the next subsection, a set of five test coverage criteria are described to guide the generation of test sequences independent of the DAS domain.

4.4.2 Test Criteria for DAS testing

In order to support the DAS testing, TestDAS proposes to use the properties defined in Section 4.3.2 to guide the definition of test sequences. Therefore, a test suite can be built to test the DAS in order to verify if there are failures related to the faults already verified in the DAS design during the model checking approach. In the following paragraphs, for each behavioral property, it is defined the related test criteria, as well as examples of test sequences to achieve the criterion based on the Mobile Guides DSPL (see Section 2.1).

Test Sequence for Property 01 - Configuration Correctness. A violation of this property indicates that the DAS achieved an invalid configuration with regards to its feature

model rules. Thus, to test if the product has a **invalid configuration fault**, the adaptation test sequence should cover all the DFTS states, since this modeling describes all possible system configurations according to the context changes of the environment.

Definition 4.4.3 (Configuration Correctness Coverage) *A set TS of adaptation test sequences satisfies the Configuration Correctness Coverage criterion if and only if for all state s' from the DFTS, there is at least one test sequence $ts \in TS$ such that ts covers the sate s'.*

Different strategies can be used by the tester to define a test sequence to satisfy the Configuration Correctness Coverage. In the Algorithm 1, one of these strategies is presented. By following this algorithm, first, the tester must identify the context states of the DAS (line 4) and the initial context state (line 5). This can be done by analyzing the context model of the DAS and the environment where the product will be deployed.

Next, the initial context (`iCtx`) is added to the list of already visited context states (line 6), and it is defined the first system state as this initial context plus the expected product configuration. In Algorithm 1, the resulting product configuration (i.e., the product configuration adapted according to the triggered adaptation rules) is returned by the function $effect$ (line 7), which receives as argument the set R of active rules in a given context and an adaptive system configuration. After that, the created system state (*sysSt*) is appended to the test sequence (line 8). Then, while there are context states not analyzed, the tester should define the following context state that should be used in the tests (lines 10-11), and identify the corresponding system state that should be added in the test sequence (lines 12-14). For the latter, it is used the previous DAS configuration, obtained with the function $features()$ over the last system state created (line 12). The return of Algorithm 1 (line 16) is a test sequence represented by a path $\mathscr{T}$, which contains a sequence of context changes and the expected DAS configurations.

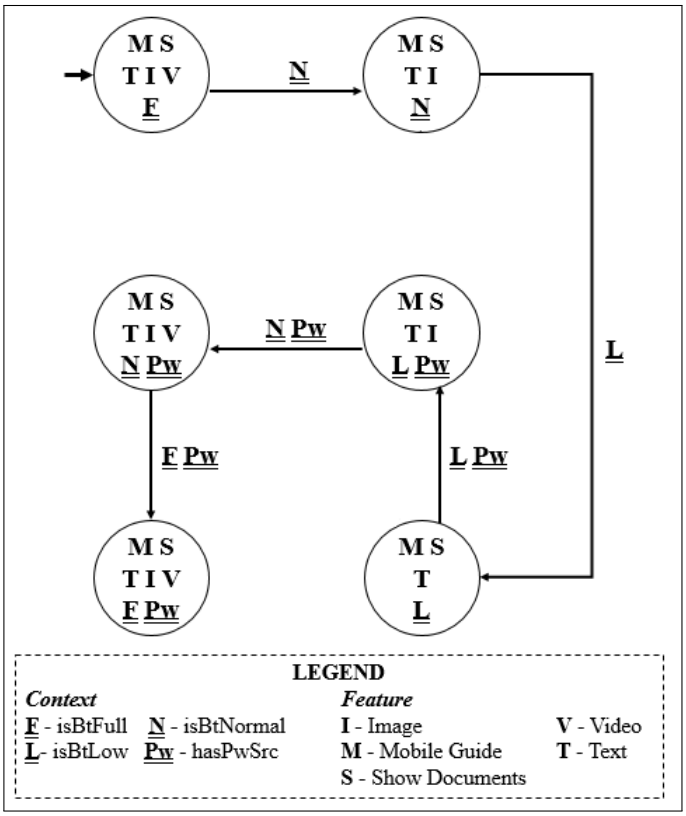
It is important to notice that this coverage criterion can require exhaustive testing. So the tester can apply a "relaxed" Configuration Correctness Coverage criterion, by choosing a prioritization criterion. For instance, instead of covering all DFTS states, the tester can choose the pairwise covering (IEEE, 2015) of the context propositions to reduce the number of context states to be tested.

Figure 17 depicts an example of test sequence to verify if the GREat Tour (LIMA *et al.*, 2013), which is a product from the Mobile Guides DSPL (see Section 2.1), achieves some invalid configuration during its adaptations process. Note that such test sequence covers all states of the DFTS created to this DAS (see Section 2.2.1).

Algorithm 1 Test Sequence Specification for Property 01.

```
1:  function TESTSEQUENCE_PROP1(E)
2:      Path 𝒯 = ∅
3:      Set VisitedCtx = ∅
4:      Identify the set 𝒞𝒮 of context states of the DAS
5:      Define the initial context state iCtx
6:      VisitedCtx = VisitedCtx ∪ {iCtx}
7:      sysSt = iCtx ∪ effect(R_iCtx, E)
8:      𝒯.append(sysSt)
9:      while (𝒞𝒮 \ VisitedCtx) ≠ ∅ do
10:         Define the next context state nCtx from 𝒞𝒮
11:         VisitedCtx = VisitedCtx ∪ {nCtx}
12:         nSysSt = nCtx ∪ effect(R_nCtx, features(sysSt))
13:         𝒯.append(nSysSt)
14:         sysSt = nSysSt
15:     end while
16:     return 𝒯
17: end function
```

Figure 17 – Example of Test Sequence for Property 01



Source – the author.

Test Sequence for Property 02 - Rule Liveness. A violation in the Rule Liveness property indicates rules that are never triggered. For instance, this can occur in reason of a fault in the context guard definition. Aiming to test if the product has the **dead predicate fault**, the

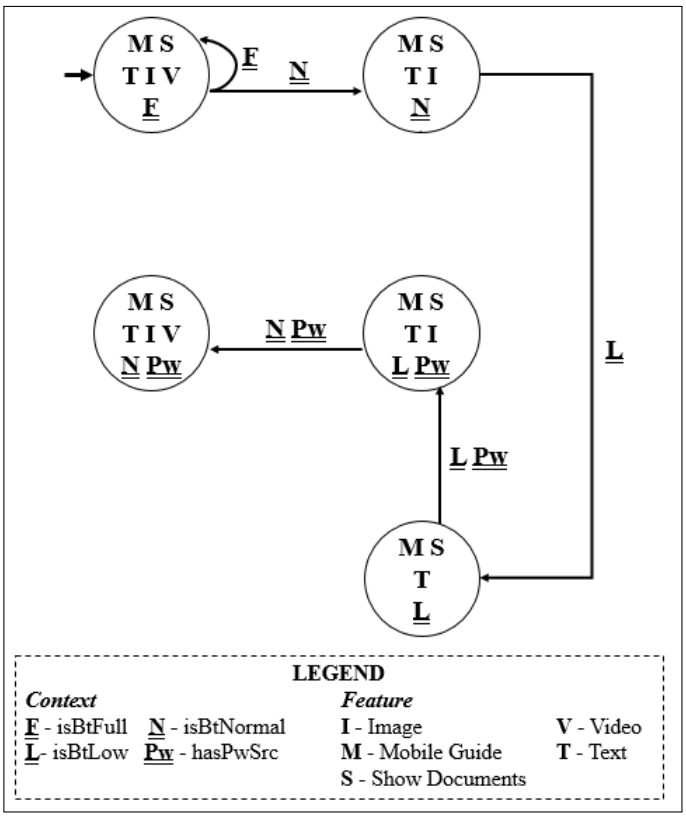
n-test sequence should cover the context triggers of all adaptation rules of the DAS under testing.

Definition 4.4.4 (Rule Liveness Coverage) *A set TS of adaptation test sequences satisfies the Rule Liveness Coverage criterion if and only if for all context guard conditions ω from the DAS adaptation rules, there is at least one test sequence $ts \in TS$ such that ts covers at least one context state that satisfies the guard condition ω.*

Again, different strategies can be used by the tester to generate a test sequence satisfying this criterion. One possibility is to adapt the Algorithm 1 by limiting the context states of the set $\mathscr{CS}$ to those that trigger some adaptation rule of the DAS.

Figure 18 depicts an example of test sequence to verify if some adaptation rule of the GREat Tour is not triggered when it should be. Note that the goal here is to validate that when the context condition of one adaptation rule of the Mobile Guide (see Table 1) is satisfied, this rule triggers its adaptation actions. For instance, in Figure 18, from the first state when the context is `BtNormal` (N), the adaptation rule AR03 (see Table 1) is triggered and, then, the feature *Video* is expected to be deactivated in the next state.

Figure 18 – Example of Test Sequence for Property 02



Source – the author.

Test Sequence for Property 03 - Interleaving Correctness. A violation of the Interleaving Correctness property indicates that the interaction among active adaptation rules provides unexpected results. In order to test if the product has the **nondeterministic interleaving fault**, the test sequence should cover all states of the C-KS that trigger more than one adaptation rule.

Definition 4.4.5 (Interleaving Correctness Coverage) *A set TS of adaptation test sequences satisfies the Interleaving Correctness Coverage criterion if and only if for all context guard conditions ω from the DSPL adaptation rules that trigger more than one adaptation rule, there is at least one test sequence ts such that ts covers at least one context state that satisfies the guard condition ω.*

As can be observed in Table 1, the Mobile Guide DSPL has not adaptation rules that can be triggered at the same time, since all its rules have a different context guard condition. If, however, there was a rule AR06 with the context condition equals to the AR05's context condition ($isBtFull$), then the test sequence would cover a context state that satisfies this condition.

In order to obtain a test sequence for achieving the Interleaving Correctness Coverage criterion, it is possible to adapt the Algorithm 1 by limiting the context states of the set $\mathscr{CS}$ to those that trigger more than one adaptation rule of the DAS.

Test Sequence for Property 04 - Feature Liveness. A violation in the Feature Liveness Property indicates a context-aware feature that is not activated in the product. To test if the DAS has the **dead feature fault**, the test sequence should cover at least one transition for each context-aware feature, where its state changes from deactivated to activated.

Definition 4.4.6 (Feature Liveness Coverage) *A set TS of adaptation test sequences satisfies the Feature Liveness Coverage criterion if and only if for all context-aware features f from the DAS, there is at least one test sequence ts such that ts triggers the activation of the feature f from a state where this feature was deactivated.*

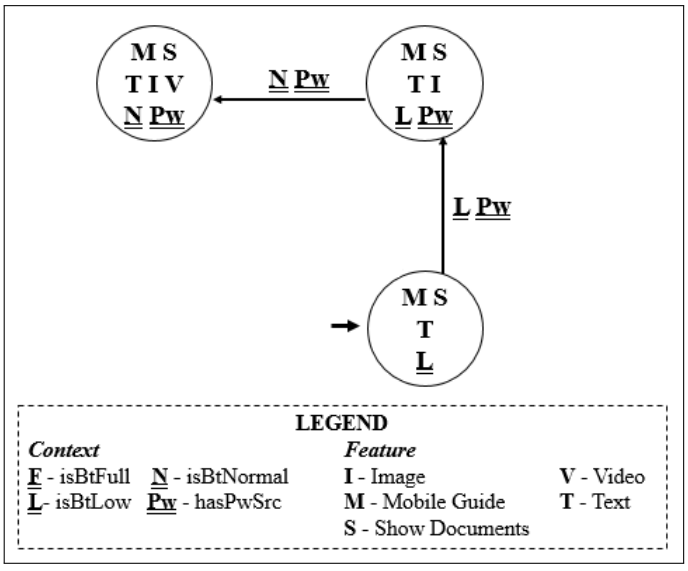
Algorithm 2 presents a strategy to generate a test sequence for this criterion. The first step is to identify the context-aware features (line 3). Next, for each context-aware feature f_i, the algorithm appends in the path $\mathscr{T}$ a system state `sysSt` (lines 6-11), where the feature f_i is deactivated. Next, it identifies another system state `nSysSt` (lines 13-14), where the feature f_i is activated by the action of some adaptation rule. Lastly, the algorithm returns the test sequence $\mathscr{T}$ (line 17).

Algorithm 2 Test Sequences Generation for Property 04.

```
1: function TESTSEQUENCE_PROP4(𝓕𝓜, ℛ)
2:     Path 𝒯 = ∅
3:     Identify the set 𝒞𝒜ℱ of context aware features of the 𝓕𝓜
4:     for all feature f_i ∈ 𝒞𝒜ℱ do
5:         if 𝒯 = ∅ then
6:             Define a valid system state sysSt where f_i ∉ features(sysSt)
7:             𝒯.append(sysSt)
8:         end if
9:         if f_i ∈ features(sysSt) then
10:            Define a system state sysSt2 where f ∉ features(sysSt2)
11:            𝒯.append(sysSt2)
12:        end if
13:        Identify a system state nSysSt that satisfies the context condition of one adaptation
       rule that activates f_i
14:        𝒯.append(nSysSt)
15:        sysSt = nSysSt
16:    end for
17:    return 𝒯
18: end function
```

Figure 19 depicts an example of test sequence to verify if in the GREat Tour the features are activated as expected. In this example, with one test sequence is possible to test if both features *Image* and *Video* are activated in some context state. Note that the order of the adaptation test cases is important to ensure that the DAS changes from a configuration where *Video* and *Image* are deactivated to another configuration where they are activated.

Figure 19 – Example of Test Sequence for Property 04



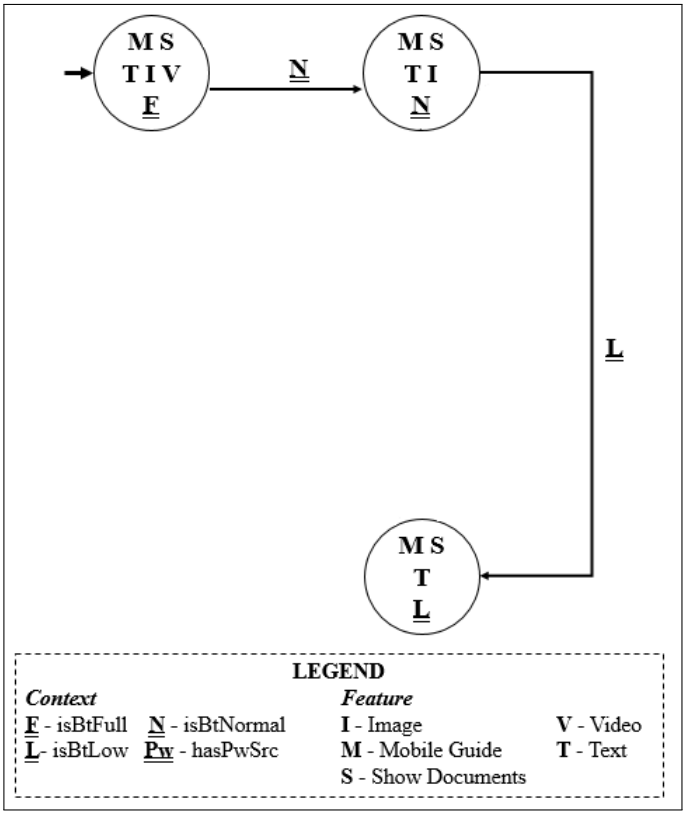
Source – the author.

Test Sequence for Property 05 - Variation Liveness. A violation in Variation Liveness property indicates that a context-aware feature is not deactivated in the product as expected. In order to test if the product has the **false variable feature fault**, the test sequence should cover at least one transition for each context-aware feature, where its state changes from activated to deactivated.

Definition 4.4.7 (Variation Liveness Coverage) *A set TS of adaptation test sequences satisfies the Variation Liveness Coverage criterion if and only if for all context-aware feature f from the DAS, there is at least one test sequence ts such that ts triggers the deactivation of the feature f from a state where this feature was activated.*

Figure 20 depicts an example of adaptation test sequence to verify if in the GREat Tour the context-aware features are deactivated as expected. In this case, the first configuration state has both features *Image* and *Video* activated. Next, an adaptation test case validates that the feature *Video* is deactivated when the context is `isBtNormal`. After that, there is an ATC to test if the feature *Image* is deactivated as expected when the context is `isBtLow`.

Figure 20 – Example of Test Sequence for Property 05



Source – the author.

Regarding the algorithm to generate an adaptation test sequence for the Variation Liveness Coverage criterion, it can be defined in a similar way to Algorithm 2. The difference is that for this criterion, the algorithm should focus on the transitions from states where the context-aware feature is activated to states where it is deactivated.

4.4.3 Interactions among Test Coverage Criteria

The coverage percentage of the test criterion proposed in this section is measured by the Formula 2.3, which uses the number of test coverage items covered by the test cases, and the total number of test coverage items identified. For instance, for the criterion Rule Liveness Coverage, the adaptation test sequence should cover the context guard conditions of each adaptation rule. In the running example (see Section 2.1), there are five adaptation rules and, thus, five test coverage items. If the test sequence covers all these items, the criterion coverage is 100%; otherwise, the percentage is proportional to the number of adaptation rules that have been covered. Thus, the tester can combine the test criteria proposed and choose the desired coverage percentage for each criterion to satisfy him/her needs.

It is also important to note that the coverage of one test criterion can affect the coverage of another criterion. This relationship among two test coverage criteria A and B can be classified into three types: (i) *Strong*, a test criterion A with a *Strong* relationship with a criterion B means that a full (100%) coverage of A will leads to a full coverage of B; (ii) *Medium*, a test criterion A with a *Medium* relationship with a criterion B means that a full coverage of A does not ensure a full coverage of B; and (iii) *Weak*, a test criterion A with a *Weak* relationship with a criterion B means that the coverage of A often has a few impact or does not have impact in the coverage of B.

Based on this classification, Table 7 presents the relationships among the test coverage criteria proposed. It is worth noting that the relationship between two criteria is not mutual, that is, the relationship type from the criterion A to the criterion B could not be the same from B to A. For instance, the Configuration Correctness Coverage has a *Strong* relationship with Rule Liveness Coverage, but the latter has only a *Medium* relationship with Configuration Correctness.

As presented in Table 7, the criterion Configuration Correctness Coverage is the unique that has a *Strong* relationship with other criteria. Since it covers all DFTS states, its full coverage will leads to the full coverage of the Rule Liveness Coverage and Interleaving Correctness Coverage.

Table 7 – Usual relationship among the test criteria

Test Criterion	Configuration Correctness	Rule Liveness	Interleaving Correctness	Feature Liveness	Variation Liveness
Configuration Correctness	-	Strong	Strong	Medium	Medium
Rule Liveness	Medium	-	Medium	Medium	Medium
Interleaving Correctness	Weak	Weak	-	Weak	Weak
Feature Liveness	Medium	Medium	Weak	-	Weak
Variation Liveness	Medium	Medium	Weak	Weak	-

Source – the author.

However, the criterion Configuration Correctness Coverage does not determine the order of the adaptation test cases in the adaptation test sequence, but only which (all) DFTS states should be covered by the test sequence. Thus, the full Configuration Correctness Coverage should cover some test coverage items of both criteria Feature Liveness and Variation Liveness, but it does not ensure their full coverage.

With regards to the Rule Liveness Coverage, it has a *Medium* relationship with all other criteria. It covers some of the DFTS states, but cannot ensure the coverage of at all (e.g., different DFTS states reached by different contexts that trigger the same adaptation rule). It covers the interleaving of rules that have the same context guard condition. However, it does not ensure the coverage of the interleaving of those rules that have different context conditions that can be true at the same time. Also, this coverage criterion does not specify the order of the adaptation test cases and, thus, its full coverage cannot ensure the full coverage of both Feature Liveness and Variation Liveness.

The Interleaving Correctness Coverage criterion has a *Weak* relationship with all other criteria. This test criterion deals with the coverage of the context conditions that trigger more than one adaptation rule. Thus, any adaptation rule that is not triggered with another rule at the same time is not covered, as well as the DFTS states generated by these rules. In addition, besides not specifying the order of the adaptation test cases, the Interleaving Correctness Coverage criterion cannot cover the (de)activation of features triggered by rules that have not the same context guard condition of other rules.

The criteria Feature Liveness Coverage and Variation Liveness Coverage have a *Weak* relationship with each other because they have opposite purposes. The first one deals with the features activation, whereas the other deals with feature deactivation. Since these criteria cover the activation (Feature Liveness) and deactivation (Variation Liveness) of all context-aware features, they cover some adaptation rules and achieve some states of the DFTS. Thus, these

criteria have a *Medium* relationship with the Rule Liveness and Configuration Correctness. On the other hand, with regards to the Interleaving Correctness Coverage, both criteria have a *Weak* relationship because their full coverage only ensures to cover the interleaving of rules that have the same context condition guard, and only if these rules are the unique available for the activation or deactivation of some context-aware feature.

It is important to highlight that Table 7 describes a common relationship among the test coverage criteria proposed that should be valid in the most of the DAS, but that can be different in some cases. For instance, given a DAS in which all adaptation rules can be interleaved, then the full Interleaving Correctness Coverage will leads to the full Rule Liveness Coverage. In this case, the relationship between the Interleaving Correctness Coverage and Rule Liveness Coverage is *Strong*. Therefore, it is important to analyze the DAS under testing to eventually adjust the relationships depicted in Table 7.

4.5 Supporting Tools

Another contribution of this thesis is the development of a tool and a library for supporting the DAS model checking and testing according to the TestDAS method presented in Figure 13.

The tool and the library are called TestDAS Tool and CONTroL (CONtext variability based software Testing Library), respectively. Both are Java applications and were developed by using the Eclipse IDE[5]. The first one supports the DAS model checking and the generation of adaptation test sequences according to the test criteria proposed in Section 4.4. The other one supports the test sequences execution in the DAS under testing.

The following subsections present details about the functionality, interfaces, design and limitations of the TestDAS tool (Subsection 4.5.1) and the CONTroL (Subsection 4.5.2).

4.5.1 TestDAS tool

The TestDAS tool is a Java application that helps the user (Software Engineer/Tester) to perform the model checking according to the approach described in Section 4.3, and the DAS testing following the test coverage criteria depicted in Section 4.4.

In summary, the TestDAS tool receives as input the DAS feature model and produces an excel file with the context states in an adjacency matrix. Next, the user fills the matrix with the

[5] https://www.eclipse.org/

context variation (see Section 2.2.1). By using this matrix and the feature model, the TestDAS tool generates then a DFTS, provides an interface for the DAS model checking and exports a test sequence in the JavaScript Object Notation (JSON) format [6].

This tool was developed by using the following third-party APIs: (i) JExcelApi[7], which is a Java library for reading/writing Excel (ii) Gson[8], which is a Java serialization/deserialization library to convert Java Objects into JSON and back; and (iii) Sat4j[9] that is the a library for boolean satisfaction and optimization in Java. The latter was used to identify the contexts that can satisfy the Interleaving Correctness Coverage criterion (see Section 4.4.2).

The next subsection describes the overview of this tool regarding its package diagram. After that, the functionality and main user interfaces are presented in Subsection 4.5.1.2. Then, the limitations of the TestDAS tool are summarized in Subsection 4.5.1.3.

4.5.1.1 Tool Overview

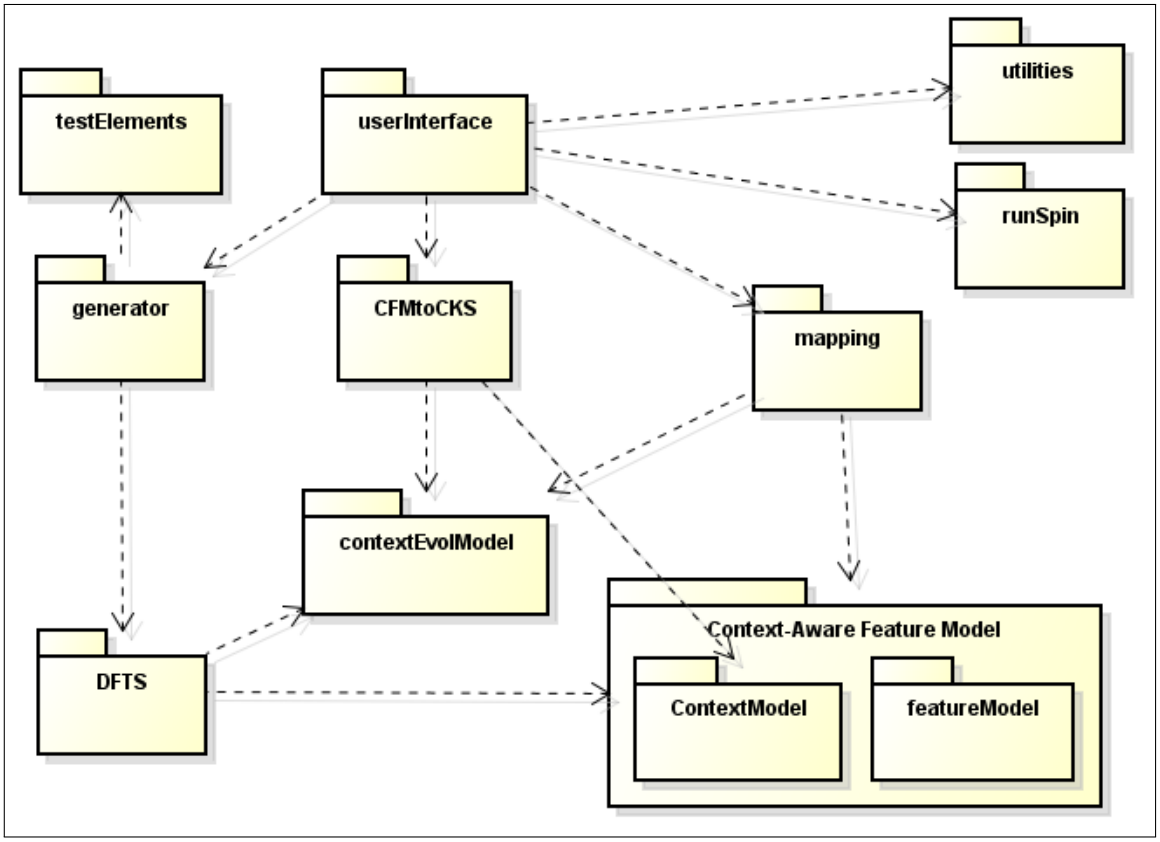
Figure 21 presents the package diagram of the TestDAS tool. As depicted in this figure, this tool is composed by the following packages: (i) contextEvolModel; (ii) contextModel; (iii) featureModel; (iv) CFMtoCKS; (v) mapping; (vi) runSPin; (vii) DFTS; (viii) testElements; (ix) generator; (x) userInterface; and (xi) utilities.

The packages *contextEvolModel*, *contextModel* and *featureModel* are related to the input of the TestDAS tool. The package *contextEvolModel* contains the classes that represent the context variation model (see Section 2.2.1). In the *contextModel* package are the classes that represent the context variability, whereas the package *featureModel* has the classes that represent the feature variability. The classes of both packages compose the DAS feature model.

The package *CFMtoCKS* has two classes: CtxStateExtractor and CKSExtrator. The first one is in charge of extracting from the context-aware feature model the context states to generate an excel file with an adjacency matrix. In this matrix, the columns and rows refer to the context states, that is, the valid combinations of contexts. The purpose of this matrix is to allows the user to specify the possible transitions among the context states (see Section 2.2.1). The class CKSExtractor, in turn, reads the generated *.xls* file to identify the relationships among the context states, and creates internally the Context Kripke Structure of the DAS under testing.

[6] http://www.json.org
[7] http://jexcelapi.sourceforge.net/
[8] https://github.com/google/gson
[9] http://www.sat4j.org/

Figure 21 – Package Diagram of the TestDAS tool



Source – the author

The role of the classes in the package *mapping* is to generate the *.pml* file [10] with the DFTS and the behavioral properties according to the context variation model, feature model and context model. The classes in the package *runSpin* are in charge of running the Windows prompt command and execute the SPIN tool to check the behavioral properties over the DFTS.

The package *DFTS* contains the classes to generate the DFTS from the context-aware feature model and the C-KS, and the classes to represent the DFTS in a graph. This representation in the form of a graph was used to support the generation of the test sequences.

The package *testElements* has the classes that represent an adaptation test sequence and the adaptation test cases. The classes in the package *generator* generate the test sequences according to the test coverage criteria proposed in Section 4.4.2.

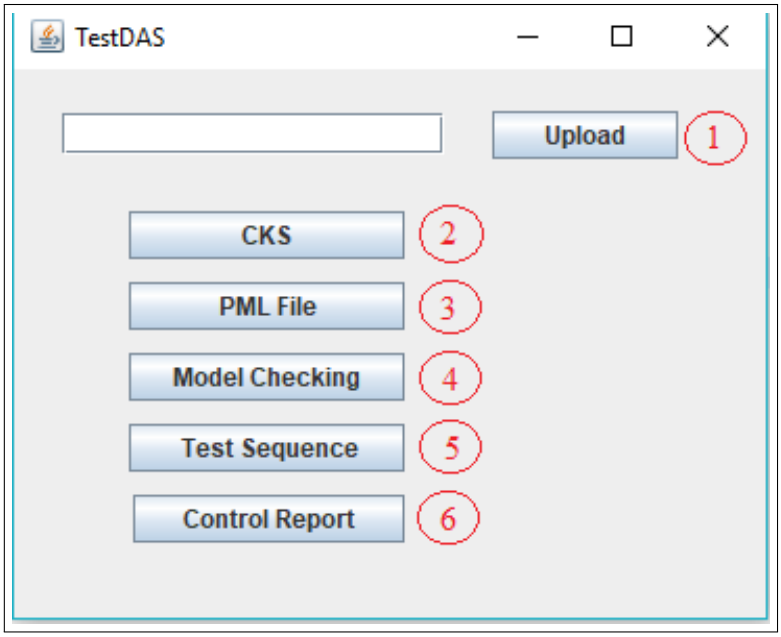
The package *userInterface* contains the user interfaces built to support the user interaction with the functionality provided by the TestDAS tool. Lastly, the package *utilities* gathers the classes with common functions such as generating and reading files.

