Inference of family models for software product line testing

Carlos Diego N. Damasceno

Qualificação de Doutorado do Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional (PPG-CCMC)

Carlos Diego N. Damasceno

Inference of family models for software product line testing

Monograph submitted to the Institute of Mathematics and Computer Sciences – ICMC-USP, as part of the qualifying exam requisites of the of the Doctorate Program in Computer Science and Computational Mathematics.

Concentration Area: Computer Science and Computational Mathematics

Advisor: Prof. Dr. Adenilso da Silva Simão

USP – São Carlos
November 2017

Carlos Diego N. Damasceno

Inferência de modelos de famílias de produtos para teste de linhas de produto de software

Monografia apresentada ao Instituto de Ciências Matemáticas e de Computação – ICMC-USP, para o Exame de Qualificação, como parte dos requisitos para obtenção do título de Doutor em Ciências – Ciências de Computação e Matemática Computacional.

Área de Concentração: Ciências de Computação e Matemática Computacional

Orientador: Prof. Dr. Adenilso da Silva Simão

USP – São Carlos
Novembro de 2017

RESUMO

DAMASCENO, C. D. N. **Inferência de modelos de famílias de produtos para teste de linhas de produto de software**. 2017. 109 p. Monografia (Doutorado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2017.

Engenharia de linha de produto de software (LPS) é um paradigma para produção em massa e customização de famílias de produtos de software compartilhando comonalidades e variabilidades. A fim de possibilitar uma especificação rigorosa, várias notações tem sido propostas para análise e teste de LPS. Notações baseadas em famílias permitem representar todo o comportamento válido que emerge a partir das características (do inglês, *features*) de uma LPS, seja individualmente ou por meio de interações, em um único modelo, denominado modelo de família. Notações baseadas em famílias extendem notações tradicionais com condições de presença expressadas na forma de fórmulas proposicionais para decrever produtos válidos de uma LPS. Recentemente, técnicas de teste baseado em modelos (TBM) especificamente projetadas para modelos de famílias têm sido propostas e permitido a geração de testes a um custo reduzido, quando comparadas ao teste exaustivo de produtos de LPS. Apesar disso, a criação e manutenção de modelos para TBM e LPS ainda são tarefas bastante desafiadoras. Para isso, pesquisadores têm proposto o TBM a partir de modelos inferidos. TBM a partir de modelos inferidos é categorizada em dois grupos: (i) TBM a partir de modelos extraídos, onde modelos comportamentais são automaticamente extraídos para o TBM; e (ii) Teste Baseado em Aprendizagem (TBA), que iterativamente combina os procedimentos de inferência e teste caixa preta. Inferência de modelos pode ser usado para detectar interações entre características, realizar verificação de modelos, e formalmente descrever falhas de software. Além disso, inferência de modelos permite a aplicação de TBM quando modelos de teste estão indisponíveis, desatualizados, ou incompletos. Nesse projeto de pesquisa de doutorado serão investigadas abordagens automáticas de inferência de modelos de famílias de produtos de LPSs. Existe uma lacuna de estudos investigando técnicas de inferência de modelos de famílias de produtos de LPS. Primeiramente, serão investigadas formas de adaptar algoritmos tradicionais de inferência de modelos (e.g., L^*) para inferir modelos de famílias de produtos. Após isso, serão investigadas meios de usar princípios de TBA para detectar e formalmente descrever problemas de interação de características de LPSs. Como resultado, pretende-se fornecer meios de se realizar a inferência e o teste de modelos de famílias de produtos de LPS.

Palavras-chave: Teste Baseado em Modelos, Aprendizado ativo de autômatos Inferência de modelos, Linha de Produto de Software, Teste Baseado em Aprendizagem.

ABSTRACT

DAMASCENO, C. D. N. **Inference of family models for software product line testing**. 2017. 109 p. Monografia (Doutorado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2017.

Software Product Line (SPL) engineering is a paradigm for mass production and customization of families of software products sharing commonalities and variabilities. To provide rigorous specification, several notations have been proposed for analyzing and testing SPLs. Family-based notations allow to represent all valid behaviors that emerge from individual and interacting features of an SPL in a single model, called the family model. Family-based notations often extend traditional notations with presence conditions expressed as propositional formulae to describe the corresponding valid products of an SPL. Recent model-based testing (MBT) techniques specifically tailored to family models have enabled test generation at reduced cost, when compared to product-by-product testing. Although, the creation and maintenance of useful models are still challenging for SPL and MBT. To overcome these shortcomings, researchers have proposed MBT from inferred models. These are categorized into two groups: (i) MBT from ripped models, where behavioral models are automaticaly extracted for MBT; and (ii) Learning-Based Testing (LBT), which iteratively combines inference and testing procedures for black-box testing. Model inference can be used to detect feature interaction, model checking, and formaly describe scenarios under which programs present failures. Moreover, model inference has been useful to MBT when test models are unavailable, outdated, or incomplete. In this Ph.D. research project, we propose to investigate model inference approaches to support the automatic extraction of family models from SPLs. To the best of our knowledge, there are no studies investigating family model inference of SPLs. First, we propose to take model inference algorithms (e.g., Angluin's L^*) as starting point to design family model inference algorithms. Second, we propose to investigate LBT for detecting and formally describing feature interaction in SPLs. As result, we intend to provide guidelines for inferring and testing family models of SPLs.

Keywords: Model-based testing, Active automata learning, Model inference, Software product lines, Learning-based testing.

LIST OF FIGURES

LIST OF ALGORITHMS

LIST OF SOURCE CODES

LIST OF TABLES

CONTENTS

CHAPTER 1 INTRODUCTION

1.1 Context

In its early days, the information technology (IT) industry mostly followed "artisanal", *ad hoc* methods. As time passed by, software has started to be more pervasive and affect nearly every aspect of our everyday activities. Today, the IT industry has evolved from a creator of specialized problem-solving tools to a highly competitive and segmented market which produces indispensable technologies for business, science, and engineering.

Typically, software technologies are developed in a standardized way to simultaneously address a wide range of domains or customer requirements (BENAVIDES; SEGURA; RUIZ-CORTÉS, 2010). As result of this diversity, *variability* has to be anticipated and managed to meet individual needs at reduced cost and time-to-market (SCHAEFER *et al.*, 2012). These tasks are referred to as *mass production* and *mass customization*. To cope with these issues, IT companies have invested in establishing common platforms to efficiently build families of software products named as software product lines (POHL; BÖCKLE; LINDEN, 2005).

Software product line (SPL) is a family of software products sharing a common and managed set of assets that satisfy the specific needs of a particular market segment, and that is developed in a prescribed way (CLEMENTS; NORTHROP, 2001). Unlike traditional systems which are independent and self-contained, SPLs are developed for reuse (i.e., commonalities and variabilities are designed to support the actual development of reusable assets) and with reuse (i.e., products are not created anew, but derived from reused assets) (LINDEN; SCHMID; ROMMES, 2007).

Due to the reuse, the cost and time-to-market to develop and maintain SPLs tend to decrease, while the quality of individual products increases (OSTER *et al.*, 2011a). However, quality assurance of SPLs also becomes more challenging than to single systems as it has to guarantee that reused assets work as intended, regardless combinations (MCGREGOR, 2001).

Model-based testing (MBT) proposes a model-centric approach which relies on explicit test models to support test case selection, generation, and execution (UTTING; PRETSCHNER; LEGEARD, 2012). Finite state machine-based testing is a specialization of MBT which uses finite state machines (FSM) as test models and aims at proving the equivalence (i.e., conformance) between an SUT and its behavioral specification given as an FSM (BROY *et al.*, 2005).