[10] ProMeLa specification file

4.5.1.2 Functionality

Figure 22 presents the initial user interface of the TestDAs tool. It is worth noting that the TestDAS tool does not support the DAS feature modeling. Then, this tool receives as input the context-aware feature model that should be upload (1) through a JSON file. So, the TestDAS tool is independent of a feature modeling tool since the unique constraint is that the feature model should be parsed into a JSON file following the class diagram presented in Appendix C.

Figure 22 – Initial Screen of the TestDAS tool



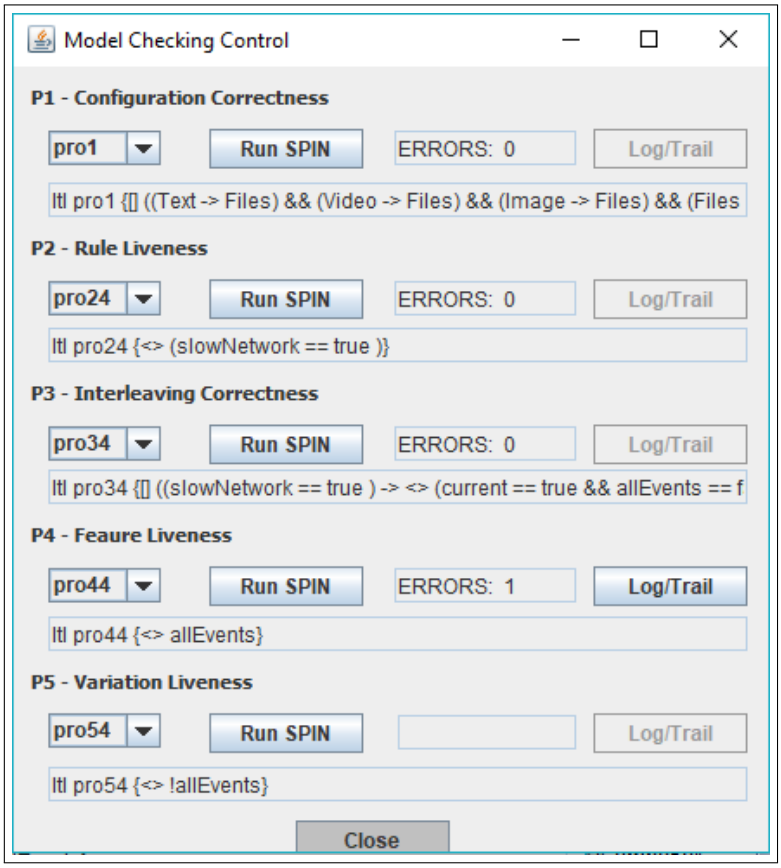
Source – the author

Since the feature model is uploaded, the user can generate the .xls file with the context states by clicking on the option *CKS* (2). This .xls file has the adjacent matrix in which the user should fill the allowed transitions among the context states. After that, the user should click on the option *PML file* (3) that receives as input the .xls file with the C-KS filled in the adjacent matrix and generates the Promela file with the DFTS and the behavioral properties presented in Section 4.3.2.

At this moment, the software engineer can use the Promela file generated and run it with the SPIN model checker tool using the Windows prompt command (see Section 4.3) or it can click on the option *Model Checking* (4) to use the interface created to support the DAS model checking. Figure 23 presents this user interface. In such interface, the user can select the

behavioral property and run it with the option “Run SPIN”. If an error is found, the application also gets the SPIN logs and shows through the option *Log/Trail*.

Figure 23 – Model checking screen of the TestDAS tool



Source – the author

The advantage of the model checking directly from the Promela source is that the user can specify other properties to be checked beyond the properties specified automatically. On the other hand, the advantage of the user interface depicted in Figure 23 is that it makes easier to apply the model checking, especially to who has not experience on model checking (see the results of the evaluation of this tool in Chapter 5).

The *Test Sequence* option (5) generates the test sequences using as input the context-aware feature model and context variation model. Figure 24 presents an example of test sequence generated from the running example (see Section 2.1). This test sequence has six adaptation test cases. Each test case has: (i) a test case ID; (ii) the ID of the DFTS transition covered; (iii) the context state specified by the context features that should be active; and (iv) the expected features that should be activated when the context state is true.

Figure 24 – Example of Test Sequence generated by TestDAS

Source – the author

It is worth noting that for the sake of simplicity, the adaptation test cases in Figure 24 shows only the context-aware features that are required to be activated. In this way, the features not depicted in the block *expectedFeatureStatus* should be deactivated. Similarly, in Figure 24 are depicted only the context features that are true in the context state of the test case, meaning that the other context features of the context-aware feature model should be false.

The option *Control Report* (6) is used to support the generation of the report from the CONTroL. It receives as input a JSON file generated by the CONTroL with the test cases results and generates a `.html` file with the results. More details about this report are presented in Subsection 4.5.2.

4.5.1.3 Limitations

The TesTDAS tool was developed for supporting the use the TestDAS method. However, it has some limitations as described as follows:

- The tool only accepts context-aware feature model converted in a JSON file following the structure depicted in Appendix C. This makes the input more generic since it does not

depend on a specific modeling tool. Nevertheless, the user must create a parser to generate the JSON file from a context-aware feature model specified in him/her feature modeling tool;

- The TestDAS tool does not deal with feature attributes or feature cardinality that are presented in cardinality-based feature models (BENAVIDES *et al.*, 2010);
- At this moment, the TestDAS tool only supports the SPIN model checker tool. The choice of this tool was made because it is widely known by the model checking community (HOLZMANN, 2003; SHARMA, 2013);
- The tool does not support logic formulas with the context features, that is, a combination with AND or OR among the context features to trigger the (de)activation of features. In the current version, the software engineer/tester should specify only one context feature as the context guard condition.

4.5.2 CONTroL

CONTroL is a Java application that aims to support the user (Tester) to execute test sequences in the DAS under testing. In summary, it receives as input a JSON file with test sequences, controls the context captured by application during its execution, monitors the features (de)activation and generates an HTML file with the test results. It is important to highlight that the CONTroL requires the instrumentation of the DAS under testing. This is needed to allow CONTroL to use Aspect-Oriented Programming (KISELEV, 2002) to intercept the execution of the DAS under testing in order to control the context and monitor the feature status.

With regards to the CONTroL implementation, it was made by using the following third-party APIs: (i) Jackson[11], which is a Java library to convert Java Objects into JSON and back; (ii) Extent[12], which is a java library to support the creation of test reports in HTML; and (iii) Aspectj [13], which is an aspect-oriented extension to the Java programming language.

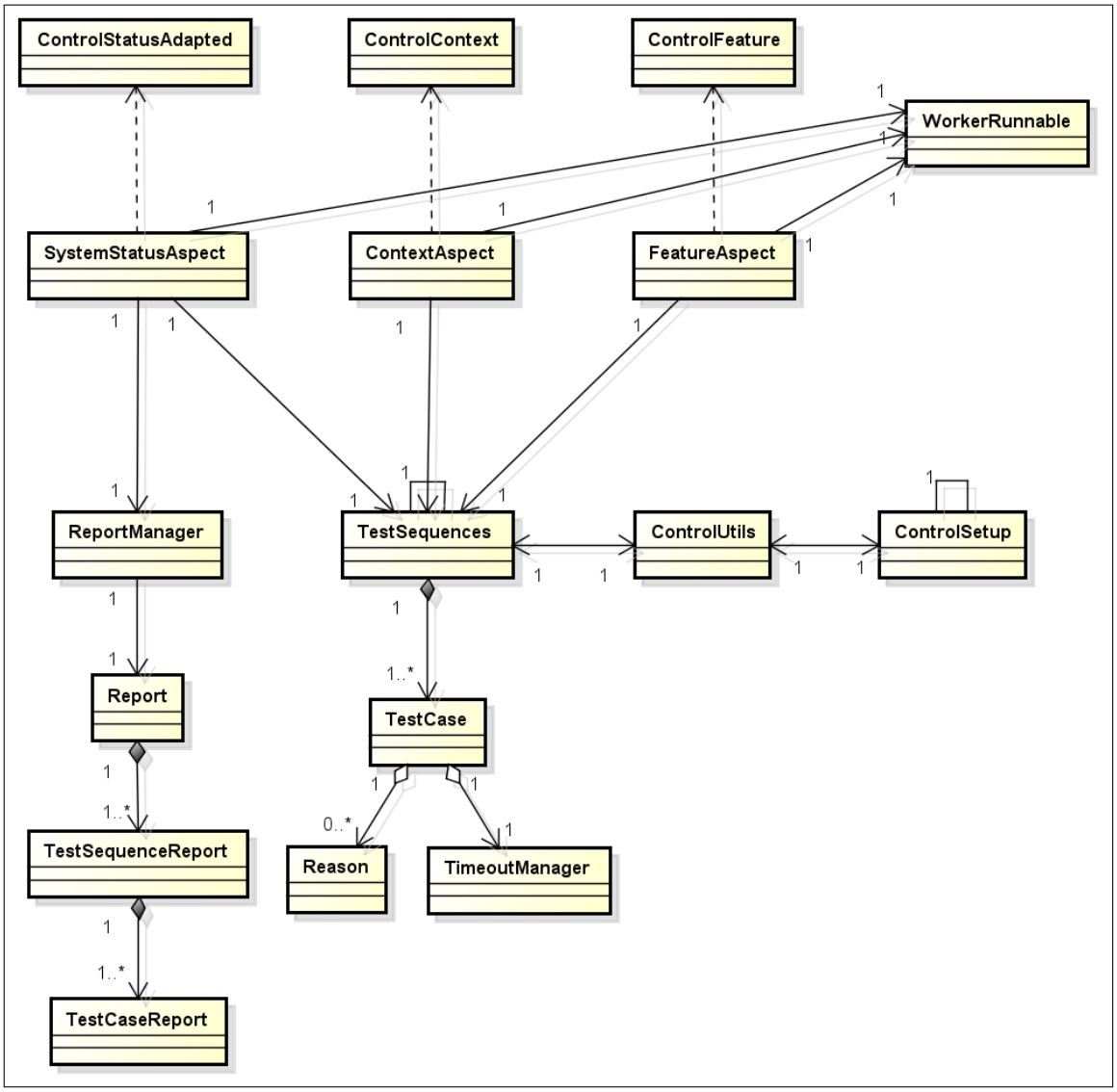
The next subsection describes the overview of this tool regarding their class diagram. After that, the functionality and an example of test report are presented in Subsection 4.5.2.2. Finally, the limitations of the CONTroL are summarized in Subsection 4.5.2.2.

[11] https://github.com/FasterXML/jackson
[12] http://extentreports.com
[13] https://www.eclipse.org/aspectj/

4.5.2.1 *Library Overview*

Figure 25 presents the class diagram of the CONTroL. The use of this library is based on three Java annotations defined by the classes: *ControlSystemAdapted*, *ControlContext* and *ControlFeature*. The class *ControlContext* is used to annotate the methods of the DAS under testing that are in charge of capturing the context values. The class *ControlFeature* is used to annotate the methods related to the status of the DAS features, that is, its (de)activation. The class *ControlSystemAdapted*, in turn, should be used to annotate the methods in charge of performing the DAS adaptations based on the context changes.

Figure 25 – Class Diagram of the CONTroL



Source – the author

These annotations are used by the aspects implemented in the classes *ContextAspect*,

FeatureAspect and *SystemStatusAspect*. The aspect *ContextAspect* uses the annotation *ControlContext* to create a *pointcut* to intercepts the methods that capture the context and change the context values according to the test sequence in execution. The aspect *FeatureAspect* uses the annotation *ControlFeature* to create a *pointcut* to intercept the methods that change the feature status and store the feature current status in the test sequence in execution. The aspect *SystemStatusAspect* uses the annotation *ControlSystemAdapted* to create a *pointcut* to intercept the method in charge of the DAS adaptations and verify the test case results by comparing the expected feature status with the real feature status. These results are stored in an instance of the classe *Test Sequence*. All these three aspects use the class *WorkerRunnable* to perform operations over the intercepted methods by using threads.

The class *TestSequence* contains a list of *TestCase* and stores the current test case in the execution. The class *TestCase* has attributes to indicate the context state, expected features, actual features and test result. These last two attributes are defined at runtime by the aspects *ControlFeature* and *ControlSystemAdapted*, while the context state is read during the CONTroL execution by the aspect *ControlContext*. Moreover, the class *TestCase* has one object of the class *TimeoutManager* to monitor if the test case execution spent more time than the timeout limit and zero ou more objects from the class *Reason*, which is related to the warnings that could be triggered during the tests (e.g., an warning informing that a given context was not ready from the test sequence during the tests).

With regards to the report generation, there are the following classes: (i) *ReportManager*, which managers the test results; (ii) Report that represent an instance of the test report; (iii) *TestSequenceReport* that gathers all instances of *TestCaseReport*; and (iv) *TestCaseReport*, which stores the warnings (from the class *Reason*), the test results (*i.e.*, passed, failed, warning) and other information (*e.g.*, list of features that should be activated but were not).

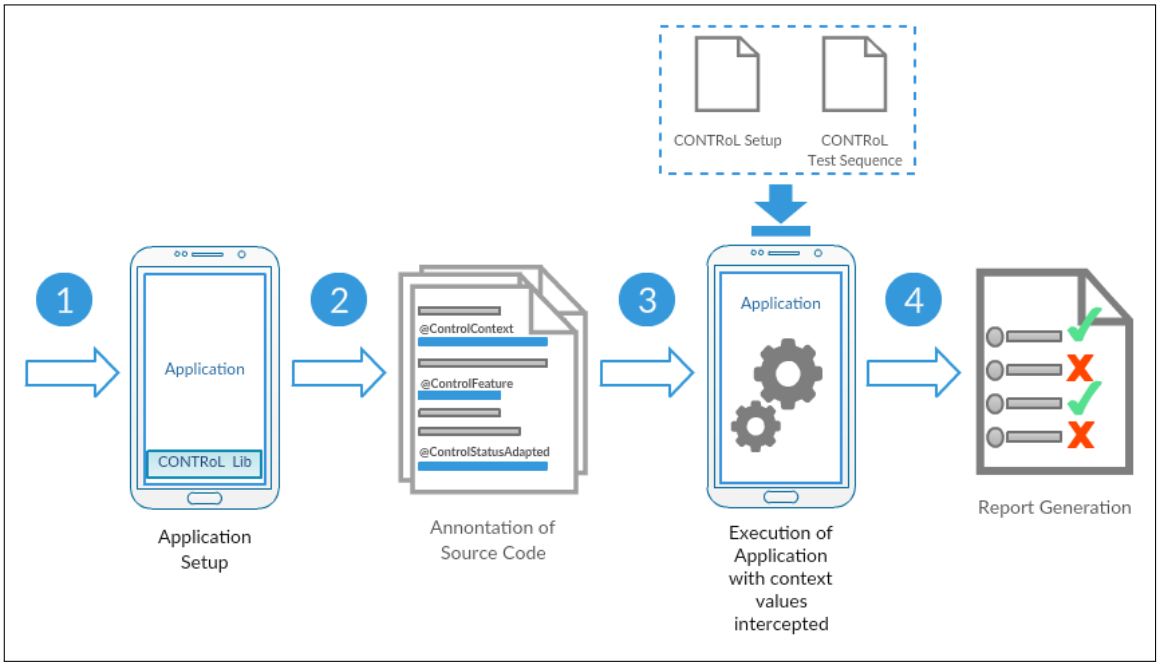
Aiming to support the CONTroL execution there are yet the class *ControlUtils* with methods to read and write files, and the class *ControlSetup* that stores the location of the test sequence, the duration of the timeout and the directory to save the test results.

4.5.2.2 Functionality

Figure 26 presents an overview of the CONTroL execution. First, the Tester should add the CONTroL library in the source code of the application. Next, he/she should instrument the code by putting the annotations *ControlContext*, *ControlFeature* and *ControlSystemAdapted*

in the methods of the application under testing for context capture, changing the feature status and adapting the application, respectively.

Figure 26 – Overview of the CONTroL



Source – the author

Figure 27 presents the code of the running example (see Section 2.1) instrumented with the annotations before mentioned. As depicted in this figure, both the annotations *Control-Context* and *ControlFeature* has one parameter. In the first annotation, the user should specify the name of the context feature, whereas in the second one it should be specified the feature name.

After the code instrumentation, the user should add in the device a folder called "control" with two files: one (CONTroL setup) with information about the location of the test sequence and the directory to save the tests results; and a JSON file with the test sequence. Therefore, when the DAS under testing starts its execution, the CONTroL starts to control its context and to observe its feature (de)activation based on the annotations. The test execution ends when the CONTroL performs all the test sequence. Based on the test results, the CONTroL generated a HTML report using the Extent library [14]. Figure 28 presents an example of the report generated. For each test sequence it shows the result (passed, failed, warning) and by clicking on the test sequence, it is possible to see the results of each adaptation test case.

[14] http://extentreports.com

Figure 27 – Example of code annotated. (A) class VideoFeature; (B) class BatteryContext; (C) class ContextManager

```
ContextManager mContextManager;

public VideoFeature() {
    mContextManager = ContextManager.getInstance();

    super.addReqContext(ContextManager.ContextTypes.NETWORK);
    super.addReqContext(ContextManager.ContextTypes.BATTERY);

    super.addAffectedActivity(FilesActivity.class.getSimpleName());
    super.addAffectedActivity(VideoActivity.class.getSimpleName());
}

@ControlFeature(feature = "video")
public boolean isFeatureActivated() {
    return super.isActivated();
}

@Override
public void verifyFeature() {
    NetworkContext networkContext = mContextManager.getNetworkContext();
    BatteryContext batteryContext = mContextManager.getBatteryContext();
    String currentActivity = mContextManager.getNameCurrentActivity();
```

A

```
    @ControlContext(contextName = "isBatteryLow")
    public boolean isLevelLow() {
        if (getmBatteryLevel() == BatteryLevels.LOW) {
            return true;
        }
        return false;
    }
```

B

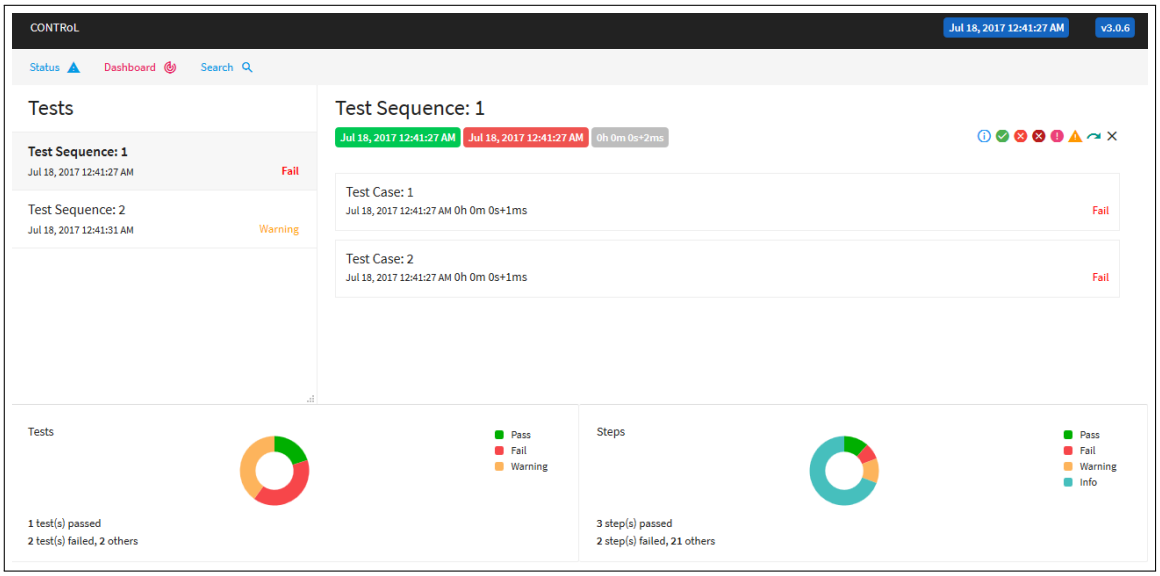
```
private Runnable mContextRunnable = new Runnable() {
    @ControlStatusAdapted
    @Override
    public void run() {
        Log.d(ContextManager.class.getSimpleName(), "HANDLER DE VERIFICAÇÃO INICIADO.");

        for (DeviceContext deviceContext : mContextList) {
            if (deviceContext.updateContextInfo(mAppContext)) {
                Log.d(ContextManager.class.getSimpleName(), "ALGUM CONTEXTO MUDOU.");
                deviceContext.notifyObservers();
            }
        }
        if (mHandler != null) {
            mHandler.postDelayed(mContextRunnable, 3000);
        }
    }
};
```

C

Source – the author

Figure 28 – Example of report generated by the CONTroL



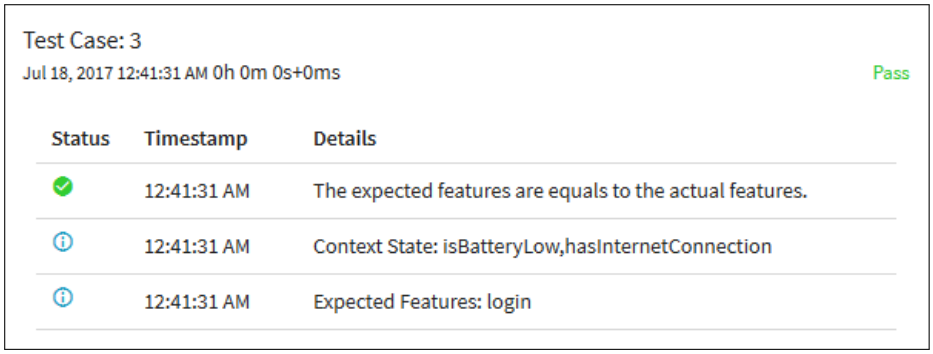
Source – the author

Figure 29 presents the results of a test case that has passed. In this case, the report presents only the context state, the expected activated context-aware features and a message stating that the expected features matched with the actual activated features in the DAS under testing.

Figure 30 presents the results of a test case that failed during the execution. In this case, besides information about the test case (context state and expected features), the report presents other information: (i) *Activated Feature*, which is the list of actual activated features; (ii) *Deactivate Expected Features*, which is the list of expected features that during the test execution

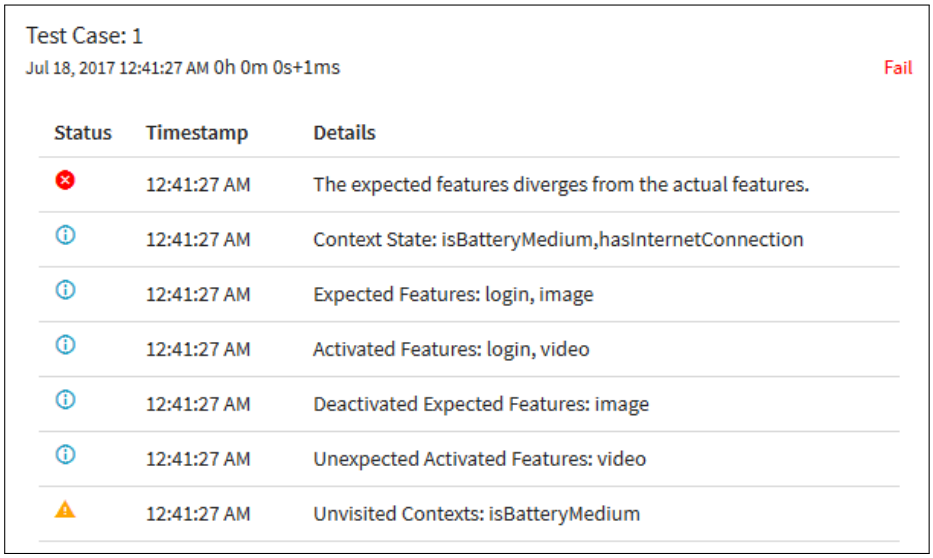
were deactivated when should not; (iii) *Unexpected Activated Features* that shows the features activated that were not expected by the test case; and (iv) *Unvisited Contexts*, which presents the contexts that were in the test sequence, but were not used during the test execution.

Figure 29 – Example of a passed test case in the report generated by the CONTroL



Source – the author

Figure 30 – Example of a failed test case in the report generated by the CONTroL



Source – the author

4.5.2.3 Limitations

The limitations of the CONTroL use are described as follows:

- Currently, this library does support the testing of situations in which the context changes during the system adaptation;
- The library requires the annotation of methods for context capture, feature (de)activation

and the adaptation process itself. This could require some refactoring of the source of the application under testing;

- At this moment, CONTroL has support only for Java and Android applications, and requires the set up of the DAS to use AspectJ (KISELEV, 2002); and
- CONTroL supports only contexts that can be represented by Boolean, Integer, Double or String variables.

4.6 Conclusion

In this chapter, TestDAS, which is the method proposed for testing the DAS adaptive behavior, was presented. Besides the generation of tests, this method involves a model checking approach to identify design faults. Therefore, within the TestDAS method, this chapter presented: (i) a model for the DAS adaptive behavior, called Dynamic Feature Transition System (DFTS); (ii) An approach to model checking the DAS design; (iii) A set of test coverage criteria to guide the DAS testing; and (iv) a supporting tool and library.

DFTS is a formalism, also proposed in this work, to model the adaptive behavior of DAS and that represent the feature (de)activation and the context changes. This model describes the snapshots of context and system features in a given instant of time. Thus, it supports the identification of design faults and adaptation failures related to the adaptation rules of dynamically adaptive systems. For the DAS model checking, it was presented how to map the concepts of the DFTS to Promela, which is the language of the SPIN model checker tool. Besides that, a set of behavioral properties was presented to support the identification of design faults in the DAS adaptation logic specified by the adaptation rules.

In order to support the tests generation, a set of test coverage criteria was created based on the behavioral properties proposed to the DAS model checking. Based on these criteria and in the DFTS of the DAS under testing, TestDAS generates adaptation test sequences. Also, to support the use of the method proposed, a tool called “TestDAS tool” was developed to support the DAS model checking and the generation of test sequences. Furthermore, a library called CONTroL was implemented to support the execution of test sequences in Android applications.

The next chapter presents the details and results of the assessment of TestDAS. This evaluation was focused on the effectiveness of both the model checking approach and the test generation proposed. Also, the feasibility of using the TestDAS tool and the CONTroL based on the results of an observation study is discussed.

5 EVALUATION

As depicted in Chapter 4, TestDAS, proposed in this thesis, addresses the generation of test sequences for testing the DAS adaptive behavior, as well as the DAS model checking. Also, two supporting tools were built to support the TestDAS use. In this chapter, TestDAS evaluations, performed to assess the method and its tools, are described.

This chapter is organized as follows. Section 5.1 presents the results of the evaluation of the model checking approach of TestDAS. Section 5.2 describes the experiment performed to assess the tests generated by TestDAS. Section 5.3 discusses the feasibility of using the TestDAS tool and CONTroL for applying the TestDAS method. Finally, Section 5.4 concludes this chapter.

5.1 Assessment of the Faults Identification Effectiveness

The TestDAS method has a model checking approach (see Section 4.3) and a set of behavioral properties (see Section 4.3.2) whose violation reveals faults in the DAS design. Thus, since the goal of this approach is to support the identification of design faults, this evaluation aims to investigate the following question:

(**Q1**) *How effective is the model checking approach in detecting behavioral faults?*

In order to answer this question, it was used the mutation analysis (OFFUTT; UNTCH, 2001), in which faulty versions (*mutants*) of a correct DAS design were created with the purpose to assess if the model checking approach can support the identification of violations of the properties specified in Section 4.3.2. The mutation analysis technique was chosen for this evaluation, because it allows the systematic seeding of faults and the statistical analysis of fault detection effectiveness of the model checking approach.

With regards to the study object, it was used the Mobile Guide DSPL (MARINHO *et al.*, 2013) that is a DAS related to the mobile and context-aware applications domain. In particular, for the experiment, it was used a part of this DAS, which is presented in Section 2.1.

This evaluation was carried out based on the following phases:

- **Phase I: Mutants Generation**. The first phase of this evaluation was the generation of *mutants* (OFFUTT; UNTCH, 2001) from the original DAS specification. Thus, from a correct design of the Mobile Guide (*i.e.*, a specification that does not violate any property) new feature models were created with different syntactic changes in the adaptation rules

design. These changes were based on a set of transformation rules called *mutation operators* (JIA; HARMAN, 2011).

- **Phase II: Mutants Model Checking**. In this phase, it was performed the model checking approach in each mutant generated to verify if they satisfy the behavioral properties defined in Section 4.3.2. The mutants that violated any property were marked as *killed*.
- **Phase III: Equivalent Mutants Identification**. The last phase is related to the manual inspection of the not killed mutants to determine whether they are equivalent to the Mobile Guide. In this evaluation, the equivalent mutants are those that produce the same adaptive behavior of the Mobile Guide. Thus, these mutants are not expected to be identified by the model checking approach.

Furthermore, aiming to provide evidence regarding the research question **Q1**, the mutant score (JIA; HARMAN, 2011) of the model checking approach was measured based on Formula 5.1.

$$MutationScore(MS) = \frac{\#KilledMutants}{(\#GeneratedMutants - \#EquivalentMutants)} \tag{5.1}$$

where:

$\#KilledMutants$ is the number of killed mutants during Phase II

$\#GeneratedMutants$ is the number of mutants generated in Phase I

$\#EquivalentMutants$ is the number of equivalent mutants identified in Phase III

Subsection 5.1.1 describes the process used to generate mutants. Next, the results of the mutants model checking are described in Subsection 5.1.2. Then, the equivalent mutants and the mutation score are described in Subsection 5.1.3. Finally, a discussion of the results and the threats to validity are addressed in Subsection 5.1.4 and Subsection 5.1.5, respectively.

5.1.1 Mutants Generation

The mutants are created by syntactic changes to the original model or program, which are derived from predefined sets of rules called *mutation operators* (OFFUTT; UNTCH, 2001). For instance, typical mutation operators include the modification of Boolean expressions by replacing the operators.

In the literature, it was not found mutation operators focused on context-aware feature models or adaptation rules. Thus, for this evaluation, mutation operators were defined based on the set of mutants operators proposed by Arcaini *et al.* (2016) for SPL feature models. Arcaini *et al.* (2016) present 14 operators and classify them into two types: feature-based and constraint-based. The first one is related to the type of features (mandatory, optional, alternative, or-group), whereas the latter concerns the require and exclude relationships.

It is worth noting that the focus of this evaluation was the mutation over the adaptation behavior. So, the mutation operators defined affect the adaptation rules (see Definition 4.2.1), represented in this work by a tuple with the following elements: (i) a context guard condition; (ii) an action from the set [activate, deactivate]; and (iii) a set of system features. Therefore, these operators were classified into three classes: (i) *context-based*, which are related to the context guard condition of the adaptation rule; (ii) *action-based* that affect the action triggered by the adaptation rule; and (iii) *system-based*, which refers to the features (de)activated by the adaptation rules.

Table 8 presents the feature-based mutation operators of Arcaini *et al.* (2016). Only mutation operators of the type *system-based* were defined from them, since these operators of Arcaini *et al.* (2016) are related to the type of feature. The defined operators for this evaluation are also depicted in Table 8.

Only from *OptToMan* was not possible to derive an operator, since the actions of the adaptation rules do not affect the mandatory features. In total, nine *system-based* mutation operators were defined from the feature-based mutation operators of Arcaini *et al.* (2016).

Table 9 presents the constraint-based mutation operators of Arcaini *et al.* (2016). These operators affect the constraints (require and exclude) among features. As presented in Section 2.2.1, the adaptation rules can be represented by include and exclude relationships and, thus, from the set of constraint-based mutation operators were defined four *action-based* mutation operators that affect the actions of adaptation rules.

With regards to the *context-based* mutation operators, they were created based on the adaptation rule (see Definition 4.2.1) and Context-Kripke Structure (see Definition 2.2.1) definitions. These operators are presented in Table 10. The first operator ($CtxINV_{AR}$) entails a change of the context guard condition of a given adaptation rule to a context that does not exist. The next one ($CtxUNR_{AR}$) refers to a context that exists, but it is not achieved by the Context-Kripke Structure of the DAS. The last operator ($CtxINT_{AR}$) changes the context guard

Table 8 – System-based mutation operators defined to the adaptation rules

Feature-based Mutation Operator (ARCAINI *et al.*, 2016)		System-based Mutation Operator	
ID	Description	ID	Description
AlToOr	an Alternative is changed to an Or	$AltToOR_{AR}$	the activation of only one feature from an alternative group is changed to activate more than one feature of this group
AlToAnd	an Alternative is changed to an And	$AltToAnd_{AR}$	the activation of only one feature from an alternative group is changed to activate all features of this group
OrToAl	an Or is changed to an Alternative	$OrToAl_{AR}$	the activation of more than one feature from an OR-group is changed to activate only one feature of this group
OrToAnd	an Or is changed to an And	$OrToAnd_{AR}$	the activation of more than one feature from an OR-group is changed to activate all features of this group
AndToOr	an And is changed to an Or	$AndToOr_{AR}$	the activation of all features from an And-group is changed to activate one or more features (not all) of this group
AndToAl	an And is changed to an Alternative	$AndToAl_{AR}$	the activation of all features from an And-group is changed to activate only one feature of this group
ManToOpt	a mandatory relation is changed to optional	-	Not applicable
OptToMan	an optional relation is changed to mandatory	$OptToMan_{AR}$	the target of an action (act/deac) of the rule is changed from an optional feature to a mandatory feature
MF	a feature `f` is removed	$DelF_{AR}$	a feature `f` is removed from an adaptation rule
MoveF	a feature `f` is moved as child of another feature (not belonging to its descendants)	$MoveF_{AR}$	a feature `f` is moved from an adaptation rule to another one

Source – The author

Note – According to Arcaini *et al.* (2016), the parent-child relation is one of the following types: (1) Or - at least one of the sub-features must be selected if the parent is selected; (2) Alternative (xor) – exactly one of the sub-features must be selected whenever the parent feature is selected; (3) And - each child of an And must be either Mandatory (it is always selected) or Optional (it may or may not be selected).

conditions of adaptation rules to make two rules, which have different actions on the same feature, be triggered with the same context.

Besides the first-order mutation operators described in Tables 8, 9 e 10, a set of high-

Table 9 – Action-based mutation operators defined to the adaptation rules

Constraint-based Mutation Operator (ARCAINI *et al.*, 2016)		Action-based Mutation Operator	
ID	Description	ID	Description
MC	an extra-constraint is removed	$DelRl_{AR}$	an adaptation rule is removed
ReqToExcl	a requires constraint is transformed into an excludes constraint	$ActToDea_{AR}$	an activation action is transformed into a deactivation action
ExclToReq	an excludes constraint is transformed into a requires constraint	$DeaToAct_{AR}$	a deactivation action is transformed into an activation action
GC	a general constraint is modified by inserting a new feature, changing a logical operator, or removing part of it	$AddRl_{AR}$	an activation/deactivation of a feature `f` is added

Source – The author

Note – According to Arcaini *et al.* (2016): (1) *extra-constraints* are cross-tree relations (require/exclude) among features; (2) *general constraint* is specified through a propositional formula (using the usual Boolean operators) representing the features as propositional variables

Table 10 – Context-based mutation operators defined to the adaptation rules

ID	Description
$CtxINV_{AR}$	change a context guard condition to one that does not exists
$CtxUNR_{AR}$	change a context guard condition to one unreachable
$CtxINT_{AR}$	change the context guard conditions of adaptation rules that have different actions (activation/deactivation) on the same feature `f` to make them be triggered at the same time

Source – The author

order mutation operators was created by performing more than one mutation. These operators are presented in Table 11 and they focus on the context and actions of the adaptation rules.

Table 11 – Higher-order mutation operators defined to the adaptation rules

ID	Type	Description
$DelRlAll_{AR}$	action-based	remove all adaptation rules that affect a feature `f`
$ActToDeaAll_{AR}$	action-based	change all activation actions that affect a feature `f` to a deactivation action
$DeaToActALl_{AR}$	action-based	change all deactivation actions that affect a feature `f` to an activation action
$CtxINVAll_{AR}$	context-based	change the context guard condition of all rules that affect a feature `f` to contexts that does not exists
$CtxUNRAll_{AR}$	context-based	change the context guard condition of all rules that affect a feature `f` to contexts unreachable
$CtxINTAll_{AR}$	context-based	change the context guard condition of adaptation rules to ensure that for each action that affects a feature `f` there is an opposite action triggered by the same context

Source – The author

Therefore, 22 mutation operators were defined for this evaluation: (i) nine system-based mutation operators; (ii) seven action-based mutation operators; and (iii) six context-based mutation operators.

For each mutation operator, one or more mutants were generated manually from the Promela code with the design of the Mobile Guide. This code refers to the Dynamic Feature Transition System of this DAS, as discussed in Section 4.2.

Figure 31 presents an example of mutant created from the operator $OptToMan_{AR}$, which changes the target of an action from a context-aware feature to a mandatory feature. In particular, the mutation operator changes the deactivation of the feature Image (`buss!imageOff`) to the deactivation of the mandatory feature Show Documents (`buss!showDocsOff`).

Figure 31 – Example of mutant created from the Mobile Guide

```
proctype AR01() {
    if
        :: (battery == 1 && hasPwSrc == false) -> buss!imageOff;buss!videoOff;buss!done
        :: else -> buss!done
    fi
}
```

original

```
proctype AR01() {
    if
        :: (battery == 1 && hasPwSrc == false) -> buss!showDocsOff;buss!videoOff;buss!done
        :: else -> buss!done
    fi
}
```

mutant

Source – the author

It is worth pointing out that in the original feature model of the Mobile Guide, both features Image and Video are optional. Thus, to apply the operators $AltToOR_{AR}$ and $AltToAnd_{AR}$, it was created a slightly modified version of the original feature model, in which the features Image and Video have an alternative relationship. Also, another slightly modified version of the original feature model was needed to apply the operators $OrToAl_{AR}$ and $OrToAnd_{AR}$. In the latter, the features Image and Video have an or-relationship.

Table 12 presents the number of mutants generated per type of mutation operator and per each operator. In total, 114 mutants were generated. The system-based mutation operators created the most of these mutants (54%). A short description of each mutant is depicted in Appendix D, whereas the *.pml* files correspondent to the mutants and the original models are available at *https://github.com/GREatResearches/TestDAS*.

Table 12 – Number of mutants generated

Group	# Mutants per Group	Operator	# Mutants
System-based	62	$AltToOR_{AR}$	4
		$AltToAnd_{AR}$	4
		$OrToAl_{AR}$	3
		$OrToAnd_{AR}$	3
		$AndToOr_{AR}$	2
		$AndToAl_{AR}$	6
		$OptToMan_{AR}$	20
		$DelF_{AR}$	10
		$MoveF_{AR}$	10
Action-based	30	$DelRl_{AR}$	5
		$ActToDea_{AR}$	6
		$DeaToAct_{AR}$	4
		$AddRl_{AR}$	10
		$DelRlAll_{AR}$	1
		$ActToDeaAll_{AR}$	2
		$DeaToActALl_{AR}$	2
Context-based	22	$CtxINV_{AR}$	5
		$CtxUNR_{AR}$	5
		$CtxINT_{AR}$	8
		$CtxINVAll_{AR}$	1
		$CtxUNRAll_{AR}$	1
		$CtxINTAll_{AR}$	2
TOTAL			114

Source – The author

5.1.2 Mutants Model Checking

During the Mutants Model Checking phase, the model checking approach of the TestDAS (see Section 4.3) was applied in each mutant generated. Then, the SPIN model checker was used through the TestDAS tool (see Section 4.5) to verify the properties defined in Section 4.3.2 over the Promela code corresponding to each mutant.

Table 13 depicts the results of this phase of the mutant analysis. In this table, the number of *killed* mutants means the number of mutants that have a violation in some of the properties checked. In total, for each mutant were checked 15 properties: (i) one from the behavioral property *Configuration Correctness* that checks whether the product states are valid regarding the feature model; (ii) five from the property *Rule Liveness* that check if each one of the five adaptation rules of the Mobile Guide is triggered at some time; (iii) five from the property *Interleaving Correctness* to check if the actions of all adaptation rules of the Mobile Guide are performed; (iv) two from the property *Feature Liveness* to check if the features Image and Video are activated at some time; and (v) two from the property *Variation Liveness* to check if the features Image and Video are deactivated at some time.

Table 13 – Results of the mutants model checking

Group	Operator	# Mutants	# Killed per Property					# Killed Mut.	% Killed Mut.
			P1	P2	P3	P4	P5		
System-based	$AltToOR_{AR}$	4	4	0	0	0	0	4	100%
	$AltToAnd_{AR}$	4	4	0	0	0	0	4	100%
	$OrToAl_{AR}$	3	1	0	0	0	1	2	67%
	$OrToAnd_{AR}$	3	0	0	0	0	1	1	33%
	$AndToOr_{AR}$	2	0	0	0	0	1	1	50%
	$AndToAl_{AR}$	6	0	0	0	0	1	1	17%
	$OptToMan_{AR}$	20	8	0	0	0	0	8	40%
	$DelF_{AR}$	10	0	0	0	0	1	1	10%
	$MoveF_{AR}$	10	0	0	10	0	0	10	100%
Action-based	$DelRl_{AR}$	5	0	0	0	0	1	1	20%
	$ActToDea_{AR}$	6	0	0	0	0	0	0	0%
	$DeaToAct_{AR}$	4	0	0	0	0	1	1	25%
	$AddRl_{AR}$	10	0	0	10	0	0	10	100%
	$DelRlAll_{AR}$	1	0	0	0	1	0	1	100%
	$ActToDeaAll_{AR}$	2	0	0	0	2	0	2	100%
	$DeaToActALl_{AR}$	2	0	0	0	0	2	2	100%
Context-based	$CtxINV_{AR}$	5	0	5	0	0	0	5	100%
	$CtxUNR_{AR}$	5	0	5	0	0	0	5	100%
	$CtxINT_{AR}$	8	0	0	8	0	0	8	100%
	$CtxINVAll_{AR}$	1	0	1	0	(1)	0	1	100%
	$CtxUNRAll_{AR}$	1	0	1	0	(1)	0	1	100%
	$CtxINTAll_{AR}$	2	0	0	2	0	0	2	100%
Total		114	17	12	30	3	9	71	62%

Source – The author

Note – P1 = Configuration Correctness; P2 = Rule Liveness; P3 = Interleaving Correctness; P4 = Feature Liveness; P5 = Variation Liveness.

Note – "(x)" indicates that the property killed *x* mutants already killed by another property

As presented in Table 13, 71 (62%) from the 114 mutants were *killed* by the model checking approach. With regards to the classes of mutation operators, the number of *killed* mutants was: (i) *System-based*: 32 (52%) from 62 mutants generated by this class of mutation operator; (ii) *Action-based*: 17 (57%) from the 30 mutants generated by this class of operator; and (iii) *Context-based*: all (100%) the 22 generated mutants by this class of operator.

Most of the mutants was *killed* by the violation of only one property. Only the mutants from the mutation operators $CtxINVAll_{AR}$ and $CtxUNRAll_{AR}$ were killed by two properties (*Rule Liveness* and *Feature Liveness*).

It is worth noting that the identification of the mutants by the properties depends on both the mutation operator and the DAS model. As depicted in Table 13, in the Mobiline, for some operators all the mutants were killed. For instance, the operator $CtxINV_{AR}$ created five mutants by changing the adaptation rules trigger to an invalid context. Thus, the affected adaptation rules were not triggered and, therefore, such mutants were *killed* by property *P2* -

Rule Liveness, which states that all adaptation rules should be triggered at some state. On the other hand, none of the mutants from the operator $ActToDea_{AR}$ was killed. An analysis of the mutants not killed was made in the Equivalent Mutants Identification phase, which is presented in the next subsection.

5.1.3 Equivalent Mutants and Mutation Score

The equivalent mutants are those that behave like the original Mobile Guide, that is, those that generate the same product configurations of the Mobile Guide in the same context situations. Since they have the same adaptive behavior, the model checking approach cannot distinguish such mutants from the original Mobile Guide.

Table 14 presents the list of alive mutants after the model checking and the result of the analysis of equivalent mutants. This analysis was made based on a comparison of the product adaptations triggered by the original DAS and the mutant.

For instance, the alive mutant AM11 has a mutation in the adaptation rule AR03 in which the action to activate the feature Image (`Image(ON)`) was changed to affect the feature ShowDocs (`ShowDocs(ON)`). Since the feature ShowDocs is mandatory, it is active all time in the original DAS and the before mentioned mutation does not violate this condition. Besides that, given the context variation model used (see Section 2.2.1), the adaptation rule AR05, which activates both features *Image* and *Video*, is triggered before the AR03. Then, the execution of the rule AR03 of the mutant AM11 only deactivated *Video*, achieving the same product configuration (with the feature *Image* activated and the feature *Video* deactivated) as the original Mobile Guide.

As presented in Table 14, from the analysis of the alive mutants were identified 15 equivalent mutants. Thus, by using the formula 5.1, the mutation score achieved is equals to 72% (71/(114-15)).

5.1.4 Discussion

The evaluation of the effectiveness of the model checking approach was focused on the investigation of the following question: *(Q1) How effective is the model checking approach in detecting behavioral faults?* For this purpose, the mutant analysis was used in order to create mutants versions from a correct DAS specification to verify whether the TestDAS supports the identification of these mutants.

Table 14 – List of Alive Mutants (AM)

Operator	ID	AR	Original	Mutation	Equivalent?
$OrToAl_{AR}$	AM1	AR05	Image(ON), Video(ON)	Image (ON), Video (OFF)	No
$OrToAnd_{AR}$	AM2	AR02	Image(ON), Video(OFF)	Image(ON), Video(ON)	No
	AM3	AR03	Image(ON), Video(OFF)	Image(ON), Video(ON)	No
$AndToOr_{AR}$	AM4	AR01	Image(OFF), Video(OFF)	Image(OFF), Video(ON)	No
$AndToAl_{AR}$	AM5	AR01	Image(OFF), Video(OFF)	Image(OFF), Video(ON)	No
	AM6	AR04	Image(ON), Video(ON)	Image(OFF), Video(ON)	No
	AM7	AR05	Image(ON), Video(ON)	Image(OFF), Video(ON)	No
	AM8	AR04	Image(ON), Video(ON)	Image (ON), Video (OFF)	No
	AM9	AR05	Image(ON), Video(ON)	Image (ON), Video (OFF)	No
$OptToMan_{AR}$	AM10	AR02	Image(ON), Video(OFF)	ShowDocs(ON), Video(OFF)	No
	AM11	AR03	Image(ON), Video(OFF)	ShowDocs(ON), Video(OFF)	Yes
	AM12	AR04	Image(ON), Video(ON)	ShowDocs(ON), Video(ON)	Yes
	AM13	AR04	Image(ON), Video(ON)	Image(ON), ShowDocs(ON)	No
	AM14	AR05	Image(ON), Video(ON)	ShowDocs(ON), Video(ON)	Yes
	AM15	AR05	Image(ON), Video(ON)	Image(ON), ShowDocs(ON)	Yes
	AM16	AR02	Image(ON), Video(OFF)	Text(ON), Video(OFF)	No
	AM17	AR03	Image(ON), Video(OFF)	Text(ON), Video(OFF)	Yes
	AM18	AR04	Image(ON), Video(ON)	Text(ON), Video(ON)	Yes
	AM19	AR04	Image(ON), Video(ON)	Image(ON), Text(ON)	No
	AM20	AR05	Image(ON), Video(ON)	Text(ON), Video(ON)	Yes
	AM21	AR05	Image(ON), Video(ON)	Image(ON), Text(ON)	Yes
$DelF_{AR}$	AM22	AR01	Image(OFF), Video(OFF)	Image (OFF)	Yes
	AM23	AR02	Image(ON), Video(OFF)	Image(ON)	Yes
	AM24	AR02	Image(ON), Video(OFF)	Video(OFF)	No
	AM25	AR03	Image(ON), Video(OFF)	Image(ON)	No
	AM26	AR03	Image(ON), Video(OFF)	Video(OFF)	Yes
	AM27	AR04	Image(ON), Video(ON)	Image(ON)	No
	AM28	AR04	Image(ON), Video(ON)	Video(ON)	Yes
	AM29	AR05	Image(ON), Video(ON)	Image(ON)	Yes
	AM30	AR05	Image(ON), Video(ON)	Video(ON)	Yes
$DelRl_{AR}$	AM31	AR02	Image(ON), Video(OFF)	delete AR	No
	AM32	AR03	Image(ON), Video(OFF)	delete AR	No
	AM33	AR04	Image(ON), Video(ON)	delete AR	No
	AM34	AR05	Image(ON), Video(ON)	delete AR	Yes
$ActToDea_{AR}$	AM35	AR02	Image(ON), Video(OFF)	Image (OFF), Video (OFF)	No
	AM36	AR03	Image(ON), Video(OFF)	Image(OFF), Video(OFF)	No
	AM37	AR04	Image(ON), Video(ON)	Image(OFF), Video(ON)	No
	AM38	AR04	Image(ON), Video(ON)	Image(ON), Video(OFF)	No
	AM39	AR05	Image(ON), Video(ON)	Image(OFF), Video(ON)	No
	AM40	AR05	Image(ON), Video(ON)	Image(ON), Video(OFF)	No
$DeaToAct_{AR}$	AM41	AR01	Image(OFF), Video(OFF)	Image(OFF), Video(ON)	No
	AM42	AR02	Image(ON), Video(OFF)	Image(ON), Video(ON)	No
	AM43	AR03	Image(ON), Video(OFF)	Image(ON), Video(ON)	No

Source – The author

Note – The mutant equivalent identification took into account the adaptation rules and the context variation model

The results of the mutant analysis showed a mutation score of 72%, indicating that the model checking approach proposed can identify design faults in the adaptive behavior of a DAS. Furthermore, it is important to note that the alive mutants did not violate any property checked. After a manual verification of these mutants, it was possible to observe that they

represent a different adaptive behavior compared to the Mobile Guide, but without design faults. So, by taking into account only the models with design faults, the model checking approach successfully identified all mutants. Therefore, in the evaluation performed, the model checking approach was effective (Q1) in the identification of design faults related to the set of properties presented in Section 4.3.2.

With regards to the behavioral properties checked, each one identified a set of mutants. The property *Interleaving Correctness* has the higher number of *killed* mutants (30), whereas the property *Feature Liveness* identified only three mutants. The following paragraphs discuss the faults identified by each property based on the *killed* mutants.

The violation of the property *Configuration Correctness* occurs when the adaptation rules generate an invalid product configuration. The faults that made this violation in this evaluation were: (F1) an adaptation rule deactivating a mandatory feature; (F2) an adaptation rule activating an optional child feature, but deactivating its parent feature; and (F3) an adaptation rule activating more than one child feature from a xor-group. For instance, a given mutant (created by the operator $OptToMan_{AR}$) deactivated the feature *Text*, which is mandatory (F1). Another one (created by the operator $OptToMan_{AR}$) deactivated the feature *ShowDocs* that is the parent of both features *Image* and *Video* (F1 and F2). The mutants from the operators $AltToOR_{AR}$ and *AltToAndAR*, in turn, activated two features in a xor-group (F3).

The property *Rule Liveness* is violated when the context condition of the adaptation rule is unsatisfiable. The faults that made this violation were: (F4) the context guard of the adaptation rule does not exist in the context variation model; and (F5) the context guard of the adaptation rule is not reachable by the context variation model. As expected, the fault F4 was present in the mutants created by the operators $CtxINV_{AR}$ and $CtxINVAll_{AR}$, whereas the mutants from the operators $CtxUNR_{AR}$ and $CtxUNRAll_{AR}$ had the fault F5.

The violation of the property *Interleaving Correctness* occurs when adaptation rules triggered at the same time have not the expected effect. The faults that made this violation were: (F6) two adaptation rules with the same context guard, but one activating a feature F and other deactivating the same feature; and (F7) two adaptation rules with different context conditions, but that are satisfiable at the same time in some context state, and where one activated and the other deactivated the same feature. The first type of fault (F6) happened with the mutants generated from the operators $CtxINT_{AR}$ and $CtxINTAll_{AR}$. The fault F7 was identified in the mutants generated from $MoveF_{AR}$ and $AddRl_{AR}$.

The violation of the property *Feature Liveness* occurs when the context-aware features are never activated. The faults that made this violation were: (F8) missing the unique adaptation rule that activates a context-aware feature F; and (F9) the rules that should activate a context-aware feature F were deactivating it. The fault F8 happened with the mutants generated from the operators $DelRlAll_{AR}$, $CtxINVAll_{AR}$ and $CtxUNRAll_{AR}$. The mutants from the $ActToDeaAll_{AR}$ had the fault F9.

The violation of the property *Variation Liveness* occurs when context-aware features are never deactivated. The faults that made this violation were: (F10) missing the unique adaptation rule that deactivated a context-aware feature F; and (F11) the rules that should deactivate a context-aware feature F were activating it. The faults F10 and F11 were present in the mutants generated from $DelTl_{AR}$ and $DeaToAct_{AR}$, respectively.

5.1.5 Threats to Validity

The main threats to the validity of the evaluation performed are related to the internal validity and external validity. Threats to internal validity concern issues that may indicate a casual relationship when there is none, whereas threats to external validity are conditions that limit the ability to generalize the results (WOHLIN *et al.*, 2014).

With regards to the internal validity, the main threat is related to the selection of the mutation operators used in the evaluation. Since mutation operators over the adaptation rules were not found, it was defined a set of operators based on the mutation operators proposed by Arcaini *et al.* (2016). The latter concern mutations on a SPL feature model, whose concepts (feature, require and exclude dependence, feature relationships) are present in the context-aware feature models of a DAS. Also, the manual creation of the mutants is a threat to the internal validity, because the wrong mutation generation could affect the results. In order to mitigate this threat, all mutants created were double inspected by the author of this thesis to check the application of the mutation operators and to confirm that all violations were correctly identified by the TestDAS.

The main threat to the external validity is the study object (*i.e.*, the Mobile Guide), which is an academic Dynamic Software Product Line. However, the set of design faults identified during the evaluation and presented in Subsection 5.1.4 is related to errors in the adaptation rules. Thus, they are not domain-specific faults and can be identified in others DAS by the model checking approach proposed.

5.2 Assessment of the Generated Tests

This section presents the empirical evaluation performed through a controlled experiment to assess the tests generated by the TestDAS method proposed in Chapter 4. An experiment is a formal, rigorous and controlled investigation that allows to draw conclusions about a hypothesis that states the relationship between the cause and the effect (WOHLIN *et al.*, 2000).

According to Wohlin *et al.* (2000), the experiment process can be divided into the following activities: (i) *Definition*, in which the experiment is defined in terms of problem, objective and goals; (ii) *Planning*, in which the design of the experiment is defined; (iii) *Operation*, which is the experiment execution for collecting the measures to evaluate the hypothesis; (iv) *Analysis and Interpretation*, which is related to the data analysis, hypothesis test and discussion of the results; and (v) *Presentation and Package* that is concerned with presenting and packaging of the experiment findings. This last activity is important to allow the experiment replication. For this purpose, all experiment data are available online [1].

The next subsections present the experiment performed following the activities suggested by Wohlin *et al.* (2000). Subsection 5.2.1 presents the definition of the experiment. Subsection 5.2.2 describes the experiment design. Subsection 5.2.3 presents the experiment operation, including the lessons learned from a pilot study performed to assess the experiment planning. Subsection 5.2.4 presents the data analysis and interpretation of results. Finally, Subsection 5.2.5 discusses the findings and Subsection 5.2.6 presents the threats to validate of the experiment.

5.2.1 Experiment Definition

By using the template of Wohlin *et al.* (2000), which follows the Goal-Question-Metric (BASILI; ROMBACH, 1994) template for goal definition, the goal of this experiment was defined as follows: to *analyze* the adaptation test sequences generated by the TestDAS *for the purpose of* characterize *with regard to* the number of tests, time spent and tests coverage *from the point of view of* researchers and testers *in the context of* post-graduated (M.Sc. and Ph.D.) students and practitioners testing two DAS in different application domains.

The application domains chosen to the experiment were smart home and mobile visit

[1] https://github.com/GREatResearches/TestDAS

guide. In the first case, it was used the SmartHome DSPL presented by Carvalho *et al.* (2016) that is a DAS for monitoring and controlling the functions of a smart house. In the second one, it was used the MobileGuide DSPL (MARINHO *et al.*, 2013), presented in Section 2.1.

The evaluation question **Q1** was defined to the Mutant Analysis (Section 5.1). Therefore, for the controlled experiment, it were defined the questions **Q2**, **Q3** and **Q4** as follow:

> ***(Q2)*** *Does the number of the tests is increased when the TestDAS is followed?* Rationale: This question intends to evaluate if the proposed testing method creates more tests than the tester's experience based testing;
> ***(Q3)*** *Does the coverage of the tests is increased when the TestDAS is followed?* Rationale: This question intends to evaluate if the proposed testing method creates tests with a better coverage than the tester's experience based testing. This coverage is measured in terms of adaptation actions and contexts; and
> ***(Q4)*** *Does the time spent to create adaptation test cases is lower when the TestDAS is followed?* Rationale: This question is related to the time spent by TestDAS in the generation of test sequences based on the input provided by the experiment's participants. The goal of this question was to assess if the participants spent more time using the TestDAS or specifying the tests manually.

Then, in order to answer the defined questions, three metrics were defined as follows:

- *Number of Adaptation Test Cases (NATC)*. It refers to the number of different test cases that compose the adaptation test sequences created during the experiment.
- *Adaptation Test Coverage (ATC)*. In this thesis, the adaptation of a DAS refers to the activation/deactivation of its features based on the context. Thus, this measure refers to the number of adaptation actions (*#ActionsCovered*) - feature action/deactivation - and context situations (*#ContextCovered*) covered by the test cases. The measure ACT of a test sequence TS is given by the formula 5.2. Note that the total number of adaptation actions (*#TotalActions*) is twice the number of context-aware features since each one should be activated and deactivated. The total number of context situations (*#TotalContexts*) refers to the number of context states.

$$ATC(TS) = \frac{\#ActionsCovered + \#ContextCovered}{\#TotalActions + \#TotalContexts} * 100 \tag{5.2}$$

- *Time Spent (TS)*. It gives the time spent in minutes and seconds by the experiment's participant to create the adaptation test cases to the DAS used in the experiment.

5.2.2 Experiment Planning

The goal of the *Planning* phase is to define *how* the experiment is conducted (WOHLIN *et al.*, 2000). Thus, during this phase it was defined the hypothesis being investigated, variables involved, subjects, as well as the experiment design and the materials used. The planning of the experiment conducted to assess the tests generated by the TestDAS is detailed in the following subsections.