Although FSM-based testing has been studied for several decades (VASILEVSKII, 1973; CHOW, 1978), progresses have still been achieved on test case generation (PETRENKO; BOCHMANN, 1995; SIMÃO; PETRENKO; YEVTUSHENKO, 2009; ENDO; SIMAO, 2013), and on testing access control systems (MASOOD *et al.*, 2009; DAMASCENO; MASIERO; SIMAO, 2016), communication protocols (DOROFEEVA *et al.*, 2010), and satellite software (PINHEIRO; SIMÃO; AMBROSIO, 2014).

In the context of SPLs, theoretically, standard MBT can be applied to all valid products in an exhaustive way. However, it becomes prohibitive due to redundant computations performed over assets shared among different products and the exponential number of valid configurations. To tackle these problems, recent approaches, referred to as *family-based*, have extended MBT for testing families of products in an effective and efficient way (THÜM *et al.*, 2014a).

In family-based testing, test models are referred to as *family* (or *150%*) *models*. Family models specify the behavior of all valid products of SPLs as a unique model annotated with presence conditions specifying what products (i.e., combinations of features) a given part of the family model describe (THÜM *et al.*, 2014a).

Family models have been exploited as a theoretical foundation for more efficient test generation techniques (BEOHAR; VARSHOSAZ; MOUSAVI, 2016), model checking families of products (SABOURI; KHOSRAVI, 2013; ter Beek; VINK; WILLEMSE, 2017), automatically generate specifications of individual products (ASIRELLI *et al.*, 2012), validating families of products (FRAGAL; SIMAO; MOUSAVI, 2017), and specifying fine-grained differences among product variants (SCHAEFER *et al.*, 2010). Despite these possibilities, the creation and maintenance of family models is still challenging (OSTER, 2012).

1.2 Motivation

The creation and maintenance of family models can become difficult and time-consuming, especially to large SPLs, and its traceability to feature models can be complex due to crosscutting features (OSTER, 2012). Added to this, building useful test models by hand is time-consuming and dependent on how familiar the engineers are with the System Under Test (SUT) (BERG; RAFFELT, 2005). To overcome some of these limitations, recent studies (MEINKE; WALKINSHAW, 2012; IRFAN; ORIAT; GROZ, 2013) have proposed to extract test models by using inference algorithms, such as the L^* (ANGLUIN, 1987) and L^*_M (SHAHBAZ; GROZ, 2009).

Model inference, also known as model learning, aims at constructing behavioral models from hardware and software systems by providing inputs (i.e., queries) and observing outputs (VAANDRAGER, 2017). It has been used to enable MBT when test models lack completeness, consistency, or they are outdated (MARIANI; PEZZÈ; ZUDDAS, 2015). Companies such as Siemens (HAGERER *et al.*, 2002), Springer Verlag (NEUBAUER *et al.*, 2012), Volvo Technology (FENG *et al.*, 2013), Orange (SHAHBAZ; GROZ, 2014), and Philips (SCHUTS; HOOMAN; VAANDRAGER, 2016) have applied model inference to solve real problems.

Model inference has been harnessed for black box model checking (PELED; VARDI; YANNAKAKIS, 1999), feature interaction detection in component-based systems (SHAHBAZ; PARREAUX; KLAY, 2007), real protocols (e.g., ABP, SIP, TCP), Biometric Passports, FIFO-Sets, Multi-Login Systems (AARTS *et al.*, 2012; FITERĂU-BROŞTEAN; HOWAR, 2017), service-oriented applications (BAINCZYK *et al.*, 2016), software evolution (HUNGAR; NIESE; STEFFEN, 2003; RUITER; POLL, 2015), analyzing non-resettable systems (GROZ *et al.*, 2015), automatic test generation (RAFFELT *et al.*, 2009), and generalization of failure models (CHAPMAN *et al.*, 2015; KUNZE *et al.*, 2016).

To the best of our knowledge, there are no model inference approaches to support the construction of family models of SLPs. Theoretically, family models could be obtained by using standard model inference in an exhaustive way. However, it becomes prohibitive due to the combinatorial explosion of features and redundant computations (THÜM *et al.*, 2014a). Thus, the model inference has to be extended to SPLs, if possible, or new techniques have to be developed.

1.3 Research Objectives

We define as the primary objective of this Ph.D. research project to **investigate how model inference can be lifted to the family-based level and enable an efficient and effective extraction of family models of SPLs**.

Model inference (VAANDRAGER, 2017) and MBT (BROY *et al.*, 2005) are well-consolidated research topics where much progress has been made, especially in a setting of Mealy machines. However, to date, there are no studies on model inference for extracting family models of SPLs. Thus, we intend to investigate six research questions (RQ).

RQ1: How can we effectively infer Mealy machines from SPLs?

Recently, Fragal, Simao and Mousavi (2017) proposed a specification language named Featured Finite State Machines (FFSM) to enable FSM-based testing of families of products. An FFSM is an extended Mealy machine where states and transitions are annotated with feature constraints to denote presence conditions given a feature model. They proposed a family-based validation criterion for FFSMs and proved that an FFSM model coincides with all FSMs derivable from it.

We opted to initially investigate Mealy machine inference from SPLs because FSM-based testing (ENDO; SIMAO, 2013) and model inference (SHAHBAZ; GROZ, 2009) are two well-consolidated research topics in a setting of Mealy machines. The FFSM model (FRAGAL; SIMAO; MOUSAVI, 2017) will be used as reference to investigate the **RQ1** as well. Solving the **RQ1** is an important part of this Ph.D. research project since all the other RQs depend on it.

RQ2: How can we merge Mealy machines to generate FFSMs?

Family models are often specified monolithically (i.e., built in its entirety). To allow modularity, Atlee *et al.* (2015) proposed a language of Feature Merge Expressions (FME) for embedding new feature-specific behavior (i.e., addition of transitions and states and manipulation of feature expressions) into existing family models denoted as featured transition systems (FTS) (CLASSEN *et al.*, 2013).

FMEs support the resolution of feature interactions (i.e., when one feature merged after another or vice-versa is non-commutative) and the specification of more generic FMEs – Extended Feature Merge Expressions (EFME), that may apply to multiple parts of an FTS (ATLEE *et al.*, 2015). We aim at investigating how FMEs can support the inclusion of new behavior into existing FFSMs.

Along with a family model inference procedure, family models will be enriched with new traces (i.e., outputs) obtained from queries asked for products of the same SPL. Thus, states equivalent to each other will have to be merged for representing a broader range of the possible behavior of the family of products it belongs to. State merging algorithms have been used for this purpose (WALKINSHAW, 2013) and we also intend to study how these algorithms can support family model inference.

RQ3: How can we efficiently infer FFSMs from products of SPLs?

Exhaustive inference of SPLs is prohibitive due to the exponential number of products and redundant computations (THÜM *et al.*, 2014a). To this end, t-wise coverage can help to reduce the number of product configurations to test and detect interactions between any $t > 2$ features. However, t-wise coverage faces scalability issues (GRINDAL; OFFUTT; ANDLER, 2005; NIE; LEUNG, 2011).

Henard *et al.* (2014) showed that similarity heuristics as a viable alternative for t-wise coverage and proposed *configuration prioritization*, an approach for ordering a set of product configurations to reach faster a t-wise coverage. We also intend to investigate how configuration prioritization can make family model inference more efficient.

Sharygina *et al.* (2005) and Chaki *et al.* (2008) investigated approaches to support the verification of evolving systems. They proposed a dynamic procedure that infers models of evolvable components by reusing the knowledge from older versions to prove or disprove global properties of updated systems. Filters to avoid unnecessary queries (MARGARIA; RAFFELT; STEFFEN, 2005) could also be designed based on the FFSM theory. Since the behavior of a product is more (or less) than the sum of the behaviors of the individual features involved (APEL *et al.*, 2013c), we want to investigate if the knowledge learned from products of one same SPL can be reused to reduce the number of queries for inferring family models.

After investigating efficient and effective ways to infer Mealy machines from SPLs, we also propose to leverage family model inference to the domains of learning-based testing (LBT), detection of feature interaction problems, and EFSMs. Thus, the following RQs will also be investigated:

RQ4 How can we take advantage of LBT for testing SPLs?