5.2.2.1 Hypothesis Investigated

As mentioned before, a controlled experiment is an empirical study that provides evidence to test a hypothesis, which states the relationship between the cause and effect investigated. Based on the questions defined during the *Definition* phase, the hypotheses defined are stated as follows:

Hypothesis related to the **Q2**

- $HN_0 : NTestsWithTestDAS = NTestsWithExperienceBasedTesting$
- $HN_1 : NTestsWithTestDAS \neq NTestsWithExperienceBasedTesting$

Hypothesis related to the **Q3**

- $HC_0 : CoverageWithTestDAS = CoverageWithExperienceBasedTesting$
- $HC_1 : CoverageWithTestDAS \neq CoverageWithExperienceBasedTesting$

Hypothesis related to the **Q4**

- $HT_0 : TimeWithTestDAS = TimeWithExperienceBasedTesting$
- $HT_1 : TimeWithTestDAS \neq TimeWithExperienceBasedTesting$

For the questions **Q2**, **Q3** and **Q4**, the null hypothesis (HN_0, HC_0 and HT_0) states that the use of the TestDAS method provides the same results as the experience-based testing, while the alternative hypothesis states that they have different results regarding the number of test cases (HN_1), test coverage (HC_1) and time spent to generate the tests (HT_1). Therefore, this experiment aims to compare the use of two treatments for DAS testing: T1 - TestDAS and T2 - Experience-based testing.

5.2.2.2 *Experiment Variables*

According to Wohlin *et al.* (2000), a controlled experiment involves two kind of variables: (i) independent variables that are those variables that can be controlled and changed in the experiment; and (ii) dependent variable that are those which are observed to measure the effect of the treatment.

In this experiment, the independent variables are the DAS used (SmartHome and Mobiline) and the background experience of the subjects. The dependent variables are the metrics *Number of Adaptation Test Cases*, *Adaptation Test Coverage* and *Time Spent*, which were defined to answer the questions of the experiment.

5.2.2.3 *Subjects*

People who apply the treatments are called subjects (WOHLIN *et al.*, 2000). In this experiment, the subjects were chosen based on convenience sampling, in which the nearest and most convenient persons are selected as subjects (WOHLIN *et al.*, 2000).

In total, twelve participants were selected for the experiment. Six of them are postgraduate students, that is, two P.hD. students and four M.Sc. students of Computer Science from the Federal University of Ceará. The other six are professionals who work as Tester or Test Analyst at the Group of Computer Networks, Software Engineering and Systems (GREat[2]).

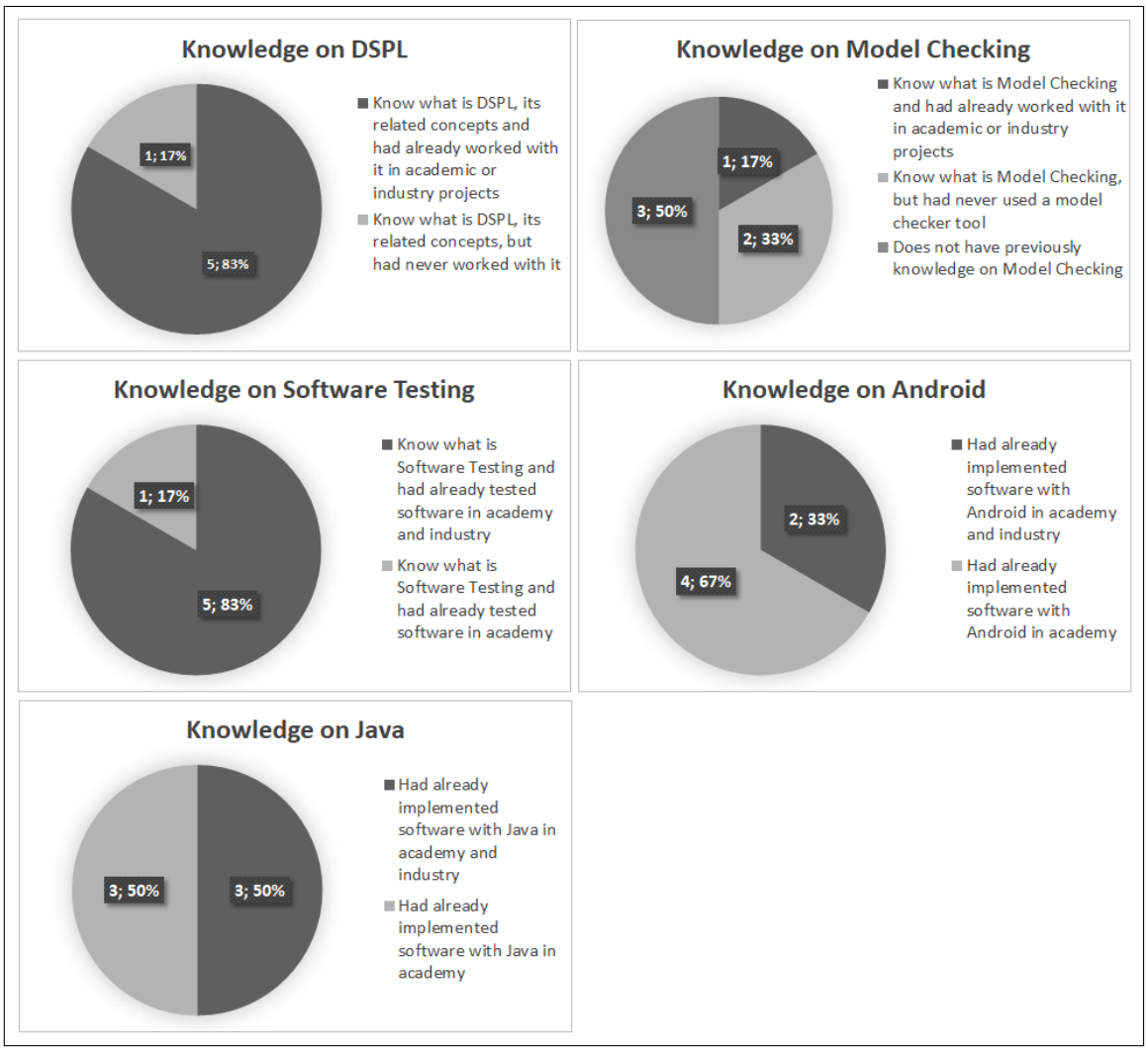
Figure 32 presents the profile of the students that have participated in the experiment. All postgraduate students already knew what is a DSPL and its main concepts. Five of them (83%) had worked with DSPL in academic projects. Besides that, all students knew about software testing and five of them (83%) have already tested software in academy and industry. All students from the experiment also had already implemented software with Java and Android[3]. Regarding the model checking, most of them (67%) knew what is model checking, but only one (17%) had used a model checker tool. Also, two of the students (33%) did not know about model checking.

From the group of professionals involved in the experiment, see Figure 33, only one (17%) knew what is a DSPL and its related concepts, such as feature model and adaptation rules. All of them had already experience on software testing in academy and industry. In this case, two

[2] http://great.ufc.br

[3] This information is important since the CONTroL, used in the observational study depicted in Section 5.3, requires basic knowledge on Java and Android

Figure 32 – Profile of the students from the experiment



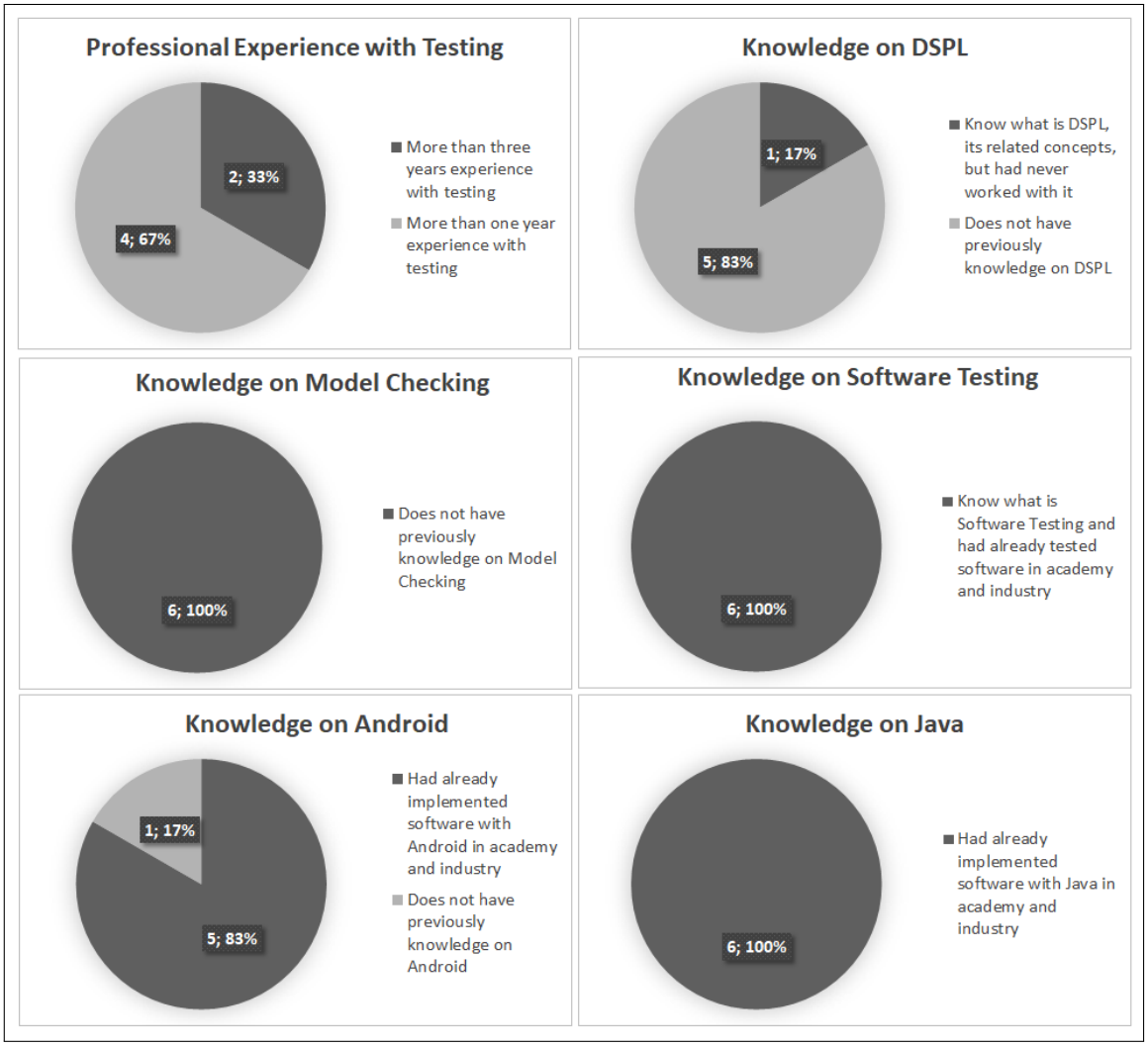
Source – the author

of the professionals (33%) had more than three years' experience, whereas the other five (67%) had at least one year's experience on software testing. Also, all of them had already implemented software with Java and the most of them (83%) had already used Android to implement a mobile application. However, none of them had used a model checker tool.

Figure 34 summarizes the previous knowledge of all subjects of the experiment on the two main concepts used in the experiment: Software Testing and DSPL.

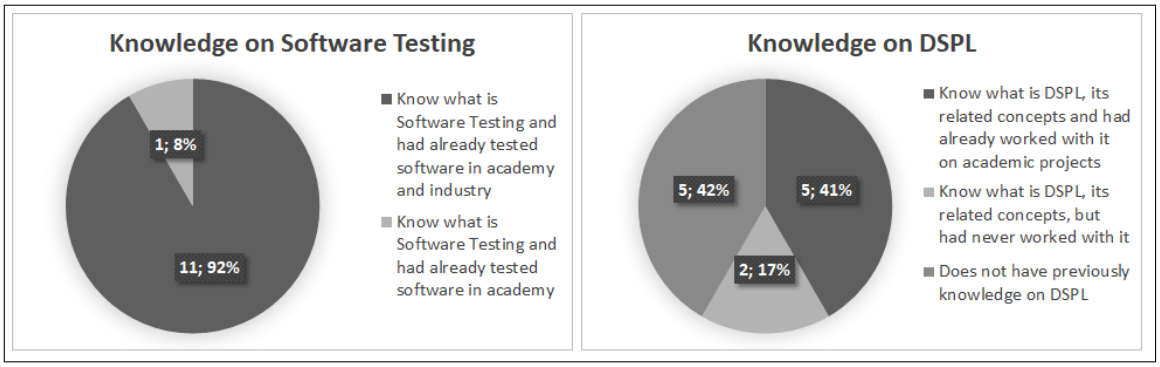
All subjects of the experiment had already experience with software testing, with the majority (92%) having tested software in both academy and industry. With regards to the knowledge on DSPL, seven of the subjects (58%) knew the DSPL concepts and five of them (41%) had already worked with DSPL/context variability in academic projects.

Figure 33 – Profile of the professionals from the experiment



Source – the author

Figure 34 – Knowledge of the subjects on Software Testing and DSPL concepts



Source – the author

5.2.2.4 Experiment Design and Instrumentation

The experiment performed used the design type composed of one factor (the method used for DAS testing) with two treatments: (T1) With the TestDAS method; (T2) Without the TestDAS, that is, based on the experience of the subject.

Table 15 presents the experiment design used, whose goal was to compare the two treatments each other (WOHLIN *et al.*, 2000). To perform the tasks of the experiment, the subjects were divided into two groups, A and B. In Group A, the subjects not used the TestDAS (T2) to perform the Task I, and used the TestDAS (T1) in Task II. In Group B, the order was the opposite, that is, the subjects of this group used the TestDAS (T1) in Task I, but did not use it in Task II (T2). It is worth noting that for the tasks I and II the study objects used were the Mobiline and SmartHome, respectively.

Table 15 – Experiment Design

Task	Group A	Group B
Task I (Mobiline)	Experience-Based (T2)	TestDAS (T1)
Task II (SmartHome)	TestDAS (T1)	Experience-Based (T2)

Source – The author

Therefore, all subjects used the two treatments, but with different objects (DPLS) in each task. The group of the subjects defined the order of the treatments used. The groups organization, in turn, was made randomly, but ensuring a balanced number of professionals and students in each group.

With regards the material used during the experiment, it is presented in Appendix E. First, the subjects filled out the *Background Form* reporting information about their knowledge on DSPL, Software Testing, Model Checking and Java/Android.

Before the execution of the experiment tasks, the subjects received training addressing the concepts related to DSPL, Model Checking, and Software Testing. Furthermore, the subjects performed an exercise to learn the format that should be used to specify manually the adaptation test cases and how to use the TestDAS tool.

For each task, the subjects received a document describing the DSPL, its feature model and a description of the environment context behavior. If the task would be performed with the TestDAS, the subject also received the context variation model of the DSPL under testing. After each task, the subjects should answer the *Post-Task Questionnaire* to provide their

feedback about the task execution.

After performing the Task II, the subjects filled out the *Post-Experiment Questionnaire*, in which they could state their feedback about the entire experiment and their feeling about the best way for testing the DAS adaptive behavior.

5.2.3 Experiment Operation

In the *Operation* phase, the experiment is carried out in order to collect the data that should be analyzed (WOHLIN *et al.*, 2000). This phase comprises three steps: (i) *Preparation*, in which the material is prepared and the subjects are invited; (ii) *Execution*, in which the subjects perform the experiment tasks and the data is collected; and (iii) *Data Validation*, in which the collected data is validated.

During the *Preparation*, the subjects 5.2.2.3 were invited to participate of the experiment. The requirements for the selection of the subjects were: (i) to have previous knowledge on Software Testing; and (ii) to have experience on software testing. Based on these requirements and the convenience sampling (WOHLIN *et al.*, 2000), 12 subjects were selected (see Section 5.2.2.3). Also, before the experiment execution, all material presented in Section 5.2.2.4 was prepared.

In the *Execution*, the experiment was performed by using the design described in Section 5.2.2.4. At this moment, the data about the background and the feedback of the subjects were collected through forms (see Appendix E), and the metrics (see Section 5.2.1) were collected by analyzing the tests generated and monitoring the time spent during the tasks. More details of the *Execution* are presented in Subsection 5.2.3.2.

With regards the *Data Validation*, it was performed after the experiment execution to check that the data were collected correctly. In this experiment, all subjects completed the tasks and filled out the required forms. Therefore, the data from all subjects could be used in the *Analysis* phase.

A pilot study was also performed to assess the experiment planning and operation. The results and lessons learned from the pilot study are described in Subsection 5.2.3.1.

5.2.3.1 Pilot Study

The pilot study (WOHLIN *et al.*, 2014) aims to identify problems in the experiment planning or operation. It is performed before the experiment tasks session, and usually involves

a few subjects.

For the pilot study, it was selected two master students from the Federal University of Ceará. These students knew software testing and had already tested software in academy and industry. One of them knew what is a Dynamic Software Product Line and its main concepts (i.e., feature model, context-based runtime adaptation), whereas the other one did not know what is a DSPL. Also, both have already implemented software with Java and Android, but they had never used the model checking before.

Each student was assigned to a different group, A or B. After that, they followed the steps planned to the experiment: (i) to answer the *Background Form* about previous experience; (ii) to review the concepts during a training session; (iii) to execute the experiment tasks and answer the *Post-Task Questionnaires*; and (iv) to answer the *Post-Experiment questionnaire*. Then, a discussion with these subjects was performed to identify their feedback about the experiment execution and difficulties they have during the activities.

Based on the pilot study, a set of improvements was made in the experiment forms, training and task instrumentation. The changes performed are summarized follows:

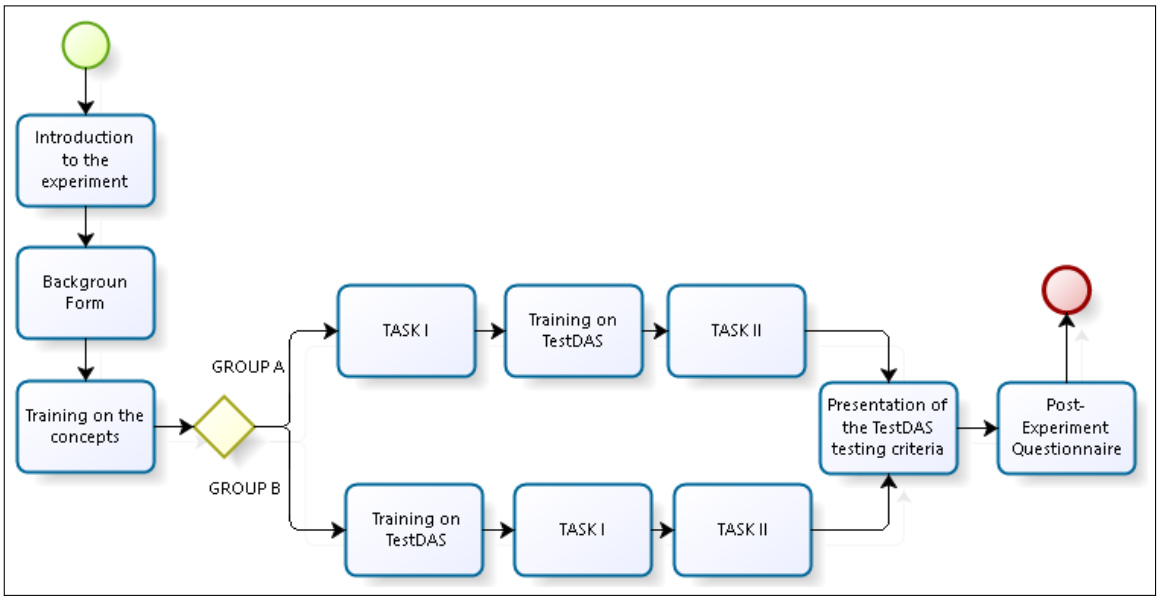
- **Forms**. Some form questions were improved to avoid misunderstanding. Also, the forms were adjusted to follow the same pattern, for example, the use of the five points Likert scale (ROBINSON, 2014) whenever possible;
- **Training**. Improvement of the material by adding examples of contexts, a context variation model and adaptation test cases; and
- **Experiment Task**. The task document was improved to better describe the features and contexts of the DAS feature model used. Besides that, it was developed a template to be used during the manual test cases specification since the subjects had doubts about the specification format. Also, it was created a figure to explain to the subjects, after the execution of Task II, the test criteria used by the TestDAS. The latter was required because in the last experiment activity (i.e., when the subjects are asked to fill out the *Post-Experiment Questionnaire*) the subjects should compare the test criteria used during the experience-based testing and the ones used by the TestDAS.

Therefore, the pilot study was important to identify problems in the experiment planning that could affect its results. All improvements before mentioned are already considered in the materials presented in Appendix E.

5.2.3.2 *Experiment Execution*

Figure 35 presents the experiment activities. Initially, the experiment tasks were introduced to the subjects. After that, they filled out the *Background Form* with information about their previous knowledge. Next, the subjects were randomly divided into two groups, A and B, keeping a balanced number of professionals and researchers in each group.

Figure 35 – Activities performed in the experiment



Source – the author

All subjects received training on the concepts related to the experiment aiming to balance the knowledge on DSPLs and testing of the adaptive behavior. This training session spent about 10 minutes, and involved a slide presentation and exercises to the better understating of the concepts used in the experiment.

The participants from the Group A specified tests in Task I based on their expertise. After that, they received training on TestDAS tool for performing the Task II. This training spent about 15 minutes, and involved a slide presentation and exercises with the TestDAS tool. On the other hand, the participants from the Group B used the TestDAS in Task I and, thus, before this task, they received the training on TestDAS tool. Next, these subjects specified tests in Task II based on their experience. Therefore, all the subjects used both treatments, allowing one group to act as the control of the other.

In each task, after the creation of the test cases (based on the experience or using the TestDAS), the subjects filled out the *Post-Task Questionnaire*. After the execution of Task II,

the test coverage criteria used by the TestDAS were presented to the subjects. It is worth noting that these criteria were presented only after the Task II in order to not influence the execution of the tasks. In particular, the Task II of the Group B that should be conducted based on the participant's experience.

Furthermore, all subjects were asked to fill out the *Post-Experiment Questionnaire*. More details about the forms used in the experiment are depicted in Section 5.2.2.4.

5.2.4 Data Analysis and Interpretation of Results

The analysis was performed based on descriptive statistics, hypothesis testing and the feedback of the subjects reported in the forms. Thus, the next subsection presents the data from the experiment tasks. SubSection 5.2.4.2 describes the qualitative data gathered from the forms. After that, Subsection 5.2.4.3 describes the hypothesis testing through statistical analysis, and Subsection 5.2.5 draws conclusions based on the presented data.

5.2.4.1 Descriptive Statistics

Table 16 presents the raw experimental data. In Task I, the mean value in the Group A for the time spent was 4m 6s, with a standard deviation (SD) of 2m 34s, whereas in the Group B the mean was 3m 52s, with SD of 1m 2s. Median values were 3m 13s and 3m 50s for the Group A and Group B, respectively.

Table 16 – Raw experimental data

Subject	Profile	Group	Task I			Task II		
			Time(m:s)	# Tests	Coverage	Time(m:s)	# Tests	Coverage
S1	Student	A	3:00	4	58.33%	8:30	6	100%
S2	Student	A	1:51	4	58.33%	4:26	6	100%
S3	Student	A	3:26	4	58.33%	5:52	6	100%
S4	Professional	A	2:23	4	58.33%	5:33	6	100%
S5	Professional	A	5:09	4	58.33%	6:48	6	100%
S6	Professional	A	8:49	6	100%	6:47	6	100%
S7	Student	B	5:10	6	100%	7:26	6	100%
S8	Student	B	4:50	6	100%	11:41	4	62.50%
S9	Student	B	3:02	6	100%	5:19	4	56.25%
S10	Professional	B	3:33	6	100%	5:16	4	56.25%
S11	Professional	B	2:30	6	100%	3:58	4	62.50%
S12	Professional	B	4:08	6	100%	3:29	4	62.50%

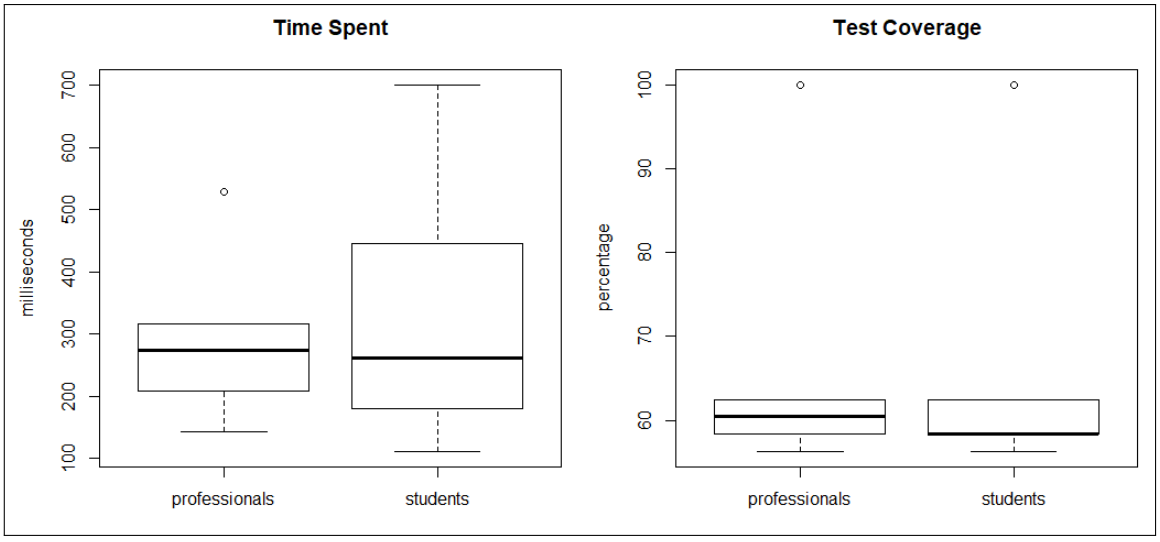
Source – The author

With regards to the test coverage, the mean value for Group A was 65.28%, with SD of 17.01% and a median value of 58.33%. On the other hand, all subjects of the Group B achieved 100%.

In Task II, the mean value in Group A for the time spent was 6m 19s, with SD of 1m 22s and the median value of 6m 19s. The Group B, in turn, has as mean 6m 11s, with SD of 3m 1s and the median value of 6m 11s. Regarding the test coverage, all subjects of Group A achieved 100%, whereas the mean value for Group B was 66.67%, with SD of 16.61% and the median value of 62.5%.

Figure 36 presents the boxplot of the raw data regarding the test coverage and time spent during the tasks using the subjects' experience. Thus, the graphs in this figure use the data from the Group A in Task I and the Group B in Task II. Also, they present the data by the group of students and professionals who participated in the experiment.

Figure 36 – Test coverage and time spent in Tasks conducted with experience-based testing



Source – the author

As presented in Figure 36, the data from professionals and students have *outliers* in the data related to test coverage. These *outliers* correspond to the participants S6 and S7 from Table 16, who achieved a 100% coverage rate. Besides that, with regards to the tests coverage, the data from the professionals have a symmetric distribution (i.e., the data is evenly split at the median) and a median value higher than the students.

In the data related to time spent, there is a *outlier* only in the group of professionals. This data is from the professional S6 (see Table 16) that spent more time than the other ones. A possible reason is that from the professional group, only the subject S6 has knowledge on DSPLs.

However, more experiments are needed to draw conclusions about that. Also, the professionals have a time spent median slightly higher than the students. Moreover, the values of time spent for the students are quite different.

5.2.4.2 Qualitative Data

As presented in Subsection 5.2.2.4, the subjects filled out forms after the tasks and at the end of the experiment. The goal of the forms after the tasks (*Post-Task Questionnaire*) was to acquire information about the feeling of the subject about: (i) the easiness of the task; (ii) the training session; (iii) the task goals; and (iv) whether the coverage of the created tests is good or not. For these questions, it was used a *Likert* scale with five levels from Strongly Agree to Strongly Disagree.

Table 17 presents the results of the *Post-Task Questionnaire* related to Task I. In Group A, which generated the tests based on the experience, all subjects "Strongly Agree" that the task was easy and that the goals were clear. Besides that, they reported that the goals were clear and the test coverage was good. In the Group B, which used the TestDAS, all subjects "Strongly Agree" that the task was easy, the training was enough and the tests coverage was good. Also, they stated that the goals were clear.

Table 17 – Subjects' feedback regarding the Task I (Mobiline)

Group	Scale	Easy Task	Enough Training	Clear Goals	Good tests coverage
A	Strongly Disagree	0 (0%)	0 (0%)	0 (0%)	0 (0%)
	Disagree	0 (0%)	0 (0%)	0 (0%)	0 (0%)
	Neutral	0 (0%)	0 (0%)	0 (0%)	0 (0%)
	Agree	0 (0%)	1 (17%)	0 (0%)	2 (33%)
	Strongly Agree	6 (100%)	5 (83%)	6 (100%)	4 (67%)
B	Strongly Disagree	0 (0%)	0 (0%)	0 (0%)	0 (0%)
	Disagree	0 (0%)	0 (0%)	0 (0%)	0 (0%)
	Neutral	0 (0%)	0 (0%)	0 (0%)	0 (0%)
	Agree	0 (0%)	0 (0%)	2 (33%)	0 (0%)
	Strongly Agree	6 (100%)	6 (100%)	4 (67%)	6 (100%)

Source – The author

Aiming to assess the agreement among the answers from the subjects presented in Table 17, it was measured the Randolph's free-marginal multirater Kappa (RANDOLPH, 2005). For this purpose, the cases were the topics addressed (Easy Task, Enough Training, Clear Goals, Good tests coverage) in the *Post-Task Form* and the categories were the values of the Likert scale used. As a result, for the Group A the free-Kappa value was 0.72, whereas for the Group B

the free-Kappa value was 0.83. Thus, by using the interpretation provided by Landis and Koch (1997)[4], there is a substantial agreement in the Group A and an almost perfect agreement in the Group B. This high agreement inside the groups was expected since the subjects have a similar background (see Section 5.2.2.3) and performed the Task I by using the same treatment.