The completeness of the test models significantly affects the quality of MBT (UTTING; PRETSCHNER; LEGEARD, 2012). At the same time, the cost of inferring and testing complete models of real systems may be prohibitive, if time and resources are limited. To overcome this issue, Meinke and Sindhu (2011) proposed LBT, an iterative approach that aims at automatically generating a large number of test cases by combining a model checking algorithm with optimized model inference algorithms.

In LBT, model checking tools and model inference algorithms are iteratively used to either reveal faults by applying *tests as queries* or augment the hypothesized model (MEINKE; SINDHU, 2013). Model checking tools such as `NuSMV` (CLASSEN *et al.*, 2011), `Rebeca` (SABOURI; KHOSRAVI, 2013) and `mcrl2` (ter Beek; VINK; WILLEMSE, 2017) have been successfully adapted for model checking families of products. Thus, we believe that LBT can be leveraged to SPLs and reduce the need for inferring complete family models for testing families of products of SPLs.

RQ5 Can we use family model inference to detect feature interaction problems?

A feature interaction occurs when one feature is influenced by the presence of a set of other features (APEL *et al.*, 2013c). These interactions can result in inadvertent deviations from the expected behavior of a product.

Chapman *et al.* (2015) proposed an inference procedure to describe assertion violations using code instrumentation. Kunze *et al.* (2016) designed an approach to build generalized failure models. These models describe under what other circumstances (e.g., interaction scenarios, different parameters) similar failures can be observed given a failing test case. We aim at using these studies as reference to design novel feature interaction detection techniques capable of inferring and describing scenarios where behavioral requirements are violated.

RQ6 How can we perform family model inference in a setting of EFSMs?

Richer behavioral models, such as EFSMs (PETRENKO; BORODAY; GROZ, 2004), also have been investigated in the context of SPLs (WEIẞLEDER; SOKENOU; SCHLINGLOFF, 2008; GONZALEZ; LUNA, 2008; OSTER *et al.*, 2011b), model inference (CASSEL *et al.*, 2016; WALKINSHAW; TAYLOR; DERRICK, 2016), and MBT (EL-FAKIH *et al.*, 2017), but as isolated problems. Thus, there are no studies that investigate efficient and effective ways to extract EFSMs from SPLs. At last, we also intend to investigate the problem of *inferring EFSMs from SPLs*.

This Ph.D. research project will be developed in collaboration with Mohammad Reza Mousavi and Neil Walkinshaw, both professors at University of Leicester, United Kingdom (UK). From a theoretical and practical perspectives, they have extensive experience with software testing, MBT, SPLs, embedded systems, and model learning.

Professor Mousavi has prior successful collaboration with Adenilso Simão, the advisor of this project, and two of his former Ph.D. students (FRAGAL; SIMAO; MOUSAVI, 2017; PAIVA *et al.*, 2016). He also has agreed to host the Ph.D. student for an extended period during his project. With this coolaboration, we believe that we will be able to take this Ph.D. research project to more realistic scenarios.

1.4 Organization

This Ph.D. research project is organized as follows:

Chapter 2 We introduce the basic definitions of software testing, and the three primary testing criteria (i.e., functional testing, structural testing, and mutation testing). After, we present the concept of MBT and the specificities of FSM-based testing. We introduce Mealy machines and how they can be used for test case generation, and coverage analysis. The W, $\mathrm{W_p}$, HSI and SPY methods are presented. We also introduce the test prioritization problem. Three examples of software testing tools are introduced.

Chapter 3 We present the concept of model inference, the minimally adequate teacher (MAT) framework, and the L_M^* algorithm for inferring Mealy machines. We discuss some possible optimizations that can reduce the cost of inferring models. The process of MBT from ripped models, LBT, detection of assertion violations and generalization of failure models are presented as examples of problems where model inference can be applied. Five examples of tools to support model inference are introduced.

Chapter 4 We define SPL, the SPLE framework and feature modeling. Strategies for specifying and analyzing SPLs are introduced. The two extremes of specification notations (i.e., product-based and family-based) are presented with examples. Two examples of tools for developing SPLs are presented.

Chapter 5 We detail our research proposal, discuss the primary and secondary objectives of this work, the research method, and analysis of results. The work plan and expected results are also presented in this part.

Part I

Background and Related Work

CHAPTER

2

SOFTWARE TESTING

Human beings have no capacity to perform and communicate with perfection. Thus, many circumstances to inject software defects may emerge along life cycle (DEUTSCH, 1981). To mitigate this issue, there is a set of activities to build quality into software systems collectively named verification and validation (IEEE, 2012).

Verification and validation (V&V) concern whether software products conform to their requirements and satisfy the intended use and user needs (IEEE, 2010b). V&V activities are often classified as *static analysis*, whether a system or component is evaluated based on its form, structure, content, or documentation; and *dynamic analysis*, whether the evaluation is based on its behavior during execution.

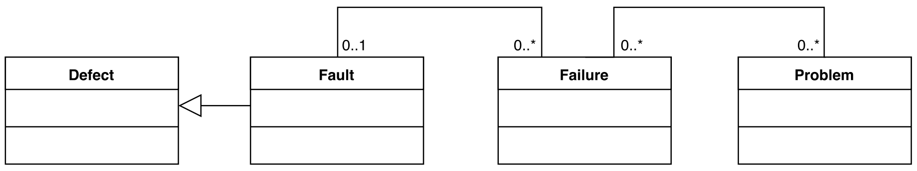
Static analysis is a testing approach which relies on the examination of software artifacts without execution to detect problems or violations of development standards (MYERS; SANDLER; BADGETT, 2012). Two prominent approaches are *code inspections* and *walkthroughs*. Code inspection aims at detecting defects by group code reading and using checklists of historically common problems. In walkthroughs, test cases are mentally executed while program states (i.e., values of variables) are monitored on paper or whiteboard, and programmers' logic and assumptions are questioned. Dynamic analysis addresses V&V by executing programs with a set of inputs and observing the resulting behavior (AMMANN; OFFUTT, 2008). Software testing is one of the primary dynamic analysis approaches.

In this chapter, we introduce the basic definitions of software testing and the functional, structural, and mutation testing criteria. We also present model-based testing (MBT), a variant of software testing that relies on explicit test models to automate test generation (UTTING; PRETSCHNER; LEGEARD, 2012) and some examples of tools for MBT.

2.1 Basic definitions

According to IEEE (2010a), a *defect* is an imperfection or deficiency in a software product that causes it to not meet its requirements, and that needs to be either repaired or replaced. Defects can be found using static and dynamic analysis. In the latter, a defect is also called a *fault*. When a fault is encountered, it may produce a *failure*. A failure is an event in which a software system or component does not perform a required function within previously specified limits. A failure may cause one or more *problems* that may or may not be readily experienced by one or more users (e.g., customers, testers). Figure 1 depicts the relationships between these anomalies.

Figure 1 – Classification for software anomalies



Source: Adapted from IEEE (2010a).

Software testing is the activity in which an SUT is executed under specified conditions, the results are observed or recorded, and an evaluation of some aspects of the SUT is made (IEEE, 2010b). A test case specifies the *test inputs*, *execution conditions*, *expected outputs*, and its *execution order* (IEEE, 2010b). A set of test cases is called *test suite*, and the evaluation of obtained outputs is performed using a test oracle (JORGENSEN, 2013).

A *test oracle* is an instrument that determines whether a given obtained output is an acceptable behavior of the SUT or not (BARR *et al.*, 2015). Test oracles are classified into four categories: (i) specified oracles, (ii) derived oracles, (ii) implicit oracles, and (ii) human oracles. Table 1 describes each of these categories of test oracles with examples.

Table 1 – Categories of test oracles

Test oracle	Relies on	Examples
Specified	Formal specifications to judge whether the SUT has an acceptable behaviour	Mealy machines (BROY *et al.*, 2005), assertions (MASSOL; HUSTED, 2003), and contracts (BURDY *et al.*, 2005)
Derived	Information derived from documentation artifacts, system executions, properties, or other versions of the SUT	Pseudo-oracles (WEYUKER, 1982), regression test suites (ELBAUM *et al.*, 2004), invariant detection (ERNST *et al.*, 2007), and model inference (ISBERNER; HOWAR; STEFFEN, 2015)
Implicit	General and implicit knowledge (e.g., buffer overflows, segfaults, exceptions)	Fuzz testing (BEKRAR *et al.*, 2011), and model inference (RUITER; POLL, 2015)
Human	Reducing human involvement in writing oracles and evaluating test outputs	Test suite minimization (YOO; HARMAN, 2012), and meta-heuristics (AFSHAN; MCMINN; STEVENSON, 2013)

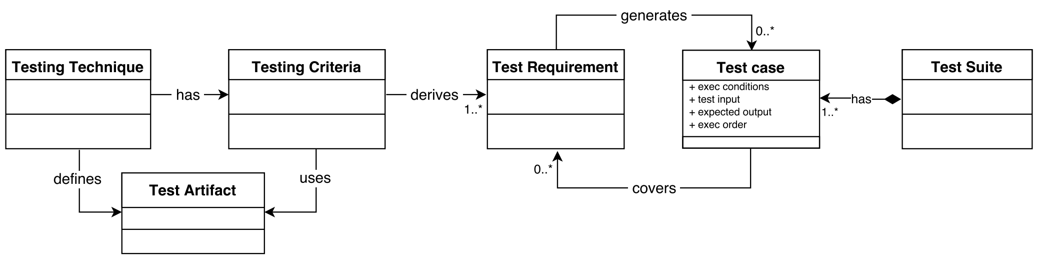
Source: Adapted from Barr *et al.* (2015).

Exhaustive testing executes the SUT using all possible test inputs from its input domain. Since input domains are often infinite or significantly large, exhaustive testing is not feasible in practice. Moreover, determining whether an SUT is correct or not is, in general, impossible due to theoretical limits. Thus, testing criteria are used to systematize the software testing activity.

2.2 Testing Criteria

Testing criteria define what specific elements of a *test artifact* (i.e., *test requirements*) a test case has to exercise (AMMANN; OFFUTT, 2008) to constitute a "*thorough*" test suite (GOODENOUGH; GERHART, 1975). They can support test case generation and evaluation using different kinds of artifacts (e.g., source code, specification models) (MACHADO; VINCENZI; MALDONADO, 2010). Figure 2 depicts the relationship between these concepts.

Figure 2 – Relationships between software testing concepts



Source: Adapted from Machado, Vincenzi and Maldonado (2010).

Given an SUT P, a test suite T, and a testing criterion C, the test suite T is C-adequate for P if it satisfies all testing requirements of C. The coverage level describes the effectiveness of T as the ratio of the number of test requirements satisfied by T to the total number of test requirements (i.e., $\text{eff}(T,P,C) = \frac{\text{req satisfied}}{\text{total of req}}$). It is important to highlight that some test requirements may be impossible to be satisfied (i.e., *infeasible*).

The three most prominent software testing techniques are presented in the next sections. These are (i) functional testing, (ii) structural testing, and (iii) mutation testing. Examples are discussed using the `validateIdentifier` program (VINCENZI *et al.*, 2010) as SUT. Then, we introduce the MBT technique and the specificities of FSM-based testing.

The validate identifier problem

Problem statement: The `validateIdentifier` program receives a string as input parameter and evaluates if it is a valid variable name for a simplified programming language. To be a valid identifier, the string must satisfy the following conditions:

1. It has at least one and no more than six characters.
2. It must begin with a letter (i.e., `a..z`, `A..Z`);
3. It must contain only letters or digits (i.e., `a..z`, `A..Z`, `0..9`);

The output of the program is a boolean value denoting `true` if all conditions are satisfied; otherwise, it returns `false`. Source code 1 shows a Java version of the validate identifier program.

Source code 1 – The validateIdentifier program

```java
public boolean validateIdentifier(String s) {
  char a_char;
  boolean valid_id = false;      /* 01 */
  if (s.length()>0) {            /* 01 */
    a_char = s.charAt(0) ;            /* 02 */
    valid_id = valid_s ( a_char );    /* 02 */
    if (s.length() > 1){              /* 02 */
        a_char = s.charAt(1) ;   /* 03 */
        int i = 1;               /* 03 */
        while ( i < s.length() - 1 ) {  /* 04 */
          a_char = s.charAt(i) ;               /* 05 */
          if ( ! valid_f( a_char ) ){          /* 05 */
            valid_id = false ;     /* 06 */
          }
          i++;   /* 07 */
        }
    }
  }
  if (      valid_id                   /* 08 */
          && ( s.length() >= 1 )        /* 09 */
          && ( s.length() < 6 ) )           /* 10 */
    return true ;   /* 11 */
  else  return false; /* 12 */
}
public boolean valid_s( char_ch ){
  if( ((ch>='A') && (ch<='Z'))
      || ((ch>='a') && (ch<='z')))
    return true;
  else return false;
}
public boolean valid_f(char_ch){
  if( ((ch>='A') && (ch<='Z'))
          || ((ch>='a') && (ch<='z'))
          || ((ch>='0') && (ch<='9')) )
    return true;
  else  return false;
}
```

Source: Adapted from Vincenzi *et al.* (2010).

2.2.1 Functional Testing

Functional testing is a testing technique which is unconcerned about the internal behavior and structure (i.e., source code) of the SUT (MYERS; SANDLER; BADGETT, 2012). In this approach, the SUT is considered to be a function mapping values between input and output domains. It aims at deriving test cases only from external descriptions (i.e., functionalities) of the SUT, such as requirements, specifications, and design models (AMMANN; OFFUTT, 2008). Other names frequently used for functional testing are *specification-based*, *black-box*, *data-driven*, and *input/output-driven* testing.

Functional testing has two main advantages: (i) test cases are independent of implementation specificities, and (ii) test case design can be done in parallel with the implementation of the SUT. However, as a disadvantage, test cases may leave some parts of the source code untested. The techniques *equivalence class testing* and *boundary value analysis* are discussed below.

2.2.1.1 Equivalence Class Testing

Equivalence class testing assumes that if a certain test case of a given partition (i.e., class) of an input domain can detect a fault, every other test of this same partition is also able to detect the same fault (VINCENZI *et al.*, 2010). To this criterion, the input domain of the SUT is divided based on its input conditions (i.e., input parameters) and classes of *valid* and *invalid* data (MYERS; SANDLER; BADGETT, 2012). Thus, more systematic analysis and size restrictions can be applied to test cases designed. Table 2 shows an example of equivalence classes to test the `validateIdentifier` program.

Table 2 – Equivalence classes of the `validateIdentifier` program

Input conditions	Valid Classes	Invalid Classes	
Identifier's size t	$1 \leq t \leq 6$ (1)	$t < 1$ (2)	$t > 6$ (3)
First character is a letter	Yes (4)	No (5)	
Only contains valid characters	Yes (6)	No (7)	

Source: Vincenzi *et al.* (2010).

Example: Given Table 2, test case `("id123",valid)` covers classes (1), (4), and (6) and has 42.8% of effectiveness. Using the four following test cases, classes (2), (3), (5), and (7) are respectively covered: `("",invalid)`, `("identifierTooLongToBeValid",invalid)`, `("3rror",invalid)`, and `("#fail",invalid)`.

Equivalence class testing can also be used to partition output domains. Other names used to refer to equivalence class testing are *input space partitioning* (AMMANN; OFFUTT, 2008), *equivalence partitioning* (MYERS; SANDLER; BADGETT, 2012), and *category-partition method* (OSTRAND; BALCER, 1988).