Table 18 presents the results of the *Post-Task Questionnaire* related to the Task II. In the Group A, which used the TestDAS in this task, all subjects stated that the task was easy, the training was enough and the task goals were clear. Regarding the test coverage, five of the subjects reported that the coverage was good, whereas one subject was undecided ("Neutral"). In the Group B, which created the tests based on the experience, all subjects reported that the training was enough, the goals were clear and the test coverage was good. However, only four subjects stated that the task was easy, whereas one did not report this task as easy ("Disagree") and another one was undecided ("Neutral").

Table 18 – Subjects' feedback regarding the Task II (SmartHome)

Group	Scale	Easy Task	Enough Training	Clear Goals	Good tests coverage
A	Strongly Disagree	0 (0%)	0 (0%)	0 (0%)	0 (0%)
	Disagree	0 (0%)	0 (0%)	0 (0%)	0 (0%)
	Neutral	0 (0%)	0 (0%)	0 (0%)	1 (17%)
	Agree	5 (83%)	1 (17%)	0 (0%)	1 (17%)
	Strongly Agree	1 (17%)	5 (83%)	6 (100%)	4 (66.6%)
B	Strongly Disagree	0 (0%)	0 (0%)	0 (0%)	0 (0%)
	Disagree	1 (17%)	0 (0%)	0 (0%)	0 (0%)
	Neutral	1 (17%)	0 (0%)	0 (0%)	0 (0%)
	Agree	2 (33%)	2 (33%)	1 (17%)	4 (67%)
	Strongly Agree	2 (33%)	4 (67%)	5 (83%)	2 (33%)

Source – The author

In order to assess the agreement among the answers from the subjects presented in Table 18, it was measured the Randolph's free-marginal multirater Kappa (RANDOLPH, 2005). Again, the cases were the topics addressed in the *Post-Task Form* and the categories were the values of the Likert scale used. As a result, for Group A the free-Kappa value was 0.60, whereas for Group B the free-Kappa value was 0.29. Thus, there is a moderate agreement (LANDIS; KOCH, 1977) in Group A. However, in Group B, the value of free-Kappa indicates only a fair agreement. By analyzing the answers of the subjects from the Group B, the main differences were in the question addressing the easiness of the task. A possible reason for this discrepancy is

[4] 0 - poor agreement; between 0.0 and 0.20 - a slight agreement; between 0.21 and 0.40 - fair agreement; between 0.41 and 0.60 - moderate agreement; between 0.61 and 0.80 - substantial agreement; and between 0.81 and 1.00 - almost perfect agreement

that some of the subjects of the Group B felt it was more easy to create the tests by using the TestDAS, as they did in Task I.

The post-task forms also had open questions related to general comments (optional) and the testing criterion applied (mandatory in Task conducted based on the subjects' experience). In the latter, all subjects reported that created the tests to cover the contexts, and one of them stated that also focused on covering all possible combinations of context. With regards to the comments, the subjects' feedback is summarized as follows:

- *"The excel file (used by the TestDAS tool to specify the context variation model) did not make easy to see the context model"*;
- *"The method presents a good coverage and is easy to understand to who already know DSPL concepts"*;
- *"Manual testing is costly"*; and
- *"Without the method the effort increased to generate tests cases and I am not sure if all cases were covered"*.

The form filled out at the end of the experiment (*Post-Experiment Questionnaire*) aimed to collect feedback about which treatment (using TestDAS or based on the subjects' experience) has better coverage and creates tests in less time. Table 19 presents the results of this form. In Group A, most of the subjects (83%) stated that the TestDAS has a better coverage and generate more tests, whereas 50% stated that it spent less time than manual testing. On the other hand, in Group B, all subjects reported that using the TestDAS they achieved a better coverage, generated more tests and that they spent less time to generate tests.

Table 19 – Subjects' answers in the Post-Experiment Form

Group	Scale	TestDAS has better coverage	TestDAS creates tests in less time	TestDAS creates more tests
A	Strongly Disagree	0 (0%)	0 (0%)	0 (0%)
	Disagree	0 (0%)	0 (0%)	0 (0%)
	Neutral	1 (17%)	3 (50%)	1 (17%)
	Agree	2 (33%)	0 (0%)	2 (33%)
	Strongly Agree	3 (50%)	3 (50%)	3 (50%)
B	Strongly Disagree	0 (0%)	0 (0%)	0 (0%)
	Disagree	0 (0%)	0 (0%)	0 (0%)
	Neutral	0 (0%)	0 (0%)	0 (0%)
	Agree	0 (0%)	0 (0%)	0 (0%)
	Strongly Agree	6 (100%)	6 (100%)	6 (100%)

Source – The author

When asked about which was the best way to specify the tests, all subjects (100%) stated that to specify tests with the TestDAS is better than to specify tests based on the tester's experience.

5.2.4.3 Hypothesis Testing

In order to analyze the experiment results, statistical tests were applied using the SPSS tool (IBM, 2017). First, it was used the Kolmogorov-Smirnov test (HOLLANDER; WOLFE, 1999; WOHLIN *et al.*, 2014) to assess if it is reasonable to assume that the data sets from the experiment come from a normal distribution. Table 20 presents the results of this test for three variables measured in the experiment: number of tests, test coverage and time spent.

Note that the p-values for the Kolmogorov-Smirnov tests related to the variables number of tests and test coverage are near 0.000 (in the row "Asymp. Sig."). This implies that these data set have not a normal distribution, because the p-value was smaller than the significance level (0.05). On the other hand, the p-value for the variable time spent (0,0771) is higher than 0.05, which means that the distribution of this data set corresponds to the theoretical distribution.

Table 20 – Data set normality test

Description	#Tests	Test Coverage	Time Spent
Kolmogorov-Smirnov Z	1,858	1,855	0,664
Asymp. Sig (2 tailed)	0,002	0,002	0,771

Source – The author

Therefore, the Kolmogorov-Smirnov tests indicated that the data set of two variables did not follow a normal distribution and, thus, it was necessary to use non-parametric tests for the hypothesis testing. In this case, it was applied the Mann-Whitney test (WOHLIN *et al.*, 2014), which is a non-parametric test of the null hypothesis and that has a greater efficiency than the t-test on non-normal distributions.

Table 21 presents the results of the hypothesis testing performed in order to compare the tests generated by Groups A and B in Task I, with regards to the number of tests, time spent and tests coverage. It is worth noting that in Task I, the subjects from Group A used experience-based testing (Treatment 2), whereas the subjects from Group B used the TestDAS (Treatment 1). So, the values presented in this table are the asymptotical significance of the

comparison between the treatments, using the Mann-Whitney test. Values below 0.05 indicate a statistically significant difference between the results. Thus, the results indicated a statistically significant difference between the use of the treatments concerning the number of test cases (*p-value* = 0.005 $<$ 0.05) and the tests coverage (*p-value* = 0.006 $<$ 0.05). On the other hand, the use of the two treatments have shown results statistically equal to each other with regards to the time spent in this task (*p-value* = 0,522 $>$ 0.05).

Table 21 – Comparison between the treatments in Task I

Description	#Tests	Test Coverage	Time Spent
Mann-Whitney U	3,000	3,000	14,000
Asymp. Sig (2 tailed)	0,005	0,006	0,522

Source – The author

Table 22 presents the results of the Mann-Whitney test for Groups A and B in Task II. Again, there is a statistically significant difference between the use of the treatments concerning the number of test cases (p-value = 0,005 $<$ 0,05) and the test coverage (p-value = 0,007 $<$ 0,05). However, the use of the two treatments have shown results statistically equal to each other with regards to the time spent in Task II (*p-value* = 0.423 $>$ 0.05).

Table 22 – Comparison between the treatments in Task II

Description	#Tests	Test Coverage	Time Spent
Mann-Whitney U	3,000	3,000	13,000
Asymp. Sig (2 tailed)	0,005	0,007	0,423

Source – The author

Based on the results depicted in Table 21 and Table 22, the null hypothesis HN_0 and HC_0 can be rejected and their respective alternative hypothesis (HN_1 and HC_1) can be accepted. On the other hand, the null hypothesis HT_0 cannot be rejected since there was not found statistically significant difference in the time spent by the use of the treatments.

5.2.5 Discussion

The evaluation of the TestDAS focused on to investigate three questions: *(Q2) Does the number of the tests is increased when the TestDAS is followed?*; *(Q3) Does the coverage of the tests is increased when the TestDAS is followed?*; and *(Q4) Does the time spent to create*

adaptation test cases is lower when the TestDAS is followed?.

With regards to the number of tests (*Q2*), the data of the controlled experiment showed that the number of test cases in the test sequences generated by the TestDAS is higher than the number of test cases created by the subjects that had used only experience-based testing. Besides, the data set from the experiment showed that the coverage of the tests generated (*Q3*) by the TestDAS is higher than the coverage achieved by the tests created manually using the subjects' experience. Regarding the time spent (*Q4*) to generate the tests with TestDAS and manually using the subjects' experience, the hypothesis testing did not reject the null Hypothesis HT_0, since the p-value was smaller than 0.05, which is out of the confidence interval of 95%. So, the data from the experiment showed that there was no gain using the TestDAS, regarding the time spent to generate the adaptation test sequences.

Beyond the quantitative data collected in the experiment, forms filled out by the subjects after the experiment tasks and after the entire experiment provided qualitative data addressing the TestDAS and the experiment itself. In these forms, all subjects reported that the training was enough and the task goals were clear. Concerning the tasks, all subjects using the TestDAS stated that the tests generation was easy. On the other hand, two of the subjects did not agree that the tests creation was easy using only their experience.

Furthermore, only one (8%) from the twelve subjects did not agree (he stated "Neutral") that the TestDAS has better coverage than experience-based testing. Also, only one subject did not agree that TestDAS generates more tests than the experience-based testing. Thus, these data corroborate with the results obtained from the quantitative data.

With regards to the time spent, three of the subjects from the Group A did not agree that TestDAS generate tests in less time. Nevertheless, all subjects from Group B reported that the TestDAS generated the tests faster than the tests creation based on their experience. In fact, as depicted in Subsection 5.2.4.1, the value of the time spent by the Group B with the TestDAS was slightly lower than the time spent in the manual creation of the tests. Thus, there is a divergence with the quantitative data since they did not show a statistically significant difference between the time spent with TestDAS and with the tests creation based on the subjects' experience.

Therefore, the collected data indicate the effectiveness of the TestDAS regarding the number of tests and test coverage compared to the manual tests creation using the subject's experience. This result was expected since the test coverage criteria used in TestDAS were proposed based on a set of behavioral properties related to the DAS adaptive behavior.

Also, despite most of the subjects had reported in the qualitative forms that the Test-DAS spent less time than experience-based testing, this was not corroborated by the quantitative data. Thus, the experiment data does not show evidence that the TestDAS has gain regarding the time spent to create the tests. The main reason for this result, according to the observations during the experiment, is the need to specify manually the DAS context variation model to use the TestDAS.

5.2.6 Threats to Validity

This section discusses the threats to the validity of the experiment results with regards to: (i) *Conclusion Validity*; (ii) *Construct Validity*; (iii) *Internal Validity*; and (iv) *External Validity*.

Threats to the *Conclusion Validity* concern the relationship between the treatment and the outcome (WOHLIN *et al.*, 2000). One of these threats refers to the reliability of the measures used. In order to have an objective measure (independent from a human judgment), it was defined a measure related to the coverage of the context and adaptation actions over the features. The background of the subjects and the division into the groups are also threats to this validity. Regarding the subjects, it was required a previous knowledge in Software Testing, and a training session was performed to balance the knowledge of the concepts used during the experiment. In order to mitigate the threat related to the division into the groups, the subjects were randomly assigned to the groups, but keeping a balanced number of students and professionals in each one.

Other threats to the *Conclusion Validity* are the low statistical power and the violation of the assumptions of statistical tests (WOHLIN *et al.*, 2000). Both of them can lead to wrong conclusions. Aiming to mitigate such threats, first, it was applied the Kolmogorov-Smirnov test (HOLLANDER; WOLFE, 1999) to assess if the data follow a normal distribution. Next, since the data did not have a normal distribution, it was applied the Mann-Whitney test (WOHLIN *et al.*, 2014) that is a non-parametric statistical test.

The *Construct Validity* concern generalizing the results of the experiment to the theory behind the experiment (WOHLIN *et al.*, 2000). The main threats are related to the experiment design, for instance, the mono-operation bias (WOHLIN *et al.*, 2000). To mitigate this kind of threats, the experiment used two different objects and the experiment design was previously assessed through a pilot study. Another possible threat is the under representation of the construct, since only parts of academic DAS were used in the experiment. A larger DAS

could yield a more reliable data set, but it could not be possible to execute the experiment tasks in an acceptable time.

As mentioned in Section 5.1.5, threats to *Internal Validity* are influences that can affect the independent variable with respect to causality (WOHLIN *et al.*, 2000). In this case, a threat is the learning effect, *i.e.*, Maturation (WOHLIN *et al.*, 2000). To mitigate this effect, the subjects applied the treatment in different orders. The subjects from Group A used experience-based testing in Task I and the TestDAS in Task II, whereas the subjects from Group B used the TesTDAS in Task I and only their experience in Task II. Thus, all subjects used the two treatments of the experiment, where the order of the treatment used was defined randomly. This also avoids social threats, such as (WOHLIN *et al.*, 2000): (i) compensatory equalization of treatments, when participants may wish they were in the other group and this affect the performance of them; (ii) compensatory rivalry, when there is a rivalry between the groups and this affect the participants' performance; and (iii) resentful demoralization, when due to the treatment used, the participant is not motivated and not perform as good as it generally does.

Another threat to the *Internal Validity* is the selection of the subjects that was made based on convenience sampling (WOHLIN *et al.*, 2014). However, it is worth noting that they were randomly assigned to the groups (A or B). Also, to mitigate the threat of a bad instrumentation, it was assessed during the pilot study and its results pointed out improvements that were made in the experiment instrumentation.

As mentioned in Section 5.1.5, threats to *External Validity* are conditions that limit the ability to generalize the results to industrial practice (WOHLIN *et al.*, 2000). The main threat to this validity is related to the DSPLs (Mobiline and SmartHome) used, since they are academic DAS; and in the experiment, it was used only part of their feature models. An experiment with a larger DAS would demand effort incompatible with the time available for the experiment. Furthermore, despite the participation of professionals, the experiment also has students as subjects, which might be not representative of the population.

5.3 Assessment of the Supporting Tools

As presented in Section 4.5, the TestDAS tool and the CONTroL were implemented to support the use of the TestDAS method. The TestDAS tool helps in the model checking process, as well as in the generation of the test sequences for a given DAS. The CONTroL, in turn, aims to support the execution of the test sequences in the application under testing.

In this way, the goal of this evaluation was to investigate the feasibility of using these tools during the activities of the TestDAS. The evaluation questions from **Q1** to **Q4** were defined to the Mutant Analysis (Section 5.1) and Controlled Experiment (Section 5.2). Therefore, for the tool evaluation, it was defined the question **Q5** as follow:

> **(Q5)** *Is the use of TestDAS tool and CONTroL feasible to perform the TestDAS method activities?*

In order to answer Q5, it was performed an Observational Study (KITCHENHAM *et al.*, 2002), in which the participants used both the TestDAS tool and the CONTroL during the execution of the TestDAS method and after that, they provided feedback concerning these supporting tools.

With regards to the participants of this study, they were the same one that performed the experiment tasks (see Section 5.2.2.3). The use of the same subjects in both evaluations brings a threat to the validity, as discussed in Section 5.3.4. However, the participants performed different tasks in these evaluations, and the goal in the observational study was to collect the users' feedback about the tools implemented, whereas the experiment focused on the comparison between TestDAS and experience based testing. It is also important to mention that CONTroL was not used in the experiment described in Subsection 5.2 since it would require the instrumentation of all source code of the Mobile Guide and Smart Home, and this would make the experiment more difficult and time-consuming to execute.

The following subsections are organized as follows. Subsection 5.3.1 describes the design and execution of the observational study. Subsection 5.3.2 presents the results obtained from this evaluation. Subsection 5.3.3 discusses the results to answer the question investigated. Finally, the threats to the validity of the results are depicted in Subsection 5.3.4.

5.3.1 Design and Execution of the Observational Study

As mentioned before, the goal of the observational study was to assess the feasibility of using the TestDAS tool and the CONTroL to perform the TestDAS activities depicted in Chapter 4. For this purpose, the observation study execution was organized into three phases, described as follows:

- **Phase I - Preparation**. In this phase, the objectives of the study were introduced to the participants. Also, this phase involved a training session about the main concepts regarding

the model checking approach;

- **Phase II - Use of the TestDAS tool**. The first activity of this phase was a training session on the functionality of TestDAS tool, presented in Section 4.5. Then, the subjects were required to run the model checking and to generate a test sequence using the TestDAS tool. For these tasks, they received a JSON (JavaScript Object Notation)[5] file with a context-aware feature model and a document describing the study object and the context variation model. After uploading the JSON file to the TestDAS tool and filling out the adjacency matrix generated with the context variation model, the participants performed the checking of the properties defined in Section 4.3.2 and generated the test sequences. At last, the subjects filled out a *Feedback Form - TestDAS tool* (see Appendix F) with their opinion about the tool; and
- **Phase III - Use of the CONTroL**. In this phase, first, the subjects received training about the CONTroL (see Section 4.5). Next, they used the CONTroL to include annotations (i.e., @*ControlContext*, @*ControlFeature* and @*ControlSystemAdapted*) in an application under testing. Then, they ran the test sequence generated in Phase II in this application and analyzed the test results exported by the tool in an HTML file. In the last activity, the subjects were asked to provide feedback concerning the use of the CONTroL by filling out the *Feedback Form - CONTroL*, which is available at Appendix F.

It is important to highlight that in this evaluation the participants were not required to fill out a background form, since they had already done this in the controlled experiment (see Section 5.2). Thus, the background of the participants of the observational study can be seen in Section 5.2.2.3. Besides that, in this evaluation, the subjects were not divided into groups since the goal was not to compare different treatments, but indeed collect evidence regarding the feasibility of using the supporting tools implemented.

With regards to the study object, it was used the Mobile Guide DSPL (see Section 2.1) and the GREat Tour application (MARINHO *et al.*, 2013) in the Phases II and III, respectively. The GREat Tour is an Android application generated from the Mobile Guide, which self-adapts according to the visitor's context to provide texts, images or videos. This application follows the adaptation rules presented in Table 1, concerning the running example depicted in the Chapter 1.

The *feedback* forms contains open and closed questions. The closed questions concern the opinion of the subject regarding the easiness of use and whether they would like to

[5] www.json.org/

use the tool in the testing/verification of DAS. For this kind of questions, it was used the five-point Likert scale (ROBINSON, 2014) varying from “Strongly Disagree” to “Strongly Agree”. The open questions, in turn, are related to difficulties faced by the participant and suggestions for improving the tools. It is worth noting that the forms’ questions were defined to provide initial evidence regarding the user perception. A complete evaluation, for instance, using the TAM (Technology Acceptance Model) (VENKATESH; DAVIS, 2000; DAVIS, 1986), would require more time and effort, and thus it was left as future work.

In addition to the qualitative data collected by the feedback forms, the time spent in the tasks was monitored to observe whether the participants have a similar performance.

5.3.2 Results

Table 23 presents the time spent by the participants during the tasks of the observational study. The mean of the time spent in the tasks (model checking and test sequence generation) with the TestDAS tool was 5m 38s with a standard deviation equals to 36 seconds. In the task related to the test execution with CONTroL, the mean of time spent was 5m 6s, with SD equals to 41 seconds. By observing these standard deviation values, it was possible to note that the data tend to be close to the mean and, thus, the participants spent a similar time in the tasks of the study.

Table 23 – Time spent by the subjects in the tasks with TestDAS tool and CONTroL

Subject	Time Spent (m:s)	
	TestDAS Tool	CONTroL
S1	5:00	4:00
S2	5:44	5:23
S3	5:36	6:00
S4	5:31	5:55
S5	5:42	5:00
S6	6:32	5:09
S7	6:06	5:15
S8	6:47	6:00
S9	5:32	5:06
S10	4:34	4:36
S11	5:28	4:03
S12	5:12	4:50
Mean	5:38	5:06
SD	0:37	0:41

Source – The author

Table 24 presents the results of the closed questions in the *Feedback Form - TestDAS*

tool filled out in the Phase II. Most of the participants stated that it was easy to perform the model checking with the TestDAS tool and that they would like to use this tool for supporting the model checking in DAS. Regarding the generation of test sequences with the TestDAS tool, most of the subjects reported that it was an easy task, whereas all of them stated that would like to use the TestDAS tool for testing DAS.

Table 24 – Subjects' feedback regarding the TestDAS Tool

Scale	Model Checking		Test Generation	
	Easy to use	Would use	Easy to use	Would use
Strongly Disagree	0 (0%)	0 (0%)	0 (0%)	0 (0%)
Disagree	0 (0%)	0 (0%)	0 (0%)	0 (0%)
Neutral	2 (16%)	1 (8%)	1 (8%)	0 (0%)
Agree	5 (42%)	4 (33%)	3 (25%)	1 (8%)
Strongly Agree	5 (42%)	7 (59%)	8 (67%)	11 (92%)

Source – The author

According to the feedback of the participants, the main difficulties concerning the use of the TestDAS tool were related to the understanding of the model checking log and the use of an excel file to specify the context variation model. In this way, the main suggestion for the improvement of this tool refers to the creation of an interface to support the specification of the context variation model. Other suggestions include improvements in the user interface and the creation of a Domain Specific Language (DSL) for the user specify her/him own properties to be checked.

Table 25 presents the results of the closed questions in the *Feedback Form - CONTroL* filled out in the Phase III. As depicted in this table, most of the participants stated that the use of the CONTroL was easy. Also, all of them would like to use this tool during the test execution over a DAS.

Table 25 – Subjects' feedback regarding the CONTroL

Scale	Easy to use	Would use
Strongly Disagree	0 (0%)	0 (0%)
Disagree	0 (0%)	0 (0%)
Neutral	3 (25%)	0 (0%)
Agree	4 (33%)	1 (8%)
Strongly Agree	5 (42%)	11 (92%)

Source – The author

In the open questions, five of the subjects reported that the main difficulty for the use of the CONTroL is related to the code understanding and the instrumentation task. For the improvement of this tool, three of the participants suggested automating the code annotation. Also, two subjects stated that would be interesting to generate the test report in the mobile device, since in the observational study the test results were exported from the device to a notebook in order to generate the corresponding *HTML* file.

5.3.3 Discussion

The evaluation of the feasibility of the implemented supporting tools was conducted in order to answer the following task: *(Q5) Is the use of TestDAS tool and CONTroL feasible to perform the TestDAS method activities?*. For this purpose, an observational study was performed, in which the participants used the TestDAS tool and the CONTroL, and, next, they provided their feedback regarding these tools.

All subjects successfully finished the tasks of the observational study using the two tools evaluated. The results of this study showed that most of the participants felt that it was easy to run model checking (84%) and generate test sequences (92%) with TestDAS tool. Also, most of them (75%) reported that it was easy to use the CONTroL for running a test sequence. Therefore, the data from the PoC provided evidence that the use of both implemented tools is feasible to support the execution of the TestDAS method.

The major difficulty during the use of the TestDAS tool was the filling out of the excel file with the context variation model. With regards to the CONTroL, the major difficulty reported by the participants refers to the application code instrumentation. However, the results were positive and most of the participants would like to use the TestDAS tool and CONTroL for DAS model checking (92%) and testing (100%).

It is worth noting that the participants did not specify the feature model, since it is not a goal of the TestDAS tool to provide support to feature modeling. Besides that, the current implementation of the TestDAS tool is not integrated with a feature modeling tool. Thus, the user should upload a JSON file with the feature model.

Moreover, the CONTroL depends on the programming language of the application under testing. Currently, the CONTroL supports Java and Android applications, but to another kind of application (e.g., IOS application), adaptations in the CONTroL code can be needed.

5.3.4 Threats to Validity

The main threats to the validity of the results from the observational study concern the internal validity and external validity.

The main threat related to the internal validity is the selection of the participants. Since the participants of the observational study were the same from the controlled experiment (see Section 5.2), the results of this study can be affected by the learning of the participants. However, the set of tasks in this study was different from the controlled experiment. In the latter, the participants did not perform model checking and did not run the tests.

With regards to the external validity, the main threat is the study objects used, because the Mobile Guide and the GREat Tour are small and academic DAS. Thus, spite of the positive results in using the TestDAS tool and CONTroL, these results can not be generalized to all kind of DAS.

5.4 Conclusion

In this chapter, three evaluations performed to assess the TestDAS were presented, as well as the feasibility study of two supporting software, TestDAS tool and CONTroL. Thus, this chapter described the following evaluations: (i) an evaluation of the model checking approach of the TestDAS by using a mutant analysis; (ii) a controlled experiment to compare the TestDAS with experience-based testing regarding the time spent to create the tests, tests coverage and number of tests; and (iii) an observational study to assess the feasibility of using the TestDAS tool to apply the TestDAS method, and the CONTroL to perform test sequences over the application under testing.

The first evaluation investigated how effective is the model checking approach proposed. For this purpose, 114 mutants were created from a correct DSPL design, and then the model checking approach was applied in each mutant to compute the number of killed mutants (i.e., identified by the approach as a faulty model). As a result, the approach successfully identified all DAS specifications with design faults.

The second evaluation was performed through a controlled experiment and focused on the comparison between TestDAS and experience based testing. Twelve subjects participated in this experiment and provided quantitative and quantitative data that supported this comparison. The results of such evaluation pointed out evidence that the TestDAS has a better results than

experience-based testing for the testing of adaptive systems, regarding the number of test cases generated and the tests coverage. On the other hand, according to the experiment data, there is no gain in the time spent using the TestDAS to generate the test sequences.

The third evaluation, the observational study, assessed the feasibility of using the supporting tools implemented to perform the TestDAS method. In this evaluation, the participants used the TestDAS tool and the CONTroL and provided their feedback through forms. The data from this study revealed evidence that the use of these tools is feasible and they support the execution of the TestDAS. Also, most of the participants reported that was easy to use the TestDAS tool and CONTroL.

The next chapter concludes this thesis by revisiting the research hypothesis, and summarizing the results and publications during the thesis work period. Furthermore, it describes the perspectives of future work related to the contributions presented in Chapter 4.

6 CONCLUSION

This thesis presented a testing method, called TestDAS, for supporting the testing of the DAS adaptive behavior. It also introduced its supporting tools and discussed the evaluations performed.

This chapter concludes the thesis and is organized as follows. Section 6.1 depicts an overview of this thesis. Section 6.2 summarizes the main results of this thesis. Section 6.3 discusses the hypothesis investigated and compares TestDAS with related work. Finally, Section 6.4 introduces the limitations of this work, and Section 6.5 presents some insights for future work.

6.1 Overview

Dynamically Adaptive Systems (DAS) self-adapt according to the context information gathered from the surrounding environment (BENCOMO *et al.*, 2008). Such dynamic behavior is typically designed using adaptation rules, which are context-triggered actions responsible for software reconfiguration actions. Thus, faults in the adaptation rules can result in failures in the adaptive behavior of the DAS at runtime.

Given the complexity introduced by the use of context information as an adaptation trigger, the verification and validation activities are important activities to ensure the correctness of the DAS adaptive behavior. Therefore, methods and tools supporting these activities are needed to ensure quality for adaptive systems. As discussed in Chapter 3, in the literature there is a lack of a formalism that represents the effects of adaptation rules in the DAS behavior, allowing the checking of properties related to the adaptation rules design and the (de)activation of features. Besides, the existing testing methods do not ensure the coverage of the adaptation rules effects (i.e., feature activation/deactivation). For example, these methods do not guarantee the coverage of DAS configurations resulting from the adaptation rules interleaving or the (de)activation of all system features by the adaptation rules.

Aiming to address this gap, the goal of this research was *to propose a method for testing the DAS adaptive behavior focused on the effects of the adaptation rules*. This thesis achieved this goal by proposing the TestDAS (described in Chapter 4), which is a testing method that includes: (i) a set of test coverage criteria used to generate test sequences to validate the adaptive behavior of dynamic systems; (ii) a model, called Dynamic Feature Transition System

(DFTS), to specify the DAS adaptive behavior; (iii) a set of behavioral properties that DAS should satisfy; and (iv) a model checking approach that can identify design faults in the adaptation rules.

Furthermore, two supporting tools were implemented: (i) TestDAS tool, which supports the generation of test sequences for validating the DAS adaptive behavior, as well as the checking of properties over the DAS design; and (ii) CONTroL, which supports the execution of test sequences to validate the DAS adaptive behavior.

To assess the TestDAS and the supporting tools, three evaluations were performed, as described in Chapter 5. The first one was carried out using a mutant analysis to evaluate if the model checking approach can identify design faults in the adaptation rules. The second one aimed to compare through a controlled experiment the tests generated by TestDAS and the experience based testing. The last one focused on the feasibility of the supporting tools implemented, and it was conducted through an observational study.

The results of all these evaluations showed evidence regarding the benefits of the TestDAS and its tools. The mutant analysis gathered data that showed that the model checking approach proposed is effective in the identification of behavioral fault patterns. In the controlled experiment, the data analysis concluded that TestDAS generates more tests and achieves a better coverage than the tests created based on the tester's experience. Also, the observational study shows that the TestDAS tool and CONTroL support the TestDAS activities successfully.

6.2 Main Results

The main results of this thesis, which were presented in Chapter 4 and evaluated in Chapter 5, are summarized as follows:

- **TestDAS method**. This method supports the generation of test sequences to validate the adaptive behavior of dynamic systems based on a set of five coverage criteria. The TestDAS also involves a model for the DAS adaptive behavior, called Dynamic Feature Transition System (DFTS), and an approach for model checking that helps in the identification of design faults in DAS specifications;
- **Set of behavioral properties**. The five properties defined can be used to identify design faults in the DAS specification, since they are related to behavior fault patterns;
- **TestDAS tool**. This tool supports the software engineer to use the DFTS to generate test sequences, as well as to identify faults in adaptation rules design; and
- **CONTroL**. A library implemented in the Java language that performs the test sequences

generated by the TestDAS on the DAS under testing.