2.2.1.2 Boundary Value Analysis

The boundary value analysis is a complementary technique to the equivalence class testing. Values at and close to boundaries often have faults; thus, they can increase the probability of exposing faults. Essentially, test cases are designed based on inputs taken from the boundaries of each equivalence class (MYERS; SANDLER; BADGETT, 2012). Below, we show an example applying the boundary value analysis applied to the `validateIdentifier` program.

Example: In the `validateIdentifier` program, the input domain consists of a string that is evaluated "*character-by-character*". If non-alphanumeric characters are found, or the identifier has more than six or less than one characters, the identifier is invalid. In this case, the input domain can be described based on the *ASCII table*. Table 3 shows a fragment of the ASCII table.

Table 3 – Boundary value analysis of the `validateIdentifier` program

Description	Space	Slash	Numbers				Colon	At sign	Uppercase letters				Square brackets	Grave accent	Lowercase letters				Curly braces
ASCII code	32	**47**	48	**49**	**56**	57	**58**	**64**	65	**66**	**89**	90	**91**	**96**	97	**98**	**121**	122	**123**
Symbol		**/ (8)**	0	**1 (9)**	**8 (10)**	9	**: (11)**	**@ (12)**	A	**B (13)**	**Y (14)**	Z	**[(16)**	**‘ (17)**	a	**b (18)**	**y (19)**	z	**{ (20)**

Source: Elaborated by the author.

The first and the last valid characters of each category (i.e., `'a'`, `'z'`, `'A'`, `'Z'`, `'0'`, `'9'`) are the boundaries of the input domain of the `validateIdentifier` program. Thus, the "neighbors" of these valid characters in the ASCII table are the candidates to be used as test inputs. These characters close to the boundaries are highlighted in **bold** and enumerated from 8 to 20. The test cases designed to satisfy the test requirements of the boundary value analysis technique are shown in Table 4.

Table 4 – Test cases generated using boundary value analysis

Test case	Test requirements covered
`("v/a",invalid)`	(8)
`("va18",valid)`	(9), (10)
`("va:",invalid)`	(11)
`("v@",invalid)`	(12)
`("BYa",valid)`	(13), (14)
`("va[",invalid)`	(16)
`("v‘a",invalid)`	(17)
`("bya",valid)`	(18), (19)
`("v{a",invalid)`	(20)

Source: Elaborated by the author.

As the equivalence class testing, boundary value analysis can also be applied to the output domain of SUTs.

2.2.2 Structural Testing

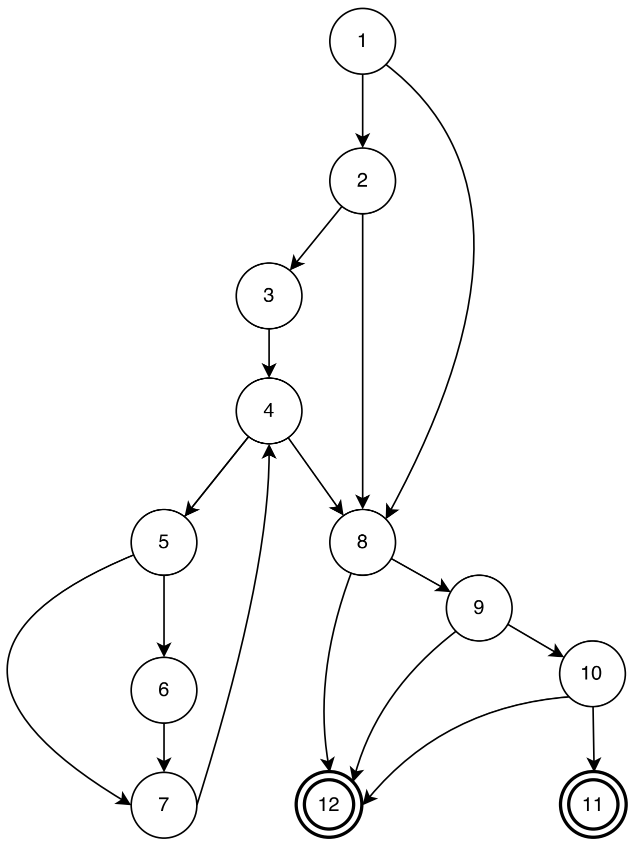
Structural testing is a complementary technique to functional testing that can be applied to code-based and model-based testing. In code-based testing, a *program graph* is derived from the source code, and graph theory concepts are used to evaluate test coverage (JORGENSEN, 2013). Program graphs can include control flow and data flow information of an SUT.

As the main advantage, structural testing enables a more detailed analysis of how much and what parts of the source code of an SUT have been exercised (i.e., covered). However, as a disadvantage, structural coverage may sometimes be never complete due to infeasible test requirements (e.g., infeasible paths and data-flow associations). The *control flow criteria* and *data flow criteria* are discussed below.

2.2.2.1 Control Flow Criteria

The control flow criteria use CFGs to describe program flow (ALLEN, 1970). A set of ordered statements of a program is named block. Blocks are represented as nodes and edges define the sequence these statements/blocks are executed. The result of a test case is a test path which defines what nodes and edges of the CFG are covered. Figure 3 depicts the CFG of the `validateIdentifier` program.

Figure 3 – Control-flow graph of the `validateIdentifier` program



Source: Adapted from Vincenzi *et al.* (2010).

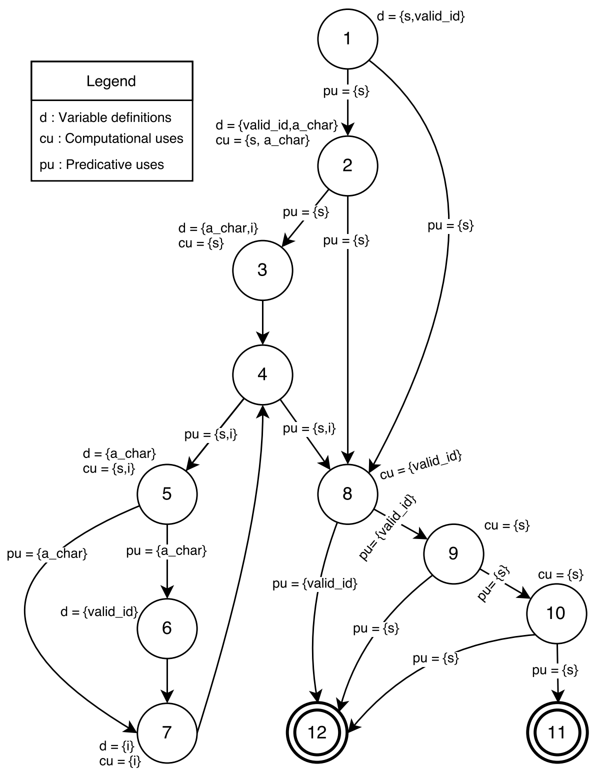
The three main control flow criteria are *All-Nodes*, which requires that for all reachable node n of a CFG, there is a test case t such that t visits n; *All-Edges*, which requires that for all reachable edge e of a CFG, there is a test case t such that t traverses e; and *All-Paths*, which requires that for all path p of a CFG, there is a test case t such that t covers p (MYERS; SANDLER; BADGETT, 2012; AMMANN; OFFUTT, 2008).

Example: The test cases `("v/a",invalid)` and `("i",valid)` respectively cover the paths $\{1,2,3,4,5,7,4,5,6,7,4,5,7,4,8,12\}$ and $\{1,2,3,4,5,7,4,8,9,10,11\}$, satisfying the *All-nodes* criteria. By adding the test cases `("",invalid)` and `("v1234567",invalid)`, the *All-edges* criteria also become satisfied. In the *All-paths* criteria, there are some infeasible requirements (e.g., $\{1,8,9,10,11\}$). Moreover, the loop of the `validateIdentifier` program makes *All-paths* impractical as it requires an infinite number of paths to be executed.