Furthermore, five papers were published in conferences and journals from the research performed in this thesis work. Also, one paper was accepted for publication. Table 26 presents the references of these papers. The first paper (SANTOS *et al.*, 2017) presents an approach to develop DAS by using an variability modeling technique and the model checking approach depicted in Section 4.3. The next paper (SANTOS *et al.*, 2017) concerns the systematic review performed on context-aware testing, which was important to identify the existing test coverage criteria related to dynamic adaptation. The DFTS (see Section 4.2) and the behavioral properties proposed (see Section 4.3.2) were introduced in the paper (SANTOS *et al.*, 2016). The paper (SANTOS *et al.*, 2015b) introduced the initial proposal of this thesis work. The papers (SANTOS *et al.*, 2015a) and (SANTOS *et al.*, 2014), in turn, describe results from the literature review activity (see Section 1.5) related to the software variability and SPL domain.

Table 26 – Papers from this thesis work

Reference	Qualis	Status
SANTOS, I. S.; SOUZA, M. L. J.; CARVALHO, M. L. L.; OLIVEIRA, T. A.; ALMEIDA, E. S.; ANDRADE, R. M. C. *Dynamically Adaptable Software is All about Modeling Contextual Variability and Avoiding Failures*. IEEE Software, 2017b.	A1	Accepted
SANTOS, I. S.; ANDRADE, R. M. d. C.; ROCHA, L. S.; MATALONGA, S.; OLIVEIRA, K. M. de; TRAVASSOS, G. H. *Test case design for context-aware applications: Are we there yet?*. Information and Software Technology, Butterworth-Heinemann, Newton, MA, USA, v. 88, n. C, p. 1–16, ago. 2017a. ISSN 0950-5849	A2	Published
SANTOS, I. S.; ROCHA, L. S.; NETO, P. A. S.; ANDRADE, R. M. C. *Model Verification of Dynamic Software Product Lines*. In: Proceedings of the 30th Brazilian Symposium on Software Engineering. New York, NY, USA: ACM, 2016. (SBES '16), p. 113–122.	B2	Published
SANTOS, I. S.; ANDRADE, R. M. C.; NETO, P. A. S. *Um método para geração otimizada de testes a partir de requisitos para linhas de produto de software dinâmicas*. In: Proceedings of the V Workshop de Teses e Dissertações do CBSoft, 2015b	-	Published
SANTOS, I. S.; ANDRADE, R. M.; NETO, P. A. S. *Templates for textual use cases of software product lines: results from a systematic mapping study and a controlled experiment*. Journal of Software Engineering Research and Development, v. 3, n. 1, 2015a	B3	Published
SANTOS, I. S.; ANDRADE, R. M. C. C.; NETO, P. A. S. *How to Describe SPL Variabilities in Textual Use Cases: A Systematic Mapping Study*. In: Eighth Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS), 2014. p. 64–73	B3	Published

Source – the author.

Other 19 papers were also published during the thesis work period. Although they do not present results from this thesis work, they were important for the improvement of the research skills such as critical thinking, research methods, and academic writing. The main topics addressed by these papers were: Test Generation, Testing Process, Agile Methods, Ubiquitous

Systems, Teaching of Software Engineering, and Software Measures. The references of these 19 published papers are presented in Appendix G.

6.3 Revisiting the Research Hypothesis and Related Work

In Chapter 1, it was presented the research hypothesis that guided this thesis work. So, based on the results presented in Chapter 4 and the evaluations described in Chapter 5, it is possible to analyze (accept or reject) this research hypothesis. This analysis is presented as follows:

Research Hypothesis: *A DAS testing method using a model to specify the DAS features configuration based on adaptation rules provides better coverage of the adaptive behavior and supports the identification of faults in the adaptation rules design.*

Hypothesis Analysis: **Accepted**. Based on the evaluations presented in Chapter 5, it was possible to gather evidence that the TestDAS supports the generation of adaptation test sequences achieving a better coverage of the DAS adaptive behavior. This method also supports the model checking for the faults identification in the adaptation rules of DAS. Furthermore, it was possible to observe that the Dynamic Feature Transition Systems (DFTS), which models the features configuration based on the context changes and adaptation rules, supports the TestDAS during the DAS testing and model checking.

With regards to the related work, Chapter 3 points out that there is a lack of a formalism to represent the effects of adaptation rules in the DAS configuration, allowing the checking of properties related to the adaptation rules and (de)activation of features. Also, only one work (LOCHAU *et al.*, 2015) supports the use of a well-known model checker, and the checking of properties related to fault patterns, as well as user-defined properties. In this scenario, the differential of TestDAS is that it models the DAS configurations based on the context and adaptation rules triggered. Thus, the proposed method supports the checking of properties to reason about the effects of the adaptation rules over the DAS features. Also, it uses the SPIN and not only supports the identification of adaptation fault patterns, but also supports the checking of properties defined by the software engineer. Table 27 presents TestDAS according to the criteria

used in Chapter 3 (see Table 4) to present the studies related to DAS model checking.

Table 27 – TestDAS in comparison to the work related to DAS model checking.

Work	System Model	Tool	Properties Checked	
			Fault Patterns	User-Defined
TestDAS	Dynamic Feature Transition System	SPIN	Yes	Yes

Source – the author.

Regarding the related work to DAS testing, described in Chapter 3, most of the studies propose black-box approaches and some studies propose context-based test coverage criteria. However, even measuring the context coverage, the existing test coverage criteria do not ensure the coverage of the effects of the adaptation rules over the DAS features. Thus, there is a lack of approaches to guide the DAS testing that take into account the coverage of the adaptation rules actions. TestDAS addresses this gap by generating adaptation test sequences based on the context and adaptation rules. For this purpose, it uses a model of the DAS adaptive behavior (i.e., the Dynamic Feature Transition System) that is built based on the DAS feature model and the context variation model. Table 28 presents TestDAS according to the criteria used in Chapter 3 (see Table 5) to present the studies related to DAS testing.

Table 28 – TestDAS in comparison to the work related to DAS testing.

Work	Test Technique	Test Type	Context-Based Coverage
TestDAS	Adaptation Test Sequences	Black-Box	Yes

Source – the author.

6.4 Limitations

The method proposed in this thesis, called TestDAS, contributes to the quality assurance of dynamic systems since the design from the implementation. However, TestDAS has some limitations. TestDAS does not address the runtime testing or model checking. In this way, it does not concern the verification and validation of changes in the structural variability at runtime, for example, the inclusion of new context-aware features while the DAS is running (CAPILLA *et al.*, 2014b).

Besides that, TestDAS addresses only atomic adaptations and, thus, it does not allow changes in the context state during the test of a DAS reconfiguration. Also, the tools implemented

to support the TestDAS help in the use of this method, but they lack usability and have limitations. For instance, the TestDAS tool does not address feature attributes and CONTroL only handles Java and Android applications.

Lastly, the evaluation performed used only academic DSPLs and a low number of participants. So, there is no evidence regarding the use of TestDAS in large DSPLs. It is also necessary the replication of these evaluations with other study objects and participants in order to gather more evidence concerning the TestDAS benefits.

6.5 Future Work

This thesis proposed a method, called TestDAS, for DAS testing that supports the identification of faults and failures concerning the DAS adaptive behavior. From the results of this thesis, the main future research directions are described as follows:

- *TestDAS at runtime*. The method proposed works at design time, identifying faults and failures before the deploy of DAS. So, an extension of this method to make it applicable at runtime could complement the testing and model checking at design time, since it could support the identification of failures that occur only at runtime. For instance, failures related to the inclusion of a new feature or a new adaptation rule while DAS is running.
- *Context changes during testing*. TestDAS does not address context changes during a adaptation of the DAS. Hence, it does not allow the context to vary during testing freely, not addressing the *truly context-aware testing* (SANTOS *et al.*, 2017)(MATALONGA *et al.*, 2017). Therefore, an extension of the TestDAS to consider context changes could take into account the unpredictable behavior of the context, which may change at any time.
- *Use of a Domain Specific Language*. In Chapter 4, a set of behavioral properties was presented to support the software engineer in the identification of faults related to the DAS design. He/she could also specify their properties and run them on SPIN by using the proposed mapping of the Dynamic Feature Transition System in Promela code. In this way, a Domain Specific Language (DSL)(KOSAR *et al.*, 2016) could make easier the creation of new properties to be checked. Also, a DSL could support the feature model specification in the TestDAS tool and, thus, it will not be required a third tool for the DAS modeling.
- *Automated code instrumentation with CONTroL*. As pointed out in Chapter 5, the main difficulty with regards to the CONTroL use was the code instrumentation. Thus, the

automatic code instrumentation using the CONTroL annotation might lead to reductions in the effort related to the use of this tool.

- *Automated generation of the context variation model*. The Context Kripke Structure, which is used by the TestDAS, models the evolution of the context of the DAS environment. Such model could be automatically generated by monitoring the DAS environment and inferring relationship among the contexts. This automatic generation is interesting to reduce the effort related to the use of TestDAS.
- *New empirical evaluations*. This thesis presented a controlled experiment regarding the test generation and a mutant analysis to assess the model checking approach. They provided evidence of the TestDAS benefits. However, new studies, in other domains and with more subjects, are needed to gather more evidence. Also, the tools implemented require evaluations regarding the scalability and usability.
- *Use of Search-Based Software Engineering*. By using the testing coverage criteria and the Search-Based Software Engineering (SBSE) (HARMAN *et al.*, 2012; HARMAN *et al.*, 2015), test sequences could be generated to optimize a given goal (e.g., to maximize the Interleaving Correctness Coverage while minimizing the size of the test sequence). Since the use of SBSE could generate optimal or near-optimal test sequences in an acceptable time, it could also address the testing of DAS with large feature models or large context variation models.

BIBLIOGRAPHY

ALMEIDA, E. S.; ALVARO, A.; GARCIA, V. C.; MASCENA, J. C. C. P.; BUREGIO, V. A. A.; NASCIMENTO, L. M.; LUCREDIO, D.; MEIRA, S. L. **C.R.U.I.S.E: Component Reuse in Software Engineering**. [S.l.]: C.E.S.A.R e-book, 2007.

ALVES, V.; SCHNEIDER, D.; BECKER, M.; BENCOMO, N.; GRACE, P. Comparative study of variability management in software product lines and runtime adaptable systems. In: **International Workshop on Variability Modelling of Software-intensive Systems**. [S.l.: s.n.], 2009. p. 9–17.

AMALFITANO, D.; FASOLINO, A. R.; TRAMONTANA, P.; AMATUCCI, N. Considering context events in event-based testing of mobile applications. In: **Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on**. [S.l.: s.n.], 2013. p. 126–133.

ARCAINI, P.; GARGANTINI, A.; RICCOBENE, E. Asmetasmv: A way to link high-level asm models to low-level nusmv specifications. In: **Proceedings of the Second International Conference on Abstract State Machines, Alloy, B and Z**. Berlin, Heidelberg: Springer-Verlag, 2010. (ABZ'10), p. 61–74. ISBN 3-642-11810-0, 978-3-642-11810-4.

ARCAINI, P.; GARGANTINI, A.; VAVASSORI, P. Automatic detection and removal of conformance faults in feature models. In: **2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)**. [S.l.: s.n.], 2016. p. 102–112.

ARCAINI, P.; RICCOBENE, E.; SCANDURRA, P. Formal design and verification of self-adaptive systems with decentralized control. **ACM Trans. Auton. Adapt. Syst.**, ACM, New York, NY, USA, v. 11, n. 4, p. 25:1–25:35, jan. 2017. ISSN 1556-4665.

BAIER, C.; KATOEN, J.-P. **Principles of Model Checking**. [S.l.]: The MIT Press, 2008. ISBN 026202649X, 9780262026499.

BARESI, L.; QUINTON, C. Dynamically evolving the structural variability of dynamic software product lines. In: **Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems**. Piscataway, NJ, USA: IEEE Press, 2015. (SEAMS '15), p. 57–63.

BARTOCCI, E.; LIó, P. Computational modeling, formal analysis, and tools for systems biology. **PLOS Computational Biology**, Public Library of Science, v. 12, n. 1, p. 1–22, 01 2016.

BASHARI, M.; BAGHERI, E.; DU, W. Dynamic software product line engineering: A reference framework. **International Journal of Software Engineering and Knowledge Engineering**, v. 27, n. 02, p. 191–234, 2017.

BASILI, V.; ROMBACH, H. Goal question metric paradigm. **Encyclopedia of Software Engineering**, v. 2, p. 528–532, 1994.

BENAVIDES, D.; SEGURA, S.; RUIZ-CORTéS, A. Automated analysis of feature models 20 years later: A literature review. **Information Systems**, v. 35, n. 6, p. 615 – 636, 2010. ISSN 0306-4379.

BENCOMO, N.; HALLSTEINSEN, S.; ALMEIDA, E. A view of the dynamic software product line landscape. **Computer**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 45, n. 10, p. 36–41, out. 2012. ISSN 0018-9162.

BENCOMO, N.; HALLSTEINSEN, S.; ALMEIDA, E. Santana de. A view of the dynamic software product line landscape. **Computer**, v. 45, n. 10, p. 36–41, Oct 2012.

BENCOMO, N.; SAWYER, P.; BLAIR, G. S.; GRACE, P. Dynamically adaptive systems are product lines too: Using model-driven techniques to capture dynamic variability of adaptive systems. In: **Second International Workshop DSPL**. [S.l.: s.n.], 2008. p. 23–32.

BENDUHN, F.; THüM, T.; LOCHAU, M.; LEICH, T.; SAAKE, G. A survey on modeling techniques for formal behavioral verification of software product lines. In: **Proceedings of the Ninth International Workshop on Variability Modelling of Software-intensive Systems**. NY, USA: ACM, 2015. (VaMoS '15), p. 80–87. ISBN 978-1-4503-3273-6.

BIERE, A.; CIMATTI, A.; CLARKE, E. M.; ZHU, Y. Symbolic model checking without bdds. In: **Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems**. London, UK, UK: Springer-Verlag, 1999. (TACAS '99), p. 193–207. ISBN 3-540-65703-7.

BOURQUE, P.; FAIRLEY, R. (Ed.). **Guide to the Software Engineering Body of Knowledge, Version 3.0**. [S.l.]: IEEE Computer Society, 2014.

BROWN, P. J.; BOVEY, J. D.; CHEN, X. Context-aware applications: from the laboratory to the marketplace. **IEEE Personal Communications**, v. 4, n. 5, p. 58–64, 1997. ISSN 1070-9916.

BRUGALI, D.; CAPILLA, R.; HINCHEY, M. Dynamic variability meets robotics. **Computer**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 48, n. 12, p. 94–97, dez. 2015. ISSN 0018-9162.

CAFEO, B. B.; NOPPEN, J.; FERRARI, F. C.; CHITCHYAN, R.; RASHID, A. Inferring test results for dynamic software product lines. In: **Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering**. New York, NY, USA: ACM, 2011. (ESEC/FSE '11), p. 500–503. ISBN 978-1-4503-0443-6.

CAPILLA, R.; BOSCH, J.; TRINIDAD, P.; CORTÉS, A. R.; HINCHEY, M. An overview of dynamic software product line architectures and techniques: Observations from research and industry. **Journal of Systems and Software**, v. 91, p. 3–23, 2014.

CAPILLA, R.; ORTIZ, O.; HINCHEY, M. Context variability for context-aware systems. **Computer**, v. 47, n. 2, p. 85–87, Feb 2014. ISSN 0018-9162.

CARVALHO, M. L. L.; GOMES, G. S. D. S.; SILVA, M. L. G. D.; MACHADO, I. D. C.; ALMEIDA, E. S. d. On the implementation of dynamic software product lines: A preliminary study. In: **X Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS)**. [S.l.: s.n.], 2016. p. 21–30.

CHAKRAVARTHY, V.; REGEHR, J.; EIDE, E. Edicts: Implementing features with flexible binding times. In: **Proceedings of the 7th International Conference on Aspect-oriented Software Development**. New York, NY, USA: ACM, 2008. (AOSD '08), p. 108–119. ISBN 978-1-60558-044-9.

CLARKE JR., E. M.; GRUMBERG, O.; PELED, D. A. **Model checking**. Cambridge, USA: MIT Press, 1999. ISBN 0-262-03270-8.

CLASSEN, A.; CORDY, M.; SCHOBBENS, P.-Y.; HEYMANS, P.; LEGAY, A.; RASKIN, J.-F. Featured transition systems: Foundations for verifying variability-intensive systems and their application to ltl model checking. **IEEE Trans. Softw. Eng.**, IEEE Press, Piscataway, NJ, USA, v. 39, n. 8, p. 1069–1089, ago. 2013. ISSN 0098-5589.

CLASSEN, A.; HEYMANS, P.; SCHOBBENS, P.-Y.; LEGAY, A.; RASKIN, J.-F. Model checking lots of systems: Efficient verification of temporal properties in software product lines. In: **Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1**. NY, USA: ACM, 2010. p. 335–344. ISBN 978-1-60558-719-6.

CORDY, M.; CLASSEN, A.; HEYMANS, P.; LEGAY, A.; SCHOBBENS, P.-Y. Assurances for self-adaptive systems: Principles, models, and techniques. In: ____. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. cap. Model Checking Adaptive Software with Featured Transition Systems, p. 1–29. ISBN 978-3-642-36249-1.

CORDY, M.; SCHOBBENS, P.-Y.; HEYMANS, P.; LEGAY, A. Behavioural modelling and verification of real-time software product lines. In: **Proceedings of the 16th International Software Product Line Conference - Volume 1**. NY, USA: ACM, 2012. p. 66–75. ISBN 978-1-4503-1094-9.

CZARNECKI, K.; WASOWSKI, A. Feature diagrams and logics: There and back again. In: **Proceedings of the 11th International Software Product Line Conference**. Washington, DC, USA: IEEE Computer Society, 2007. (SPLC '07), p. 23–34. ISBN 0-7695-2888-0.

DAVIS, F. D. **A technology acceptance model for empirically testing new end-user information systems: theory and results**. Tese (Thesis) — Massachusetts Institute of Technology, 1986.

DEY, A. K. Understanding and using context. **Personal Ubiquitous Computing**, Springer-Verlag, London, UK, v. 5, n. 1, p. 4–7, 2001. ISSN 1617-4909.

DIMOVSKI, A. S.; AL-SIBAHI, A. S.; BRABRAND, C.; WĄSOWSKI, A. Efficient family-based model checking via variability abstractions. **International Journal on Software Tools for Technology Transfer**, p. 1–19, 2016.

DJOUDI, B.; BOUANAKA, C.; ZEGHIB, N. A formal framework for context-aware systems specification and verification. **J. Syst. Softw.**, Elsevier Science Inc., New York, NY, USA, v. 122, n. C, p. 445–462, dez. 2016. ISSN 0164-1212.

DOBSON, S.; STERRITT, R.; NIXON, P.; HINCHEY, M. Fulfilling the vision of autonomic computing. **Computer**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 43, n. 1, p. 35–41, jan. 2010. ISSN 0018-9162.

EKER, S.; MESEGUER, J.; SRIDHARANARAYANAN, A. The maude LTL model checker. **Electronic Notes in Theoretical Computer Science**, v. 71, n. Supplement C, p. 162 – 187, 2004. ISSN 1571-0661. WRLA 2002, Rewriting Logic and Its Applications.

ELBERZHAGER, F.; ROSBACH, A.; MüNCH, J.; ESCHBACH, R. Reducing test effort: A systematic mapping study on existing approaches. **Information and Software Technology**, v. 54, n. 10, p. 1092 – 1106, 2012. ISSN 0950-5849.

ENGSTRöM, E.; RUNESON, P. Software product line testing - a systematic mapping study. **Inf. Softw. Technol.**, Butterworth-Heinemann, Newton, MA, USA, v. 53, n. 1, p. 2–13, jan. 2011. ISSN 0950-5849.

FOGDAL, T.; SCHERREBECK, H.; KUUSELA, J.; BECKER, M.; ZHANG, B. Ten years of product line engineering at danfoss: Lessons learned and way ahead. In: **Proceedings of the 20th International Systems and Software Product Line Conference**. New York, NY, USA: ACM, 2016. (SPLC '16), p. 252–261. ISBN 978-1-4503-4050-2.

GRIEBE, T.; GRUHN, V. A model-based approach to test automation for context-aware mobile applications. In: **Proceedings of the 29th Annual ACM Symposium on Applied Computing**. New York, NY, USA: ACM, 2014. (SAC '14), p. 420–427. ISBN 978-1-4503-2469-4.

GROOTE, J. F.; MATHIJSSEN, A.; RENIERS, M.; USENKO, Y.; WEERDENBURG, M. V. The formal specification language mcrl2. In: **In Proceedings of the Dagstuhl Seminar**. [S.l.]: MIT Press, 2007.

GUEDES, G.; SILVA, C.; SOARES, M.; CASTRO, J. Variability management in dynamic software product lines: A systematic mapping. In: **Proceedings of the 2015 IX Brazilian Symposium on Components, Architectures and Reuse Software**. Washington, DC, USA: IEEE Computer Society, 2015. (SBCARS '15), p. 90–99. ISBN 978-1-4673-9630-1.

HALLSTEINSEN, S.; HINCHEY, M.; PARK, S.; SCHMID, K. Dynamic software product lines. **Computer**, v. 41, n. 4, p. 93–95, April 2008. ISSN 0018-9162.

HäNSEL, J.; GIESE, H. Towards collective online and offline testing for dynamic software product line systems. In: **Proceedings of the 2Nd International Workshop on Variability and Complexity in Software Design**. Piscataway, NJ, USA: IEEE Press, 2017. (VACE '17), p. 9–12. ISBN 978-1-5386-2803-4.

HARMAN, M.; JIA, Y.; ZHANG, Y. Achievements, open problems and challenges for search based software testing. In: **2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)**. [S.l.: s.n.], 2015. p. 1–12. ISSN 2159-4848.

HARMAN, M.; MANSOURI, S. A.; ZHANG, Y. Search-based software engineering: Trends, techniques and applications. **ACM Comput. Surv.**, ACM, New York, NY, USA, v. 45, n. 1, p. 11:1–11:61, dez. 2012. ISSN 0360-0300.

HARTMANN, H.; TREW, T. Using feature diagrams with context variability to model multiple product lines for software supply chains. In: **2008 12th International Software Product Line Conference**. [S.l.: s.n.], 2008. p. 12–21.

HASLINGER, E. N.; LOPEZ-HERREJON, R. E.; EGYED, A. Using feature model knowledge to speed up the generation of covering arrays. In: **Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems**. New York, NY, USA: ACM, 2013. (VaMoS '13), p. 16:1–16:6. ISBN 978-1-4503-1541-8.

HOLLANDER, M.; WOLFE, D. A. **Nonparametric Statistical Methods**. United States: John Wiley & Sons, 1999.

HOLZMANN, G. **Spin Model Checker, the: Primer and Reference Manual**. First. [S.l.]: Addison-Wesley Professional, 2003. ISBN 0-321-22862-6.

IBM. **Predictive analytics software and solutions**. 2017. Available at <http://www-01.ibm.com/software/analytics/spss/>. Last Access in nov. 2017.

IEEE. IEEE Standard Classification for Software Anomalies. **IEEE Std 1044-2009**, January 2010.

IEEE. Ieee standard for system and software verification and validation. **IEEE Std 1012-2012 (Revision of IEEE Std 1012-2004)**, p. 1–223, May 2012.

IEEE. IEEE Draft International Standard for Software and Systems Engineering–Software Testing–Part 4: Test Techniques. **ISO/IEC/IEEE P29119-4-FDIS April 2015**, p. 1–147, April 2015.

JIA, Y.; HARMAN, M. An analysis and survey of the development of mutation testing. **IEEE Trans. Softw. Eng.**, IEEE Press, Piscataway, NJ, USA, v. 37, n. 5, p. 649–678, set. 2011. ISSN 0098-5589.

JOHANSEN, M. F.; HAUGEN, O.; FLEUREY, F. An algorithm for generating t-wise covering arrays from large feature models. In: **Proceedings of the 16th International Software Product Line Conference - Volume 1**. New York, NY, USA: ACM, 2012. (SPLC '12), p. 46–55. ISBN 978-1-4503-1094-9.

KANG, K. C.; COHEN, S. G.; HESS, J. A.; NOVAK, W. E.; PETERSON, A. S. **Feature-Oriented Domain Analysis (FODA) Feasibility Study**. Pittsburgh, PA, 1990.

KISELEV, I. **Aspect-Oriented Programming with AspectJ**. Indianapolis, IN, USA: Sams, 2002. ISBN 0672324105.

KITCHENHAM, B. A.; PFLEEGER, S. L.; PICKARD, L. M.; JONES, P. W.; HOAGLIN, D. C.; EMAM, K. E.; ROSENBERG, J. Preliminary guidelines for empirical research in software engineering. **IEEE Trans. Softw. Eng.**, IEEE Press, Piscataway, NJ, USA, v. 28, n. 8, p. 721–734, ago. 2002. ISSN 0098-5589.

KOSAR, T.; BOHRA, S.; MERNIK, M. Domain-specific languages. **Inf. Softw. Technol.**, Butterworth-Heinemann, Newton, MA, USA, v. 71, n. C, p. 77–91, mar. 2016. ISSN 0950-5849.

KOWAL, M.; SCHULZE, S.; SCHAEFER, I. Towards efficient SPL testing by variant reduction. In: **Proceedings of the 4th International Workshop on Variability & Composition**. New York, NY, USA: ACM, 2013. (VariComp '13), p. 1–6. ISBN 978-1-4503-1867-9.

LAMANCHA, B. P.; POLO, M.; PIATTINI, M. PROW: A pairwise algorithm with constRaints, Order and Weight. **Journal of Systems and Software**, v. 99, p. 1 – 19, 2015. ISSN 0164-1212.

LANDIS, J.; KOCH, G. The measurement of observer agreement for categorical data. Biometrics, v. 33, n. 1, p. 159–174, 1977.

LEMOS, R.; GIESE, H.; MÜLLER, H. A.; SHAW, M.; ANDERSSON, J.; LITOIU, M.; SCHMERL, B.; TAMURA, G.; VILLEGAS, N. M.; VOGEL, T.; WEYNS, D.; BARESI, L.; BECKER, B.; BENCOMO, N.; BRUN, Y.; CUKIC, B.; DESMARAIS, R.; DUSTDAR, S.; ENGELS, G.; GEIHS, K.; GÖSCHKA, K. M.; GORLA, A.; GRASSI, V.; INVERARDI, P.; KARSAI, G.; KRAMER, J.; LOPES, A.; MAGEE, J.; MALEK, S.; MANKOVSKII, S.; MIRANDOLA, R.; MYLOPOULOS, J.; NIERSTRASZ, O.; PEZZÈ, M.; PREHOFER, C.; SCHÄFER, W.; SCHLICHTING, R.; SMITH, D. B.; SOUSA, J. P.; TAHVILDARI, L.; WONG,

K.; WUTTKE, J. Software engineering for self-adaptive systems: A second research roadmap. In: ____. **Software Engineering for Self-Adaptive Systems II: International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. p. 1–32. ISBN 978-3-642-35813-5.

LESTA, U.; SCHAEFER, I.; WINKELMANN, T. Detecting and explaining conflicts in attributed feature models. In: **Proceedings of the Workshop on Formal Methods and Analysis in Software Product Line Engineering (FMSPLE)**. [S.l.: s.n.], 2015. (FMSPLE 2015).

LIMA, E. R. R.; ARAUJO, I. L.; SANTOS, I. S.; OLIVEIRA, T. A.; MONTEIRO, G. S.; COSTA, C. E. B.; SEGUNDO, Z. F. S.; ANDRADE, R. M. C. Great tour: Um guia de visitas móvel e sensível ao contexto. In: **XII Workshop on Tools and Applications. 19th Brazilian Symposium on Multimedia and the Web**. [S.l.: s.n.], 2013.

LIU, Y.; XU, C.; CHEUNG, S. C. AFChecker: Effective model checking for context-aware adaptive applications. **J. Syst. Softw.**, Elsevier Science Inc., New York, NY, USA, v. 86, n. 3, p. 854–867, mar. 2013. ISSN 0164-1212.

LOCHAU, M.; BÜRDEK, J.; HÖLZLE, S.; SCHÜRR, A. Specification and automated validation of staged reconfiguration processes for dynamic software product lines. **Software & Systems Modeling**, p. 1–28, 2015.

LOPEZ-HERREJON, R. E.; CHICANO, F.; FERRER, J.; EGYED, A.; ALBA, E. Multi-objective optimal test suite computation for software product line pairwise testing. In: **Proceedings of the 2013 IEEE International Conference on Software Maintenance**. Washington, DC, USA: IEEE Computer Society, 2013. (ICSM '13), p. 404–407. ISBN 978-0-7695-4981-1.

LOPEZ-HERREJON, R. E.; FERRER, J.; CHICANO, F.; EGYED, A.; ALBA, E. Comparative analysis of classical multi-objective evolutionary algorithms and seeding strategies for pairwise testing of software product lines. In: **2014 IEEE Congress on Evolutionary Computation (CEC)**. [S.l.: s.n.], 2014. p. 387–396. ISSN 1089-778X.

LOPEZ-HERREJON, R. E.; FISCHER, S.; ; RAMLER, R.; EGYED, A. A first systematic mapping study on combinatorial interaction testing for software product lines. In: **Proceedings of the 4th International Workshop on Combinatorial Testing (IWCT 2015), International Conference on Software Testing, Verification and Validation Workshops (ICSTW)**. [S.l.: s.n.], 2015.

MANIKAS, K.; HANSEN, K. M. Software ecosystems - a systematic literature review. **J. Syst. Softw.**, Elsevier Science Inc., New York, NY, USA, v. 86, n. 5, p. 1294–1306, maio 2013. ISSN 0164-1212.

MARINHO, F. G. **PRECISE: Um Processo de veRificação Formal para modElos de CaracterístIcas de Aplicações Móveis e Seníveis ao ContExto**. Tese (Thesis) — Universidade Federal do Ceará, 2012.

MARINHO, F. G.; ANDRADE, R. M. C.; WERNER, C.; VIANA, W.; MAIA, M. E. F.; ROCHA, L. S.; TEIXEIRA, E.; FILHO, J. B. F.; DANTAS, V. L. L.; LIMA, F.; AGUIAR, S. Mobiline: A nested software product line for the domain of mobile and context-aware applications. **Sci. Comput. Program.**, Elsevier North-Holland, Inc., Amsterdam, The Netherlands, The Netherlands, v. 78, n. 12, p. 2381–2398, dez. 2013. ISSN 0167-6423.

MARINHO, F. G.; MAIA, P. H. M.; ANDRADE, R. M. C.; VIDAL, V. M. P.; COSTA, P. A. S.; WERNER, C. Safe adaptation in context-aware feature models. In: **Proceedings of the 4th International Workshop on Feature-Oriented Software Development**. New York, NY, USA: ACM, 2012. (FOSD '12), p. 54–61. ISBN 978-1-4503-1309-4.

MATALONGA, S.; RODRIGUES, F.; TRAVASSOS, G. H. Characterizing testing methods for context-aware software systems: Results from a quasi-systematic literature review. **Journal of Systems and Software**, v. 131, p. 1 – 21, 2017. ISSN 0164-1212.

MAURO, J.; NIEKE, M.; SEIDL, C.; YU, I. C. Context aware reconfiguration in software product lines. In: **Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems**. NY, USA: ACM, 2016. (VaMoS '16), p. 41–48. ISBN 978-1-4503-4019-9.

MENS, K.; CAPILLA, R.; CARDOZO, N.; DUMAS, B. A taxonomy of context-aware software variability approaches. In: **Companion Proceedings of the 15th International Conference on Modularity**. New York, NY, USA: ACM, 2016. (MODULARITY Companion 2016), p. 119–124.

MICSKEI, Z.; SZATMÁRI, Z.; OLÁH, J.; MAJZIK, I. A concept for testing robustness and safety of the context-aware behaviour of autonomous systems. In: **Proceedings of the 6th KES International Conference on Agent and Multi-Agent Systems: Technologies and Applications**. Berlin, Heidelberg: Springer-Verlag, 2012. (KES-AMSTA'12), p. 504–513. ISBN 978-3-642-30946-5.

MOSTEFAOUI, G. K.; PASQUIER-ROCHA, J.; BREZILLON, P. Context-aware computing: a guide for the pervasive computing community. In: **The IEEE/ACS International Conference onPervasive Services, 2004. ICPS 2004. Proceedings.** [S.l.: s.n.], 2004. p. 39–48.

MUCCINI, H.; SHARAF, M.; WEYNS, D. Self-adaptation for cyber-physical systems: A systematic literature review. In: **Proceedings of the 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems**. New York, NY, USA: ACM, 2016. (SEAMS '16), p. 75–81. ISBN 978-1-4503-4187-5.

MUNOZ, F. **Validation of reasoning engines an adaptation mechanisms for self-adaptive systems**. Tese (Thesis) — Université Rennes, 2010.

MURGUZUR, A.; CAPILLA, R.; TRUJILLO, S.; ORTIZ, O.; LOPEZ-HERREJON, R. E. Context variability modeling for runtime configuration of service-based dynamic software product lines. In: **Proceedings of the 18th International Software Product Line Conference: Companion Volume for Workshops, Demonstrations and Tools - Volume 2**. New York, NY, USA: ACM, 2014. (SPLC '14), p. 2–9. ISBN 978-1-4503-2739-8.

MUSCHEVICI, R.; CLARKE, D.; PROENCA, J. Feature petri nets. In: **Proc. Int'l Workshop Formal Methods and Analysis in Software Product Line Engineering**. [S.l.: s.n.], 2010. (FMSPLE '10), p. 99–106.

MUSCHEVICI, R.; PROENÇA, J.; CLARKE, D. Feature nets: behavioural modelling of software product lines. **Software & Systems Modeling**, p. 1–26, 2015. ISSN 1619-1374.

MYERS, G. J.; SANDLER, C.; BADGETT, T. **The Art of Software Testing**. Hoboken, New Jersey: John Wiley & Sons, Inc., 2011.

NETO, P. A. d. M. S.; MACHADO, I. d. C.; MCGREGOR, J. D.; ALMEIDA, E. S. de; MEIRA, S. R. de L. A systematic mapping study of software product lines testing. **Inf. Softw. Technol.**, Butterworth-Heinemann, Newton, MA, USA, v. 53, n. 5, p. 407–423, maio 2011. ISSN 0950-5849.

NORTHROP, L. M. SEI's software product line tenets. **IEEE Softw.**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 19, n. 4, p. 32–40, jul. 2002. ISSN 0740-7459.

NORTHROP, L. M.; CLEMENTS, P. C. **A framework for software product line practice, version 5.0.** [S.l.]: Software Engineering Institute, 2007. Available at <http://www.sei.cmu.edu/productlines/>. Last Access in nov. 2017.

OFFUTT, A. J.; UNTCH, R. H. Mutation testing for the new century. In: WONG, W. E. (Ed.). Norwell, MA, USA: Kluwer Academic Publishers, 2001. cap. Mutation 2000: Uniting the Orthogonal, p. 34–44. ISBN 0-7923-7323-5.

OUAKNINE, J.; WORRELL, J. Some recent results in metric temporal logic. In: **Proceedings of the 6th International Conference on Formal Modeling and Analysis of Timed Systems**. Berlin, Heidelberg: Springer-Verlag, 2008. (FORMATS '08), p. 1–13. ISBN 978-3-540-85777-8.

PUSCHEL, G.; GOTZ, S.; WILKE, C.; PIECHNICK, C.; ABMANN, U. Testing self-adaptive software: Requirement analysis and solution scheme. **International Journal on Advances in Software,**, v. 7, n. 1 & 2, p. 88 – 100, 2014. ISSN 1942-2628.

PüSCHEL, G.; SEIGER, R.; SCHLEGEL, T. Test modeling for context-aware ubiquitous applications with feature petri nets. In: **Modiquitous Workshop**. [S.l.: s.n.], 2012.

QIN, Y.; XU, C.; YU, P.; LU, J. SIT: Sampling-based interactive testing for self-adaptive apps. **Journal of Systems and Software**, v. 120, p. 70 – 88, 2016. ISSN 0164-1212.

RANDOLPH, J. J. **Free-marginal multirater kappa: An alternative to Fleiss fixed-marginal multirater kappa**. [S.l.]: Joensuu University Learning and Instruction Symposium 2005, Joensuu, Finland, 2005.

ROBINSON, J. Likert scale. In: ____. **Encyclopedia of Quality of Life and Well-Being Research**. Dordrecht: Springer Netherlands, 2014. p. 3620–3621. ISBN 978-94-007-0753-5.

ROCHA, L. S.; ANDRADE, R. M. C. Towards a formal model to reason about context-aware exception handling. In: **Proceedings of the 5th International Workshop on Exception Handling**. Piscataway, NJ, USA: IEEE Press, 2012. (WEH '12), p. 27–33. ISBN 978-1-4673-1766-5.

ROCHA, L. S.; ANDRADE, R. M. C. Towards a formal model to reason about context-aware exception handling. In: **Proceedings of the 5th International Workshop on Exception Handling**. Piscataway, NJ, USA: IEEE Press, 2012. (WEH '12), p. 27–33. ISBN 978-1-4673-1766-5.

ROCHA, L. S.; F., J. B. F.; LIMA, F. F. P.; MAIA, M. E. F.; VIANA, W.; CASTRO, M. F. d.; ANDRADE, R. M. C. Ubiquitous software engineering: Achievements, challenges and beyond. In: **2011 25th Brazilian Symposium on Software Engineering**. [S.l.: s.n.], 2011. p. 132–137.

RODRIGUES, F.; MATALONGA, S.; TRAVASSOS, G. H. CATS design: A context-aware test suite design process. In: **Proceedings of the I Brazilian Symposium on Systematic and Automated Software Testing**. [S.l.: s.n.], 2016.

SALLER, K.; LOCHAU, M.; REIMUND, I. Context-aware DSPLs: Model-based runtime adaptation for resource-constrained systems. In: **17th International Software Product Line Conference Co-located Workshops**. New York, NY, USA: ACM, 2013. p. 106–113.

SAMA, M.; ELBAUM, S.; RAIMONDI, F.; ROSENBLUM, D. S.; WANG, Z. Context-aware adaptive applications: Fault patterns and their automated identification. **IEEE Transactions on Software Engineering**, IEEE Computer Society, Los Alamitos, CA, USA, v. 36, n. 5, p. 644–661, 2010. ISSN 0098-5589.

SAMA, M.; ROSENBLUM, D. S.; WANG, Z.; ELBAUM, S. Model-based fault detection in context-aware adaptive applications. In: **Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering**. New York, NY, USA: ACM, 2008. (SIGSOFT '08/FSE-16), p. 261–271. ISBN 978-1-59593-995-1.

SANTOS, I. S.; ANDRADE, R. M.; NETO, P. A. S. Templates for textual use cases of software product lines: results from a systematic mapping study and a controlled experiment. **Journal of Software Engineering Research and Development**, v. 3, n. 1, 2015.

SANTOS, I. S.; ANDRADE, R. M. C.; NETO, P. A. S. Um método para geração otimizada de testes a partir de requisitos para linhas de produto de software dinâmicas. In: **Proceedings of the V Workshop de Teses e Dissertações do CBSoft**. [S.l.: s.n.], 2015.

SANTOS, I. S.; ANDRADE, R. M. C.; ROCHA, L. S.; MATALONGA, S.; OLIVEIRA, K. M. de; TRAVASSOS, G. H. Test case design for context-aware applications: Are we there yet? **Inf. Softw. Technol.**, Butterworth-Heinemann, Newton, MA, USA, v. 88, n. C, p. 1–16, 2017. ISSN 0950-5849.

SANTOS, I. S.; ANDRADE, R. M. C. C.; NETO, P. A. S. How to describe SPL variabilities in textual use cases: A systematic mapping study. In: **Software Components, Architectures and Reuse (SBCARS), 2014 Eighth Brazilian Symposium on**. [S.l.: s.n.], 2014. p. 64–73.

SANTOS, I. S.; ROCHA, L. S.; NETO, P. A. S.; ANDRADE, R. M. C. Model verification of dynamic software product lines. In: **Proceedings of the 30th Brazilian Symposium on Software Engineering**. New York, NY, USA: ACM, 2016. (SBES '16), p. 113–122. ISBN 978-1-4503-4201-8.

SANTOS, I. S.; SOUZA, M. L. J.; CARVALHO, M. L. L.; OLIVEIRA, T. A.; ALMEIDA, E. S.; ANDRADE, R. M. C. Dynamically adaptable software is all about modeling contextual variability and avoiding failures. **IEEE Software**, 2017.

SEI. **SEI Case Studies**. Last Access in Nov. 2017, 2017. Available at <http://www.sei.cmu.edu/productlines/casestudies/index.cfm>. Last Access in nov. 2017.

SHAHID, M.; IBRAHIM, S.; MAHRI, M. N. A study on test coverage in software testing. In: **International Conference on Telecommunication Technology and Applications**. [S.l.: s.n.], 2011.

SHARMA, A. End to end verification and validation with SPIN. **CoRR**, abs/1302.4796, 2013.

SIQUEIRA, B. R.; FERRARI, F. C.; SERIKAWA, M. A.; MENOTTI, R.; CAMARGO, V. V. de. Characterisation of challenges for testing of adaptive systems. In: **Proceedings of the 1st Brazilian Symposium on Systematic and Automated Software Testing**. New York, NY, USA: ACM, 2016. (SAST), p. 11:1–11:10. ISBN 978-1-4503-4766-2.

SPÍNOLA, R. O.; TRAVASSOS, G. H. Towards a framework to characterize ubiquitous software projects. **Inf. Softw. Technol.**, Butterworth-Heinemann, Newton, MA, USA, v. 54, n. 7, p. 759–785, jul. 2012. ISSN 0950-5849.

TRAVASSOS, G. H.; SANTOS, P. S. M. d.; MIAN, P. G.; NETO, A. C. D.; BIOLCHINI, J. An environment to support large scale experimentation in software engineering. In: **Proceedings of the 13th IEEE International Conference on on Engineering of Complex Computer Systems**. Washington, DC, USA: IEEE Computer Society, 2008. (ICECCS '08), p. 193–202. ISBN 978-0-7695-3139-7.

VARSHOSAZ, M.; KHOSRAVI, R. Discrete time markov chain families: Modeling and verification of probabilistic software product lines. In: **Proceedings of the 17th International Software Product Line Conference Co-located Workshops**. NY, USA: ACM, 2013. (SPLC '13 Workshops), p. 34–41. ISBN 978-1-4503-2325-3.

VENKATESH, V.; DAVIS, F. D. A theoretical extension of the technology acceptance model: Four longitudinal field studies. **Manage. Sci.**, INFORMS, Institute for Operations Research and the Management Sciences (INFORMS), Linthicum, Maryland, USA, v. 46, n. 2, p. 186–204, fev. 2000. ISSN 0025-1909.

VERMESAN, O.; FRIESS, P.; GUILLEMIN, P.; GUSMEROLI, S.; SUNDMAEKER, H.; BASSI, A.; JUBERT, I. S.; MAZURA, M.; HARRISON, M.; EISENHAUER, M.; DOODY, P. **Internet of things strategic research roadmap**. [S.l.], 2011. Available at <http://www.internet-of-things-research.eu/pdf/IoT_Cluster_Strategic_Research_Agenda_2011.pdf>. Last Access in jul. 2017.

VIANA, W.; MIRON, A. D.; MOISUC, B.; GENSEL, J.; VILLANOVA-OLIVER, M.; MARTIN, H. Towards the semantic and context-aware management of mobile multimedia. **Multimedia Tools Appl.**, Kluwer Academic Publishers, Hingham, MA, USA, v. 53, n. 2, p. 391–429, jun. 2011. ISSN 1380-7501.

WANG, H.; CHAN, W. K. Weaving context sensitivity into test suite construction. In: **Automated Software Engineering, 2009. ASE '09. 24th IEEE/ACM International Conference on**. [S.l.: s.n.], 2009. p. 610–614. ISSN 1938-4300.

WANG, H.; CHAN, W. K.; TSE, T. H. Improving the effectiveness of testing pervasive software via context diversity. **ACM Trans. Auton. Adapt. Syst.**, ACM, New York, NY, USA, v. 9, n. 2, p. 9:1–9:28, jul. 2014. ISSN 1556-4665.

WANG, Z.; ELBAUM, S.; ROSENBLUM, D. S. Automated generation of context-aware tests. In: **Proceedings of the 29th International Conference on Software Engineering**. Washington, DC, USA: IEEE Computer Society, 2007. (ICSE), p. 406–415. ISBN 0-7695-2828-7.

WEYNS, D.; IFTIKHAR, M. U.; IGLESIA, D. G. de la; AHMAD, T. A survey of formal methods in self-adaptive systems. In: **Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering**. New York, NY, USA: ACM, 2012. (C3S2E '12), p. 67–79. ISBN 978-1-4503-1084-0.

WOHLIN, C.; RUNESON, P.; HöST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLÉN, A. **Experimentation in Software Engineering: An Introduction**. Norwell, MA, USA: Kluwer Academic Publishers, 2000. ISBN 0-7923-8682-5.

WOHLIN, C.; RUNESON, P.; HST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLN, A. **Experimentation in Software Engineering**. [S.l.]: Springer Publishing Company, Incorporated, 2014. ISBN 3642432263, 9783642432262.

XU, C.; CHEUNG, S.; MA, X.; CAO, C.; LV, J. Detecting faults in context-aware adaptation. **International Journal of Software and Informatics**, v. 7, n. 1, p. 85–111, 2013. ISSN 1673-7288.

XU, C.; CHEUNG, S. C.; MA, X.; CAO, C.; LU, J. Dynamic fault detection in context-aware adaptation. In: **Proceedings of the Fourth Asia-Pacific Symposium on Internetware**. New York, NY, USA: ACM, 2012. (Internetware '12), p. 1:1–1:10. ISBN 978-1-4503-1888-4.

YU, L.; TSAI, W. T.; JIANG, Y.; GAO, J. Generating test cases for context-aware applications using bigraphs. In: **Software Security and Reliability (SERE), 2014 Eighth International Conference on**. [S.l.: s.n.], 2014. p. 137–146.

YU, L.; TSAI, W. T.; PERRONE, G. Testing context-aware applications based on bigraphical modeling. **IEEE Transactions on Reliability**, v. 65, n. 3, p. 1584–1611, Sept 2016. ISSN 0018-9529.

ZHANG, W.; ZHAO, H.; MEI, H. Binary-search based verification of feature models. In: **Proceedings of the 12th International Conference on Top Productivity Through Software Reuse**. Berlin, Heidelberg: Springer-Verlag, 2011. (ICSR'11), p. 4–19. ISBN 978-3-642-21346-5.

APPENDIX A – PROMELA BASIC GRAMMAR

This appendix presents the basic grammar of the Promela concerning the main tokens used in the code depicted in Section 4.3.1 of Chapter 4, as well as the codes presented in Appendix B.

Promela is a verification modeling language that consist of variables, processes, and message channels [1] as follows:

- **Variables** are used to store either global information about the system, or information local to a specific process. The basic data types available in Promela are: *bool*, *byte*, *short*, and *int*. Code 2 presents the declaration of two variables.

Code 2 – Example of variables in Promela

```
1        bool flag = true; int state = 2
```

- **Process** are used to specify the system behavior. The behavior of a process is defined in a *proctype* declaration. If the process has the keyword *active* prefixed to the proctype declaration, then the process is active (i.e., running) in the initial system state. Code 3 presents an active process called "processA" that declares a local variable (called *value*) in the initial system state.

Code 3 – Example of process in Promela

```
1       active proctype processA()
2      { byte value;
3
4        value = 3
5      }
```

In the process, it can be used control flow constructs, such as: selection and repetition. The first one is defined by the syntax `if` :: `sequence`[:: `sequence`] ∗ `fi` and specifies statements whose execution depends on a guard condition. The other one is defined by the syntax `do` :: `sequence`[:: `sequence`] ∗ `od` and determines a cyclic process. The stop state of the latter construct is only reachable via a break statement. Code 4 presents an example of these constructs. The *processB* executes option1 or option2 depending on the guard

[1] http://spinroot.com/spin/Man/Manual.html

statement related to the values of “a” and “b”. The process *counter* increments a variable named *count*, and it ends the loop when the value of *count* is equals to 10.

Code 4 – Example of constructs in Promela

```
1   active proctype processB()
2   {
3     if
4       :: atomic {(a != b) -> option1}
5       :: atomic {(a == b) -> option2}
6     fi
7   }
8   active proctype counter()
9   {
10    do
11      :: count++
12      :: (count == 10) -> break
13    od
14  }
```

The statements of the processess can also be enclosed prefixed with the keyword `atomic` (lines 4 and 5 in Code 4). In this case, such statements are executed as one indivisible unit, non-interleaved with other processes.

- **Message channels** are used to model the transfer of data from one process to another. They are specified by the type of message and the number of messages they can store. For instance, the channel “chA” in Code 5 stores up to 3 messages of type short. The statement in line 7 appends the value “12” to the tail of this channel, whereas the next statement (line 8) receives the value from the head of the “chA” and stores it in the variable “msg”. It is worth noting that if the channel size is “0”, then it cannot store message, but only pass single messages from a process to another one.

Code 5 – Example of channel in Promela

```
1   chan chA = [3] of { short };
2   short msg
3
4   active proctype processB()
5   {
6         chA!12;
7         chA?msg
8   }
```

APPENDIX B – PROMELA CODES USED IN THE FEASIBILITY STUDY

Code 6 and Code 7 present the Promela codes corresponding to Mobile Guide and Car DSPL, respectively. They were used in the feasibility study discussed in Section 4.3.3.

Code 6 – Promela code for the Mobile Guides DSPL

```
1
2  int battery = 3;
3  bool hasPwSrc = false, image = false, video = false, mobileGuide=true,
       showDocs=true, text=true;
4  mtype = {imageOn, imageOff, videoOn, videoOff,contextChanged,done,adapted
       }
5  int numAnswer = 0;
6
7  chan buss = [0] of {mtype};
8
9  active proctype ImageActuator() {
10 do
11 :: atomic{ buss?imageOn -> image = true }
12 :: atomic{ buss?imageOff -> image = false }
13 od
14 }
15
16 active proctype VideoActuator() {
17 do
18 :: atomic{ buss?videoOn -> video = true }
19 :: atomic{ buss?videoOff -> video = false }
20 od
21 }
22
23 active proctype Contextmanage() {
24 battery = 3; hasPwSrc = false;
25 buss!contextChanged;buss?adapted
26 do
27 :: (battery == 1 && hasPwSrc == false) -> hasPwSrc = true; buss!
       contextChanged;buss?adapted
28 :: (battery == 1 && hasPwSrc == true) -> battery = battery + 1; buss!
       contextChanged;buss?adapted
29 :: (battery == 2 && hasPwSrc == false) -> battery = battery - 1; buss!
       contextChanged;buss?adapted
30 :: (battery == 2 && hasPwSrc == true) -> battery = battery + 1; buss!
       contextChanged;buss?adapted
31 :: (battery == 3 && hasPwSrc == false) -> battery = battery - 1; buss!
       contextChanged;buss?adapted
32 :: (battery == 3 && hasPwSrc == true) -> hasPwSrc = false; buss!
```

```
    contextChanged;buss?adapted;break
od
}

active proctype Controller() {
do
:: buss?contextChanged ->
run AR01(); run AR02(); run AR03();run AR04();run AR05();
numAnswer = 0;
do
:: (numAnswer != 5) -> buss?done; numAnswer = numAnswer + 1;
:: else -> break
od
buss!adapted
od
}

proctype AR01() {
if
:: (battery == 1 && hasPwSrc == false) -> buss!imageOff;buss!videoOff;
    buss!done
:: else -> buss!done
fi
}

proctype AR02() {
if
:: (battery == 1 && hasPwSrc == true) -> buss!imageOn;buss!videoOff;buss!
    done
:: else -> buss!done
fi
}

proctype AR03() {
if
:: (battery == 2 && hasPwSrc == false) -> buss!imageOn;buss!videoOff;buss
    !done
:: else -> buss!done
fi
}

proctype AR04() {
if
:: (battery == 2 && hasPwSrc == true) ->  buss!imageOn;buss!videoOn;buss!
    done
:: else -> buss!done
fi
```

```
}

proctype AR05() {
if
:: (battery == 3) -> buss!imageOn;buss!videoOn;buss!done
:: else -> buss!done
fi
}


ltl pro01 {[] ((showDocs -> mobileGuide) && (text -> showDocs) && (image -> showDocs) &&
(video -> showDocs) && (mobileGuide -> showDocs) && (showDocs -> text) && (showDocs -> (image || video)) )}
ltl pro21 {<> (battery == 1 && hasPwSrc == false)}
ltl pro22 {<> (battery == 1 && hasPwSrc == true)}
ltl pro23 {<> (battery == 2 && hasPwSrc == false)}
ltl pro24 {<> (battery == 2 && hasPwSrc == true)}
ltl pro25 {<> (battery == 3)}
ltl pro31 {[] ((battery == 1 && hasPwSrc == false) -> <>(image == false && video == false))}
ltl pro32 {[] ((battery == 1 && hasPwSrc == true) -> <>(battery == 1 && hasPwSrc == true && image == true &&
video == false))}
ltl pro33 {[] ((battery == 2 && hasPwSrc == false) -> <>(battery == 2 && hasPwSrc == false && image == true &&
video == false))}
ltl pro34 {[] ((battery == 2 && hasPwSrc == true) -> <>(battery == 2 && hasPwSrc == true && image == true &&
video == true))}
ltl pro35 {[] ((battery == 3) -> <>(battery == 3 && image == true && video == true))}
ltl pro41 {<> image}
ltl pro42 {<> video}
ltl pro51 {<> !image}
ltl pro52 {<> !video}
```

Code 7 – Promela code for the Car DSPL

```
bool car = true, emergencyCall = true, positionningService = true, assistenceSystem = true,
frontDistance_S = true, distance_S=true
bool eraGlonass_R =false, glonass = false, parkAssistence = false, slowFront_DS = false, sideDistance_S = false
bool eCall_E = true, gps = true, adaptiveCC = true, fastFront_DS = true,
```

```
    infotainmentSystem = true
int maxSpeed = 200
int location = 1
int road = 1
int numAnswer = 0


mtype = {eCall_EOn, eCall_EOff, eraGlonass_ROn, eraGlonass_ROff,
    maxSpeed160,maxSpeed100,maxSpeed110,
contextChanged,done,adapted}

chan buss = [0] of {mtype};

active proctype eCall_EActuator() {
do
:: atomic{ buss?eCall_EOn -> eCall_E = true }
:: atomic{ buss?eCall_EOff -> eCall_E = false }
od
}

active proctype eraGlonassActuator() {
do
:: atomic{ buss?eraGlonass_ROn -> eraGlonass_R = true }
:: atomic{ buss?eraGlonass_ROff -> eraGlonass_R = false }
od
}

active proctype maxSpeedActuator() {
do
:: atomic{ buss?maxSpeed160 -> maxSpeed = 160 }
:: atomic{ buss?maxSpeed100 -> maxSpeed = 100 }
:: atomic{ buss?maxSpeed110 -> maxSpeed = 110 }
od
}

active proctype Contextmanage() {
location = 1; road = 1
buss!contextChanged;buss?adapted
do
:: (location == 1 && road == 1) -> road = 2; buss!contextChanged;buss?
    adapted
:: (location == 1 && road == 2) -> road = 3; buss!contextChanged;buss?
    adapted
:: (location == 1 && road == 3) -> location = 2; buss!contextChanged;buss
    ?adapted
:: (location == 2 && road == 3) -> road = 1; buss!contextChanged;buss?
    adapted
```

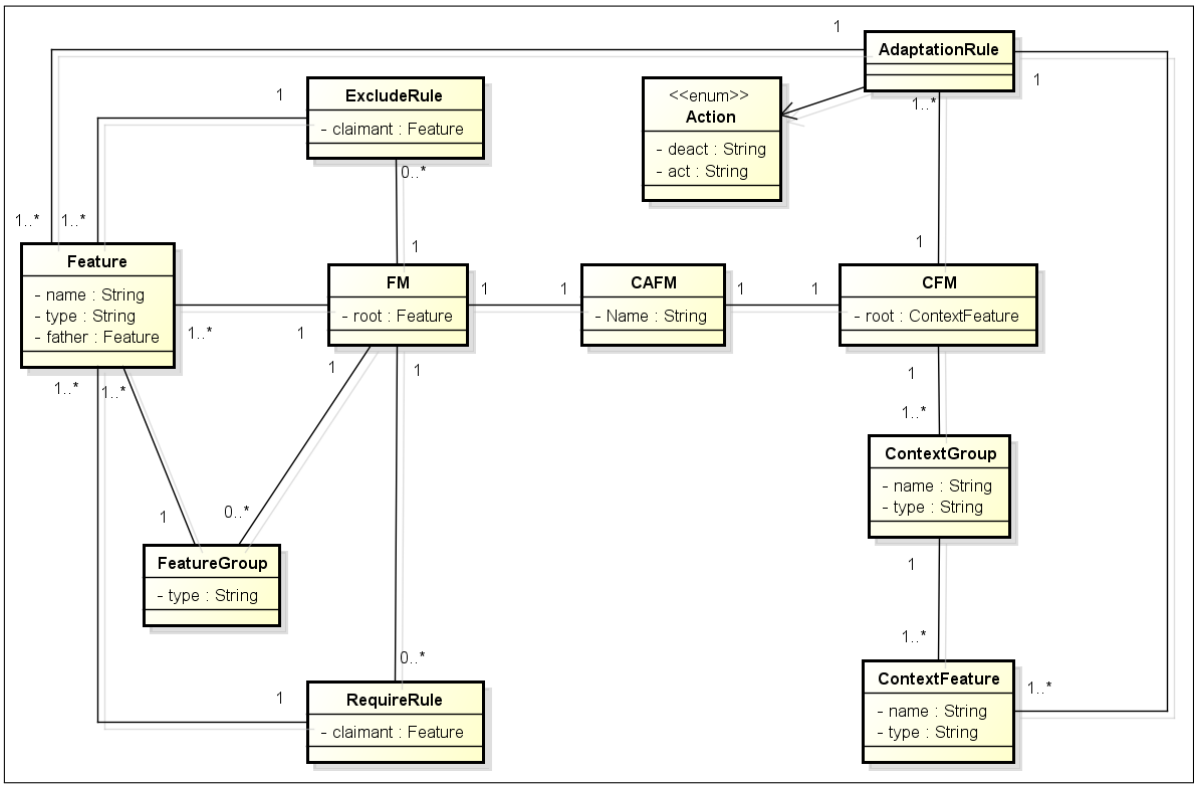
```
46 :: (location == 2 && road == 1) -> road = 2; buss!contextChanged;buss?
       adapted
47 :: (location == 2 && road == 2) -> road = 3; buss!contextChanged;buss?
       adapted;break
48 od
49 }
50
51 active proctype Controller() {
52 do
53 :: buss?contextChanged ->
54 run AR01(); run AR02(); run AR03();run AR04();
55 numAnswer = 0;
56 do
57 :: (numAnswer != 4) -> buss?done; numAnswer = numAnswer + 1;
58 :: else -> break
59 od
60 buss!adapted
61 od
62 }
63
64 proctype AR01() {
65 if
66 :: (location == 1) -> buss!eCall_EOn; buss!done
67 :: else -> buss!done
68 fi
69 }
70
71 proctype AR02() {
72 if
73 :: (location == 2) ->  buss!eraGlonass_ROn;buss!maxSpeed110; buss!done
74 :: else -> buss!done
75 fi
76 }
77
78 proctype AR03() {
79 if
80 :: (road == 2) -> buss!maxSpeed100;buss!done
81 :: else -> buss!done
82 fi
83 }
84
85 proctype AR04() {
86 if
87 :: (road == 3) -> buss!maxSpeed160;buss!done
88 :: else -> buss!done
89 fi
90 }
```

```
91
92
93  ltl pro01 {[] ( (eCall_E -> emergencyCall) && (eraGlonass_R ->
        emergencyCall) && (emergencyCall -> car) &&
94  (gps -> positionningService) && (glonass -> positionningService) && (
        positionningService -> car) &&
95  (parkAssistence -> assistenceSystem) && (adaptiveCC -> assistenceSystem)
        && (assistenceSystem -> car) &&
96  (slowFront_DS -> frontDistance_S) && (fastFront_DS -> frontDistance_S) &&
        (frontDistance_S -> distance_S) &&
97  (sideDistance_S -> distance_S) && (distance_S -> car) && (
        infotainmentSystem -> car) &&
98  (car -> emergencyCall) && (car -> positionningService) && (car ->
        assistenceSystem) &&
99  (assistenceSystem -> adaptiveCC) && (car -> infotainmentSystem) && (
        emergencyCall -> (eCall_E ∧ eraGlonass_R)) &&
100 (positionningService -> (gps ∧ glonass)) && (distance_S -> (
        sideDistance_S || frontDistance_S)) &&
101 (frontDistance_S -> (slowFront_DS ∧ fastFront_DS)) && (eCall_E -> gps) &&
        (eraGlonass_R -> glonass) &&
102 (parkAssistence -> sideDistance_S && frontDistance_S) && (adaptiveCC ->
        frontDistance_S) &&
103 ((maxSpeed > 180) -> fastFront_DS) )}
104 ltl pro21 {<> (location == 1)}
105 ltl pro22 {<> (location == 2)}
106 ltl pro23 {<> (road == 2)}
107 ltl pro24 {<> (road == 3)}
108 ltl pro31 {[] ((location == 1) -> <>(location == 1 && eCall_E == true))}
109 ltl pro32 {[] ((location == 2) -> <>(location == 2 && eraGlonass_R ==
        true))}
110 ltl pro33 {[] ((road == 2) -> <>(road == 2 && maxSpeed == 100))}
111 ltl pro34 {[] ((road == 3) -> <>(road == 3 && maxSpeed == 160))}
112 ltl pro35 {[] ((location == 2) -> <>(location == 2 && maxSpeed == 110))}
113 ltl pro41 {<> eCall_E}
114 ltl pro42 {<> eraGlonass_R}
115 ltl pro43 {<> (maxSpeed == 100)}
116 ltl pro44 {<> (maxSpeed == 160)}
117 ltl pro45 {<> (maxSpeed == 110)}
118 ltl pro51 {<> !eCall_E}
119 ltl pro52 {<> !eraGlonass_R}
120 ltl pro53 {<> (maxSpeed != 100)}
121 ltl pro54 {<> (maxSpeed != 160)}
122 ltl pro55 {<> (maxSpeed != 110)}
```

APPENDIX C – CLASS DIAGRAM FOR THE CONTEXT-AWARE FEATURE MODEL

Figure 37 presents the classes that represent a Context-Aware Feature Model in the TestDAS tool. The TestDAS tool (introduced in Section 4.5) receives as input a JSON file with a feature model following the class diagram depicted in this figure.