2.2.2.2 Data Flow Criteria

The data flow testing criteria use an extended CFG that includes the *definitions* and *uses* of variables along an SUT (RAPPS; WEYUKER, 1985). It was proposed to tackle the infinite number of paths that CFGs with loops may have. The graph used in this criteria is named data flow graph (DFG) or def-use graph (du-graph). Figure 4 depicts the data flow graph of the `validateIdentifier` program.

Figure 4 – Du-graph of the `validateIdentifier` program



Source: Adapted from Vincenzi *et al.* (2010).

In a definition (*def*), a variable has a value stored in memory (e.g., assignment, input). In a *use*, a variable's value is accessed to compute other values, these are named computational use (*c-use*), or in conditions, these are named predicative uses (*p-use*). The goal of the data flow criteria is to cover paths where values are carried from defs to uses (i.e., definition-use, def-use, and du associations, or simply *du-pairs*) (AMMANN; OFFUTT, 2008).

A path is def-clear w.r.t. a variable x if x is not redefined along this path. Associations between *def* and a *c-use* are expressed as a tuple $\langle i, j, x\rangle$, such that i is a node where variable x is *def*, j is a node with a *c-use* of x, and there is a path from i to j that is def-clear w.r.t. x. Associations between *def* and a *p-use* are expressed as a tuple $\langle i, (j,k), x\rangle$, such that i is a node where variable x is *def* and (j,k) is a transition from i to j with a *p-use* of x.

Rapps and Weyuker (1985) proposed a family of test criteria for which the number of paths is always finite. These criteria complement those designed to CFGs. The three main criteria for DFGs are: (i) *All-defs*, which requires that all *defs*, *c-use* and *p-use* have to be covered at least once by a def-clear path; (ii) *All-uses*, which requires that all *defs* and subsequent uses (*c-uses* or *p-uses*) have to be exercised by at least one def-clear path; and (iii) *All-du-paths*, which requires that all *defs* and subsequent uses (*c-uses* or *p-uses*) have to be exercised by all def-clear and loop-free paths covering one same definition.

Example: The test case `("",invalid)` exercises the path $\{1,8,12\}$, the *def* of `s` and `valid_id` at node `1`, the *c-use* of `valid_id` at node `8`, and the *p-use* of `s` and `valid_id` respectively in transitions `(1,8)` and `(8,12)`. Thus, the following associations are covered: $\langle 1, 8, \text{valid\_id}\rangle$, $\langle 1, (1,8), \text{s}\rangle$, and $\langle 1, (8,12), \text{valid\_id}\rangle$.

2.2.3 Mutation Testing

Mutation testing, or mutation analysis, is a fault-based testing technique that measures the effectiveness of a test suite in terms of its ability to detect faults (JIA; HARMAN, 2011). In mutation testing, we assume the *competent programmer hypothesis* and *coupling effect*.

The competent programmer hypothesis assumes that *good programmers write correct or close-to-correct programs*. Thus, faults introduced through small syntactic modifications may lead to incorrect behaviors. By applying small changes to an SUT, mutation testing aims at designing test cases to reveal the modifications that create incorrect programs. The coupling effect states that complex faults are derived from the composition of simple faults. By assuming these two hypotheses, test cases should be designed to reveal faults.

Mutation operators specify the most frequent kinds of faults and syntactic deviations related to a test artifact (e.g., source code, test model). They are designed for specific languages to create simple syntactic changes (e.g., change the name of variables); or force test cases to include a desired property (e.g., branch coverage in an SUT). A mutant with k-changes is referred to as a k-mutant and when $k > 1$, it is also named *higher order mutant* (JIA; HARMAN, 2009).

If the behavior of a mutant diverges from its original SUT, then it is called a *killed mutant*, and the remaining are *live mutants*. A serious problem of mutation testing is the possible existence of equivalent mutants among alive mutants.

Equivalent mutants are those which syntactic changes do not result in behavioral divergences and, for all test inputs, it always computes the same outputs. Given two programs or mutants, detecting whether they are equivalent is an undecidable problem (OFFUTT; PAN, 1997).

The effectiveness of mutation testing is provided by the *mutation adequacy score*:

$$T_{\text{eff}} = \frac{\#km}{(\#tm - \#em)} \tag{2.1}$$

where $\#km$ refers to the number of killed mutants, $\#tm$ denotes the total number of mutants, and $\#em$ is the number of equivalent mutants. The mutation adequacy score ranges from 0 to 1. The higher this score is, the more adequate is the test suite. The number of detected mutants depends only on the test suite used to evaluate the SUT. The number of equivalent mutants is obtained by manually deciding that a live mutant is equivalent or supported by heuristics. The cost to detect equivalent mutants is one of the main obstacles to adopt mutation testing.

Example: An example of mutant of the *validateIdentifier* program is shown in Source code 2. In this example, we modified the `valid_s` method by replacing `(ch<=’Z’)` with `(ch<=’z’)`. Thus, a list of six special characters (i.e., ’[’, ’\’, ’]’, ’^’, ’_’, and ’‘’) placed between Z and a in the ASCII table will become valid first characters. In this case, test case `("va[",invalid)` is able to detect this fault.

Source code 2 – Fragment of the mutated validateIdentifier program

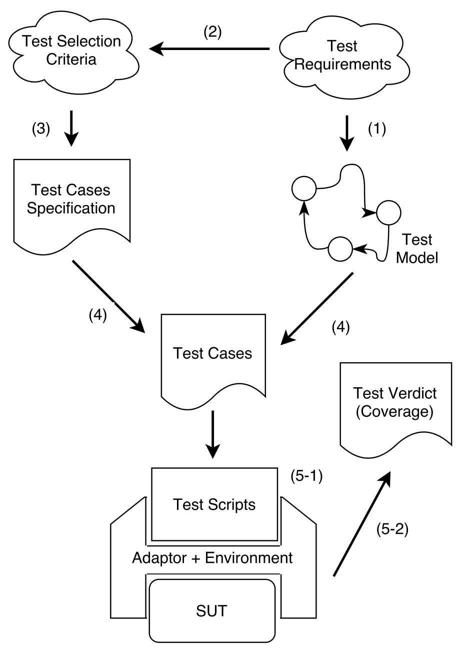
```
public boolean valid_s( char_ch ){
  if( ((ch>='A') && (ch<='z'))          /* MUTATED LINE */
      || ((ch>='a') && (ch<='z')))
    return true;
  else return false;
}
public boolean valid_f(char_ch){
  if( ((ch>='A') && (ch<='Z'))
          || ((ch>='a') && (ch<='z'))
          || ((ch>='0') && (ch<='9')) )
    return true;
  else  return false;
}
```

Source: Elaborated by the author.

2.3 Model-Based Testing

According to Binder (1999), software testing is supposed to be *model-based*. In traditional testing, test models are stuck to test engineers' minds, informally sketched on papers and heavily rely on engineers' expertise (MARINESCU *et al.*, 2015). By contrast, model-based testing (MBT) proposes a model-centric approach that relies on explicit models to support test case selection, generation, and execution (UTTING; PRETSCHNER; LEGEARD, 2012). The generic process of MBT is depicted in Figure 5.

Figure 5 – The generic process of MBT



Source: Adapted from Utting, Pretschner and Legeard (2012).

First, MBT requires a step (1) to specify an explicit *test model* of the SUT. Mostly, test models are (i) created from scratch using requirements documentation, or (ii) reused from design artifacts or pre-existing projects. Test models are specified using formal/semi-formal notations (e.g., state-based, transition-based, operational) which can depict different characteristics of the SUT (e.g., timing issues, non-determinism, continuous/discrete-event). However, building and maintaining test models is very time-consuming and dependent on how familiar test engineers are with the SUT (BERG; RAFFELT, 2005). To address this issue, researchers have explored ways to perform MBT from inferred models.