Figure 37 – Class diagram for a Context-Aware Feature Model in the TestDAS tool



Source – the author

APPENDIX D – LIST OF GENERATED MUTANTS

Table 29 summarizes the mutants created to assess the model checking approach using the Mutant Analysis.

Table 29 – Summary of mutants

Operator	Mutant ID	Rule affected	Mutation
AltToOR	M1	R2	Image (ON), Video (ON)
AltToOR	M2	R3	Image (ON), Video (ON)
AltToOR	M3	R4	Image (ON), Video (ON)
AltToOR	M4	R5	Image (ON), Video (ON)
AltToAnd	M5	R2	Image (ON), Video (ON)
AltToAnd	M6	R3	Image (ON), Video (ON)
AltToAnd	M7	R4	Image (ON), Video (ON)
AltToAnd	M8	R5	Image (ON), Video (ON)
OrToAl	M9	R4	Image (ON), Video (OFF)
OrToAl	M10	R5	Image (OFF), Video (ON)
OrToAl	M11	R5	Image (ON), Video (OFF)
OrToAnd	M12	R2	Image (ON), Video (ON)
OrToAnd	M13	R3	Image (ON), Video (ON)
OrToAnd	M14	R4	Image (ON), Video (ON)
AndToOr	M15	R1	Image(ON) Video(OFF)
AndToOr	M16	R1	Image(OFF) Video(ON)
AndToAl	M17	R1,R4,R5	Image(OFF) Video(ON)
AndToAl	M18	R1,R4,R5	Image(ON) Video(OFF)
OptToMan	M19	R1	ShowDocuments (OFF), Video (OFF)
OptToMan	M20	R1	Image (OFF), ShowDocuments (OFF)
OptToMan	M21	R2	ShowDocuments (ON), Video (OFF)
OptToMan	M22	R2	Image (ON), ShowDocuments (OFF)
OptToMan	M23	R3	Show Documents (ON), Video (OFF)
OptToMan	M24	R3	Image (ON), Show Documents (OFF)
OptToMan	M25	R4	Show Documents (ON), Video (ON)
OptToMan	M26	R4	Image (ON), Show Documents (ON)

Continued on next page

Table 29 – *Continued from previous page*

Operator	Mutant ID	Rule	Mutation
OptToMan	M27	R5	Show Documents (ON), Video (ON)
OptToMan	M28	R5	Image (ON), Show Documents (ON)
OptToMan	M29	R1	Text (OFF), Video (OFF)
OptToMan	M30	R1	Image (OFF), Text (OFF)
OptToMan	M31	R2	Text (ON), Video (OFF)
OptToMan	M32	R2	Image (ON), Text (OFF)
OptToMan	M33	R3	Text (ON), Video (OFF)
OptToMan	M34	R3	Image (ON), Text (OFF)
OptToMan	M35	R4	Text (ON), Video (ON)
OptToMan	M36	R4	Image (ON), Text (ON)
OptToMan	M37	R5	Text (ON), Video (ON)
OptToMan	M38	R5	Image (ON), Text (ON)
DelF	M39	R1	Image (OFF)
DelF	M40	R1	Video (OFF)
DelF	M41	R2	Image (ON)
DelF	M42	R2	Video (OFF)
DelF	M43	R3	Image (ON)
DelF	M44	R3	Video (OFF)
DelF	M45	R4	Image (ON)
DelF	M46	R4	Video (ON)
DelF	M47	R5	Image (ON)
DelF	M48	R5	Video (ON)
MoveF	M49	R2	Image (ON), Video (OFF), Image (OFF)
MoveF	M50	R3	Image (ON), Video (OFF), Image (OFF)
MoveF	M51	R4	Image (ON), Video (ON), Image(OFF)
MoveF	M52	R5	Image (ON), Video (ON), Image(OFF)
MoveF	M53	R4	Image (ON), Video (ON), Video (OFF)
MoveF	M54	R5	Image (ON), Video (ON), Video (OFF)
MoveF	M55	R1	Image (OFF), Video (OFF), Image(ON)

Continued on next page

Table 29 – *Continued from previous page*

Operator	Mutant ID	Rule	Mutation
MoveF	M56	R1	Image (OFF), Video (OFF), Video (ON)
MoveF	M57	R2	Image (ON), Video (OFF), Video (ON)
MoveF	M58	R3	Image (ON), Video (OFF), Video (ON)
DelRl	M59	R1	remove R1
DelRl	M60	R2	remove R2
DelRl	M61	R3	remove R3
DelRl	M62	R4	remove R4
DelRl	M63	R5	remove R5
ActToDea	M64	R2	Image (OFF), Video (OFF)
ActToDea	M65	R3	Image (OFF), Video (OFF)
ActToDea	M66	R4	Image (OFF), Video (ON)
ActToDea	M67	R4	Image (ON), Video (OFF)
ActToDea	M68	R5	Image (OFF), Video (ON)
ActToDea	M69	R5	Image (ON), Video (OFF)
DeaToAct	M70	R1	Image (ON), Video (OFF)
DeaToAct	M71	R1	Image (OFF), Video (ON)
DeaToAct	M72	R2	Image (ON), Video (ON)
DeaToAct	M73	R3	Image (ON), Video (ON)
AddRl	M74	R1	Image (OFF), Video (OFF), Image (ON)
AddRl	M75	R1	Image (OFF), Video (OFF), Video (ON)
AddRl	M76	R2	Image (ON), Video (OFF), Image (OFF)
AddRl	M77	R2	Image (ON), Video (OFF), Video(ON)
AddRl	M78	R3	Image (ON), Video (OFF), Image (OFF)
AddRl	M79	R3	Image (ON), Video (OFF), Video (ON)
AddRl	M80	R4	Image (ON), Video (ON), Image(OFF)
AddRl	M81	R4	Image (ON), Video (ON), Video (OFF)
AddRl	M82	R5	Image (ON), Video (ON),Image(OFF)
AddRl	M83	R5	Image (ON), Video (ON),Video(OFF)
CtxINV	M84	R1	CtxNotExist

Continued on next page

Table 29 – *Continued from previous page*

Operator	Mutant ID	Rule	Mutation
CtxINV	M85	R2	CtxNotExist
CtxINV	M86	R3	CtxNotExist
CtxINV	M87	R4	CtxNotExist
CtxINV	M88	R5	CtxNotExist
CtxUNR	M89	R1	isBtLow AND !hasPwSrc AND isBtFull
CtxUNR	M90	R2	isBtLow AND hasPwSrc AND isBtFull
CtxUNR	M91	R3	isBtNormal AND !hasPwSrc AND isBtFull
CtxUNR	M92	R4	isBtNormal AND hasPwSrc AND isBtFull
CtxUNR	M93	R5	isBtFull AND isBtLow
CtxINT	M94	R1	isBtLow AND hasPwSrc
CtxINT	M95	R1	isBtNormal AND !hasPwSrc
CtxINT	M96	R1	isBtNormal AND hasPwSrc
CtxINT	M97	R1	isBtFull
CtxINT	M98	R5	isBtLow AND hasPwSrc
CtxINT	M99	R4	isBtLow AND hasPwSrc
CtxINT	M100	R5	isBtNormal AND !hasPwSrc
CtxINT	M101	R4	isBtNormal AND !hasPwSrc
DelRlAll	M102	all	remove all AR
ActToDeaAll	M103	R2,R3,R4 e R5	Image(ON) -> Image (OFF)
ActToDeaAll	M104	R4,R5	Video(ON) -> Video (OFF)
DeaToActALl	M105	R1	Image(OFF) -> Image(ON)
DeaToActALl	M106	R1,R2,R3	Video(OFF) -> Video (ON)
CtxINVAll	M107	all	All to CtxNotExist
CtxUNRAll	M108	all	All to isBtFull AND isBtLow
CtxINTAll	M109	all	R1 -> +Image(ON), others -> +Image(OFF)
CtxINTAll	M110	all	R1,R2,R3 -> +Video(ON),others -> +Video(OFF)

APPENDIX E – EXPERIMENT INSTRUMENTATION

E.1 Background Form

1. Name:
2. Academic Degree
    a) Graduate
    b) Master Student
    c) Master
    d) Doctoral Student
    e) Doctor
3. Which is your experience on Software Testing?
    a) Know what is Software Testing and had already tested software in academy and industry
    b) Know what is Software Testing and had already tested software in academy
    c) Does not have previously knowledge on Software Testing
4. Which is your experience on Model Checking?
    a) Know what is Model Checking and had already used a checker tool
    b) Know what is Model Checking, but had never used a checker tool
    c) Does not have previously knowledge on Model Checking
5. Which is your experience on DSPL?
    a) Know what is DSPL, its related concepts and had already worked with it on academic or industry projects
    b) Know what is DSPL, its related concepts, but had never worked with it
    c) Does not have previously knowledge on DSPL
6. Which is your experience with Java Language?
    a) Have already used Java in industry and academic projects
    b) Have already used Java, but only in academic projects
    c) Does not have previously knowledge on Java
7. Which is your experience with Android?
    a) Have already used Android in industry and academic projects
    b) Have already used Android, but only in academic projects
    c) Does not have previously knowledge on Android

E.2 Task I - Mobiline

The Mobiline DSPL (Figure 38) is a product line of mobile applications that shows to the visitor information about the place s/he is visiting. It has the following features:

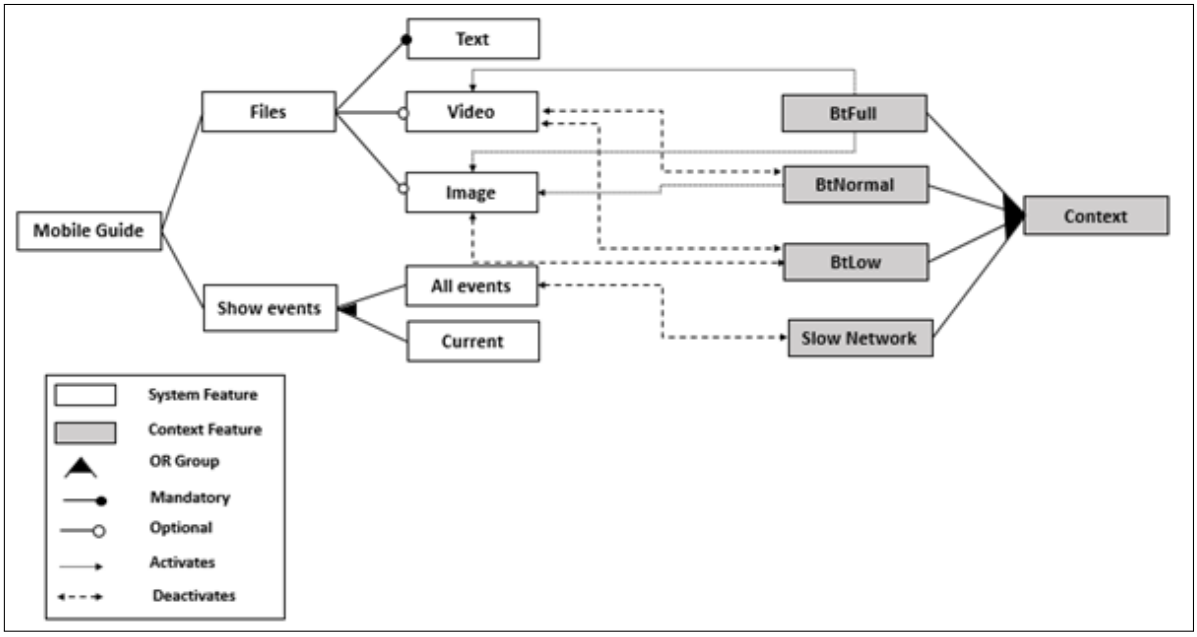
1. **Files**
    a) Text: A functionally that provides text content to the user
    b) Video: A functionally that provides video content to the user
    c) Image: A functionally that provides image content to the user
2. **Show Events**
    a) The products of the Mobile Guide can show events to the visitors. This function can be related to “all events” or “current events”.

Moreover, the Mobiline DSPL uses a set of context information to reconfigure by generating a product at runtime that follows the feature model rules. Regarding the contexts observed, they are described with their context features as follows:

1. **Battery Charge level**: identifies if the battery charge is Full (BtFull), Normal (BtNormal) or Low (BtLow). Usually, the visitor receives a device with full battery and the charge is decreasing over time.
    a) BtFull
    b) BtNormal
    c) BtLow
2. **Network Status**: identifies whether the network is slow or not. This can happen anytime.
    a) Slow Network

Now, you need to play the tester role. Please specify a **Test Sequence** of adaptation test cases, i.e., test cases related to the adaptive behavior. Notice that mandatory features do not change their status (i.e., they are always activated).

Figure 38 – Feature Model of the Mobile Visit Guide for the experiment task



Source – the author

E.3 Task II - Smart Home

The Smart Home DSPL (Figure 39) is a dynamic product line to facilitate the daily lives of people. Some functionality of this DSPL are presented as follows:

1. **Security**
   a) Call the Police: The security system may call the police
   b) Presence Illusion: The system can have a functionality to simulate presence in the house;
   c) Alarm: The security system can have an alarm functionality
2. **Temperature**
   a) The temperature is controlled either by air conditioning or the windows opening.

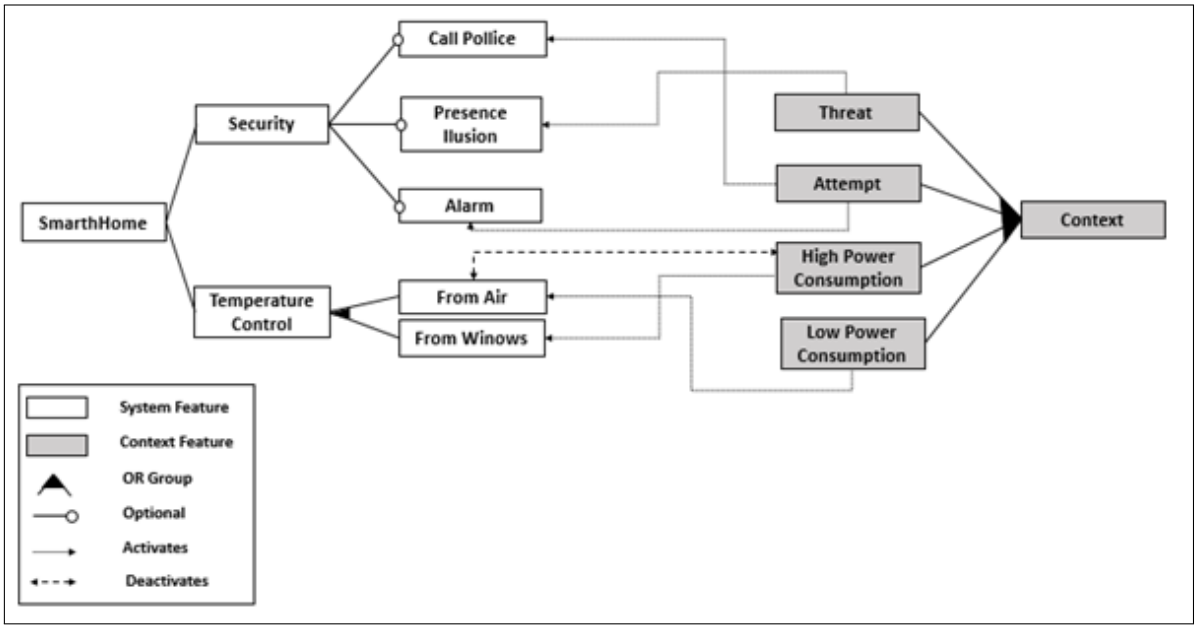
The Smart Home DSPL also uses a set of context information to reconfigure by generating a product at runtime that follows the feature model rules. Regarding the contexts observed, they are described with their context features as follows:

1. **Robbery**: identifies whether exists a threat or (exclusive) an attempt of robbery
   a) Threat
   b) Attempt
2. **Power Consumption**: identifies whether the house has low power consumption or (exclusive) high power consumption. One of these contexts are always true

a) High Power Consumption

b) Low Power Consumption

Figure 39 – Feature Model of the Smart Home for the experiment task



Source – the author

Now, you need to play the tester role. Please specify a **Test Sequence** of adaptation test cases, i.e., test cases related to the adaptive behavior. Notice that mandatory features do not change their status (i.e., they are always activated).

E.4 Pós-Task Form

1. Name:
2. The training was enough to perform the task.
    a) Strongly Disagree
    b) Disagree
    c) Neutral
    d) Agree
    e) Strongly Agree
3. The task goals were clear to me.
    a) Strongly Disagree
    b) Disagree
    c) Neutral

   d) Agree
   e) Strongly Agree
4. I had enough time to perform the task.
   a) Strongly Disagree
   b) Disagree
   c) Neutral
   d) Agree
   e) Strongly Agree
5. The task was easy.
   a) Strongly Disagree
   b) Disagree
   c) Neutral
   d) Agree
   e) Strongly Agree
6. For me, the coverage of the test cases created is enough.
   a) Strongly Disagree
   b) Disagree
   c) Neutral
   d) Agree
   e) Strongly Agree
7. Describe the approach used to create the test cases (i.e, the test coverage criteria used)
8. Do you have any comments about the task?

E.5 Pós-Experiment Form

1. Name:
2. The coverage of the tests generated by TestDAS is higher than the coverage of the tests created manually based on the tester's experience.
   a) Strongly Disagree
   b) Disagree
   c) Neutral
   d) Agree
   e) Strongly Agree

3. The number of tests generated by TestDAS is higher than the number of tests created manually based on the tester's experience.
    a) Strongly Disagree
    b) Disagree
    c) Neutral
    d) Agree
    e) Strongly Agree
4. The time spent to generate the testes with TestDAS is less than the time spent to create tests manually based on the tester's experience.
    a) Strongly Disagree
    b) Disagree
    c) Neutral
    d) Agree
    e) Strongly Agree
5. The proposed test coverage criteria help in the generation of adaptation test cases.
    a) Strongly Disagree
    b) Disagree
    c) Neutral
    d) Agree
    e) Strongly Agree
6. Which is the best way to specify adaptation test cases?
    a) Using the TestDAS and the proposed test criteria
    b) Based on the tester experience
7. Do you have any comments about the TestDAS, training or/and tasks?

APPENDIX F – INSTRUMENTATION OF THE OBSERVATIONAL STUDY

F.1 Feedback Form - TestDAS Tool

1. Name:
2. For me, it was easy to perform the model checking with the TestDAS tool.
   a) Strongly Disagree
   b) Disagree
   c) Neutral
   d) Agree
   e) Strongly Agree
3. I would like to use the TestDAS tool for DAS model checking.
   a) Strongly Disagree
   b) Disagree
   c) Neutral
   d) Agree
   e) Strongly Agree
4. For me, it was easy to generate the adaptation test cases with the TestDAS tool.
   a) Strongly Disagree
   b) Disagree
   c) Neutral
   d) Agree
   e) Strongly Agree
5. I would like to use the TestDAS tool for DAS testing.
   a) Strongly Disagree
   b) Disagree
   c) Neutral
   d) Agree
   e) Strongly Agree
6. Which are the difficulties for the TestDAS tool use?
7. Which are your suggestions for the improvement of the TestDAS tool?

F.2 Feedback Form - CONTroL

1. Name:
2. For me, it was easy to perform the DAS testing using the CONTroL.
    a) Strongly Disagree
    b) Disagree
    c) Neutral
    d) Agree
    e) Strongly Agree
3. I would like to use the CONTroL for DAS testing.
    a) Strongly Disagree
    b) Disagree
    c) Neutral
    d) Agree
    e) Strongly Agree
4. Which are the difficulties for the CONTroL?
5. Which are your suggestions for the improvement of the CONTroL?

APPENDIX G – OTHER PUBLISHED PAPERS DURING THE THESIS WORK PERIOD

1. ARAUJO, I. L.; **SANTOS, I. S.**; FERREIRA FILHO, J. B.; ANDRADE, R. M. C.; SANTOS NETO, P. A. . Generating Test Cases and Procedures from Use Cases in Dynamic Software Product Lines. In: 32nd ACM Symposium on Applied Computing (SAC), 2017, Marrakech, Morocco. Proceedings of 32nd ACM Symposium on Applied Computing, 2017.
2. ANDRADE, R. M. C.; **SANTOS, I. S.**; DANTAS, V. L. L.; OLIVEIRA, K. M.; ROCHA, A. R. C. . Software Testing Process in a Test Factory: From Ad hoc Activities to an Organizational Standard. In: 19th International Conference on Enterprise Information Systems (ICEIS), 2017, Porto, Portugal. Proceedings of International Conference on Enterprise Information Systems, 2017.
3. ANDRADE, R. M. C.; DANTAS, V. L. L.; CASTRO, R. N. S.; **SANTOS, I. S.** . Fifteen Years of Industry and Academia Partnership: Lessons Learned from a Brazilian Research Group. In: 4th International Workshop on Software Engineering Research and Industrial Practice - SER&IP, 2017, Buenos Aires, Argentina. Proceedings of ICSE Workshops, 2017.
4. ARAGAO, B.; **SANTOS, I. S.**; NOGUEIRA, T. P.; MESQUISA, L. B. M.; ANDRADE, R. M. C. . Modelagem Interativa de um Processo de Desenvolvimento com Base na Percepção da Equipe: Um Relato de Experiência. In: Simpósio Brasileiro de Sistemas de Informação, 2017, Lavras, Minas Gerais. Anais do XIII Simpósio Brasileiro de Sistemas de Informação, 2017.
5. ANDRADE, R. M. C.; **SANTOS, I. S.**; ARAUJO, I. L.; ARAGAO, B.; SIEWERDT, F. . Retrospective for the Last 10 years of Teaching Software Engineering in UFC?s Computer Department. In: Brazilian Simposium of Software Engineering, 2017, Fortaleza, Ceará. Proceedings of XXXI Brazilian Simposium of Software Engineering, 2017.
6. CUNHA, T. F. V.; **SANTOS, I. S.**; MACEDO, A. A.; HUGO, A.; ARAGAO, B.; ANDRADE, R. M. C. . CAS 2.0: Evolução e Automação do Checklist de Avaliação do Scrum para Projetos de Software. In: XV Simpósio Brasileiro de Qualidade de Software (SBQS), 2016.
7. SANTOS, R. M.; **SANTOS, I. S.**; MEIRA, R. G.; AGUIAR, P. A.; ANDRADE, R. M. C. . Machine Learning and Location Fingerprinting to Improve UX in a Ubiquitous Application. In: Human-Computer Interaction International (HCII), 2016.

8. SANTOS, R. M.; ANDRADE, R. M. C.; OLIVEIRA, K. M.; **SANTOS, I. S.**; BEZERRA, C. I. M. . Quality characteristics and measures for human computer interaction evaluation in ubiquitous systems. Software Quality Journal (SQJ),2016.
9. **SANTOS, I. S.**; FRANCO, W.; ARAGAO, B.; ANDRADE, R. M. C. . Definição e Aplicação de um Processo de Testes Ágeis: um Relato de Experiência. In: XIV Simpósio Brasileiro de Qualidade de Software, 2015 (SBQS), 2015.
10. ANDRADE, R. M. C.; **SANTOS, I. S.**; SANTOS, R. M.; ARAUJO, I. L. . Uma Metodologia para o Ensino Teórico e Prático da Engenharia de Software. In: VIII Fórum de Educação em Engenharia de Software (FEES), 2015.
11. BEZERRA, C. M.; ANDRADE, R. M. C.; SANTOS, R. M.; ABED, M.; OLIVEIRA, K. M.; MONTEIRO, J. M.; **SANTOS, I. S.**; EZZEDINE, H.. Challenges for usability testing in ubiquitous systems. In: Proceedings of the 26th Conference on l'Interaction Homme-Machine - IHM '14, 2014.
12. **SANTOS, I. S.**; ANDRADE, R. M. C.; SANTOS NETO, P. A. . ChAPTER: Um Método para Geração de Cenários de Testes para Linhas de Produto de Software Sensíveis ao Contexto. In: VI Simpósio Brasileiro de Computação Ubíqua e Pervasiva (SBCUP), 2014.
13. **SANTOS, I. S.**; ANDRADE, R. M. C.; SANTOS NETO, P. A. . Um Ambiente para Geração de Cenários de Testes para Linhas de Produto de Software Sensíveis ao Contexto. In: XIII Simpósio Brasileiro de Qualidade de Software (SBQS), 2014.
14. BEZERRA, C. I. M.; COUTINHO, E. F.; **SANTOS, I. S.**; MONTEIRO FILHO, J. M. S.; ANDRADE, R. M. C. . Evolução do Jogo ItestLearning para o Ensino de Testes de Software: Do Planejamento ao Projeto. In: XIX Conferência Internacional sobre Informática na Educação (TISE), 2014.
15. SOUSA, J.; HUGO, A.; OLIVEIRA, A.; **SANTOS, I. S.**; BRAGA, R.; ANDRADE, R. M. C.; SILVA, F. . SKAM: Um Processo usando Scrum e Kanban para Customização de Software em Dispositivos Móveis. In: X Workshop Anual do MPS (WAMPS), 2014.
16. **SANTOS, I. S.**; BEZERRA, C. I. M.; MONTEIRO, G. S.; ARAUJO, I. L.; OLIVEIRA, T. A.; SANTOS, R. M.; DANTAS, V. L. L.; ANDRADE, R. M. C. . Uma Avaliação de Ferramentas para Testes em Sistemas de Informação Móveis baseada no Método DMADV. In: IX Simpósio Brasileiro de Sistemas de Informação, 2013.
17. SANTOS, R. M.; OLIVEIRA, K. M.; ANDRADE, R. M. C.; **SANTOS, I. S.**; LIMA, E. R. R. . A Quality Model for Human-Computer Interaction Evaluation in Ubiquitous Systems.

In: Latin American Conference on Human Computer Interaction (CLIHC), 2013.

18. LIMA, E. R. R.; ARAUJO, I. L.; **SANTOS, I. S.**; OLIVEIRA, T. A.; MONTEIRO, G. S.; COSTA, C. E. B.; SEGUNDO, Z. F. S.; ANDRADE, R. M. C. . GREat Tour: Um Guia de Visitas Móvel e Sensível ao Contexto. In: XII Workshop on Tools and Applications. 19th Brazilian Symposium on Multimedia and the Web, 2013.
19. **SANTOS, I. S**.; ANDRADE, R. M. C.; SANTOS NETO, P. A. . A Use Case Textual Description for Context Aware SPL Based on a Controlled Experiment. In: CAiSE'13 FORUM, 2013.