MBT can take advantage of model inference (ANGLUIN, 1987) and model checking (BAIER; KATOEN, 2008) to extract test models from SUTs when these are incomplete, lack consistency or are outdated. Inferred test models are useful as derived and implicit test oracles as they describe the implementation rather than the intended behavior (MARIANI; PEZZÈ; ZUDDAS, 2015).

Test requirements are used in step (2) as a reference to define test selection criteria. In MBT, test selection criteria can use functional and structural requirements, heuristics for input domain coverage, randomness, or *fault models* (JIA; HARMAN, 2011). After defining test selection criteria, in step (3), test cases are specified in an abstract and non-executable format.

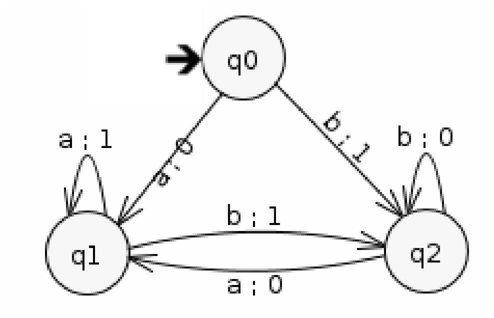
In step (4), based on test cases specifications and a test model, MBT can automatically produce concrete test cases to be executed either manually or as test scripts. In step (5-1), adaptors can be designed to support automatic test execution in specific test environments. At last, in step (5-2), a test verdict reports the test coverage and list accepted and failed test cases.

2.3.1 Finite State Machine Based Testing

Finite state machine-based testing is a specialization of MBT which uses finite state machines (FSM) as test models and aims at proving the equivalence (i.e., conformance) between an SUT and its behavioral specification given as an FSM model.

An FSM is a hypothetical machine M composed of states and transitions (GILL, 1962). Formally, an FSM is defined as a tuple $M = \langle S, s_0, I, O, D, \delta, \lambda \rangle$ where S is a finite set of states, $s_0 \in S$ is the initial state, I is the finite set of input symbols, O is the finite set of output symbols, $D \subseteq S \times I$ is the specification domain, $\delta : D \rightarrow S$ is the transition function, and $\lambda : D \rightarrow O$ is the output function. Figure 6 and Table 5 shows the graphical representation and the state transition table of an FSM with $I = \{a, b\}$, $O = \{0, 1\}$, $S = \{q0, q1, q2\}$ and $s_0 = q0$.

Figure 6 – FSM in graphical representation



Source: Elaborated by the author.

Table 5 – FSM represented as state transition table

$s_0 = q0$	λ		δ	
state \ input	a	b	a	b
q0	0	1	q1	q2
q1	1	1	q1	q2
q2	0	0	q1	q2

Source: Elaborated by the author.

At every moment, an FSM has a single current state $s_i \in S$ that changes to $s_j \in S$ by applying an input symbol $x \in I$ to the transition function, $\delta(s_i, x) = s_j$, and returns an output symbol y from the output function, i.e., $\lambda(s, x) = y$. An input x is defined for s if there is a *defined transition* in state s consuming input (i.e., $(s_i, x) \in D$). FSMs can also be represented as a set of tuples. A tuple (s_i, x, y, s_j) represents a transition outgoing from s_i to s_j with x and y as input and output symbols, respectively. An FSM is *complete* if all inputs are defined for all the states. Otherwise, it is *partial*.

An input sequence $\alpha = x_1 x_2 ... x_n \in I$ is defined in $s \in S$, if there are states $s_1, s_2, ..., s_{n+1}$ such that $s = s_1$ and $\delta(s_i, x_i) = s_{i+1}$, for all $1 \leq i \leq n$. A state s_{n+1} is reachable from s, if there is a transfer sequence $\alpha = x_1 x_2 ... x_n \in I$ from s to s_{n+1} and $\delta(s, \alpha) = s_{n+1}$. If all states are reachable from s_0, the FSM is *initially connected*. If all states are reachable from all states, it is *strongly connected*. The FSM in Figure 6 and Table 5 is completely and strongly connected.

The symbol $\Omega(s)$ denotes all input sequences defined to state s and Ω_M abbreviates $\Omega(s_0)$, referring to all defined input sequences of an FSM M. An FSM M can have a *reset operation*, denoted by r, which takes it to state s_0 regardless the current one. The concatenation of two sequences α and ω is denoted as $\alpha\omega$, or $\alpha \cdot \omega$. A sequence α is prefix of β, denoted by $\alpha \leqslant \beta$, if $\beta = \alpha\omega$, for some sequence ω. An empty sequence is denoted by ε and a sequence α is a proper prefix of β, denoted by $\alpha < \beta$, if $\beta = \alpha\omega$ for an $\omega \neq \varepsilon$.

The set of prefixes of a set T is defined as $pref(T) = \{\alpha \mid \exists \beta \in T \ and \ \alpha < \beta\}$, if $T = pref(T)$, T is *prefix-closed*. Using the definitions of prefix and empty sequence, the concept of transition and output functions can be extended to input sequences. For a state $s_i \in S$, $\delta(s_i, \varepsilon) = s_i$, and $\lambda(s_i, \varepsilon) = \varepsilon$. Given a sequence $\alpha\chi \in \Omega_M$, the output $\lambda(s_0, \alpha\chi)$ is equivalent to $\lambda(s_0, \alpha)\lambda(\delta(s_0, \alpha), \chi)$, and the state reached by $\delta(s_0, \alpha\chi)$ is the same as $\delta(\delta(s_0, \alpha), \chi)$.

A separating sequence for two states s_i and s_j is a sequence γ such that $\gamma \in \Omega(s_i) \cap \Omega(s_j)$ and $\lambda(s_i, \gamma) \neq \lambda(s_j, \gamma)$. The sequence a is the separating sequence for states $q0$ and $q1$ of FSM in Figure 6. In addition, if γ distinguishes every pair of states of an FSM, it is a distinguishing sequence, or simply DS. Formally, if $\lambda(s_i, \gamma) \neq \lambda(s_j, \gamma)$ is valid for all pairs of state $s_i, s_j \in S$, then γ is a DS. Given the FSM in Figure 6, the sequence a is a separating sequence for states q_0 and q_1 since $\lambda(q_0, a) = 0$ and $\lambda(q_1, a) = 1$. Two FSM models $M_S = \langle S, s_0, I, O, D, \delta, \lambda \rangle$ and $M_I = \langle S', s'_0, I, O', D', \delta', \lambda' \rangle$ are equivalent ($M_S \equiv M_I$) if for all states of M_S there is an equivalent state in M_I (i.e., $\forall \ s_i \in S, \ \exists \ s_j \in S' \ \mid s_i \equiv s_j$). Two states are equivalent, $s_i \equiv s_j$, if $\forall \ \alpha \in I, \ \lambda(s_i, \alpha) = \lambda'(s_j, \alpha)$.

An input sequence $\alpha \in \Omega_M$ starting with a reset r is a *test case* of M. Given two test cases $\alpha, \beta \in T$, if α is a proper prefix of β, the execution of β implies the execution of α; thus, α can be discarded from T without affecting test results. A test suite of M consists of a finite set T of test cases, such that there are no two sequences $\alpha, \beta \in T$ where $\alpha < \beta$. The number of symbols of a sequence α is represented by $|\alpha|$ and describes the length of the test sequence α. Given a test case α, the execution cost is calculated as $|\alpha| + 1$ (i.e., the length of α plus one reset operation). The number of resets of T (i.e., number of test cases) is represented by $|T|$.

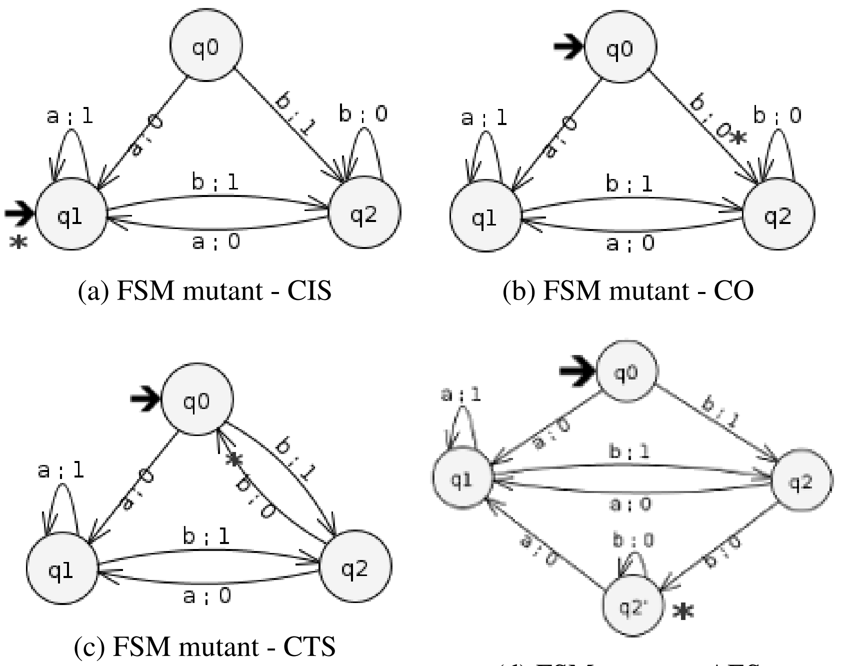
2.3.1.1 Mutation analysis for FSM-based testing

Mutation analysis is frequently used in investigations on FSM-based testing (JIA; HARMAN, 2011; FABBRI *et al.*, 1994). In FSM-based testing, given a specification M, $\Im(M)$ denotes the set of all deterministic FSMs with the same input alphabet of M for which all sequences in Ω_M are defined. Let $m \geq 1$, then $\Im_m(M)$ denotes all FSMs of $\Im(M)$ with at most m states.

Given a specification M with n states, a test suite $T \subseteq \Omega_M$ is m-complete if for each $N \in \Im_m$ distinguishable from M, there is a test case $t \in T$ distinguishing M from N. The $\Im(M)$ set is named *fault domain* for M and can be used to assess the test effectiveness. If the result of running a mutant is different from the original SUT for any test case, the seeded fault is detected and the mutant is *killed*.

The following mutation operators are used in FSM-based testing (CHOW, 1978): *Change Initial State* (CIS), that changes the s_0 of an FSM to s_k, such that $s_0 \neq s_k$; *Change Output* (CO), that modifies the output of a transition (s,x), using a different function $\Lambda(s,x)$ instead of $\lambda(s,x)$; *Change Tail State* (CTS), that modifies the tail state of a transition (s,x), using a different function $\Delta(s,x)$ instead of $\delta(s,x)$; and *Add Extra State* (AES), that inserts a new state such the mutant N is equivalent to M. Figure 7 shows examples of mutants generated from the FSM in Figure 6 using CIS, CO, CTS, and AES operators. Modifications are marked with an asterisk (*).

Figure 7 – Examples of FSM Mutants



(a) FSM mutant - CIS

(b) FSM mutant - CO

(c) FSM mutant - CTS

(d) FSM mutant - AES

Source: Elaborated by the author.

2.3.1.2 FSM-Based Testing Methods

FSM-based testing aims at proving the equivalence (i.e., conformance) between FSM models. To that, some basic sequences are frequently used to obtain partial information about the SUT. They are state cover (Q set) and transition cover (P set).

2.3.1.2.1 State cover and Transition cover

A set of input sequences Q is *state cover* of M if for each state $s_i \in S$ there is a sequence $\alpha \in Q$ such that $\delta(s_0, \alpha) = s_i$ and it includes the empty sequence ε to reach the initial state. A set of inputs P is *transition cover* of M if for each transition $(s, x) \in D$ there are sequences $\alpha, \alpha x \in P$, such that $\delta(s_0, \alpha) = s$, $x \in I$, and it includes the empty sequence ε to reach the initial state. The state cover and transition cover sets of the FSM in Figure 6 are respectively $Q = \{\varepsilon, a, b\}$ and $P = \{\varepsilon, a, aa, ba, b, ab, bb\}$.

To identify states and transitions of FSMs, traditional FSM-based testing methods require a pre-defined set named *characterization set*. A characterization set (W set) contains at least one separating sequence for each pair of states (i.e., $\forall s_i, s_j \in S, i \neq j$, $\exists \alpha \in W$ such that $\lambda(s_i, \alpha) \neq \lambda(s_j, \alpha)$). The characterization set of the FSM in Figure 6 is shown in Table 6.

Table 6 – Example of characterization set (W set)

input / state	a	b
q0	0	1
q1	1	1
q2	0	0

Source: Elaborated by the author.

A separating family, or harmonized state identifiers, is a set of sequences H_i for all states $s_i \in S$. The H_i set satisfies the following condition: $\forall s_j \in S, s_i \neq s_j$ there are $\beta \in H_i, \gamma \in H_j$ with a common prefix α such that $\alpha \in \Omega(s_i) \cap \Omega(s_j)$ and $\lambda(s_i, \alpha) \neq \lambda(s_j, \alpha)$. In the worst case, the separating family is the W set itself.

2.3.1.2.2 W method

The W method is one of the most classic FSM-based testing methods (CHOW, 1978; VASILEVSKII, 1973). It uses the P set to traverse all the FSM transitions concatenated to the W set to identify states reached (i.e., $T_W = \{P \cdot W\}$). The W method can also be extended to detect an estimated number of n extra states by using the traversal set $\bigcup_{i=0}^{m-n}(I^i)$. The set I^i contains all input sequences of length i combining the symbols of I and the traversal set consists of the union of all sets I^i with sequences of length ranging from 0 to $(m-n)$, the number of extra states.

The W set formed by the concatenation of P, the $\bigcup_{i=0}^{m-n}(I^i)$, and the W set is able to detect at most $(m-n)$ extra states. Assuming the FSM in Figure 6, no extra states ($m = n$) or proper prefixes, the test suite generated by the W method is $T_W = \{aaa, aab, aba, abb, baa, bab, bba, bbb\}$, and $|T_W| = 8$.

2.3.1.2.3 Wp method

The partial W method (Wp) is an improved version of its antecessor, the W method. It has full fault detection capability and yields shorter test suites than the W method (FUJIWARA *et al.*, 1991). The term *partial* applies as it uses a subset of the W set. It has two stages:

States verification: A C_1 set is generated by concatenating the state cover and the characterization sets (i.e., $C_1 = Q \cdot W$); The C_1 set verifies if the number of states in the implementation M_I is equal to the number of states in the specification M_S.

Uncovered transitions verification: After the C_1, the set of transitions $Q \subset P$ is already verified. Thus, only a subset $R = P - Q$ has to be assessed and concatenated to subsets of W, written $C_2 = R \otimes W$. More precisely, $R \otimes W = \bigcup_{\alpha \in R} \alpha \cdot W_i$, such that the subset $W_i \subset W$ distinguishes $s_i = \delta(s_0, \alpha)$ from all the other states of S.

The test suite of Wp is the union of C_1 and C_2, (i.e., $T_{\mathrm{Wp}} = C_1 \cup C_2$). Assuming the FSM in Figure 6, no extra states ($m = n$) or proper prefixes, the test suite generated by the W_p method is equals to $T_{\mathrm{Wp}} = \{aaa, abb, bbb, baa\}$, and $|T_{\mathrm{Wp}}| = 4$.

2.3.1.2.4 HSI method

The Harmonized State Identifiers (HSI) method (PETRENKO; BOCHMANN, 1995) uses the set of state identifiers H_i to distinguish each state $s_i \in S$ from the FSM model. First, the HSI method concatenates the state cover set to the harmonized identifiers set which is, in the worst case, the W set itself. Later, the transition cover set is also concatenated with harmonized identifiers to cover non-traversed transitions. The HSI method can also be used on partial FSM. Assuming the FSM in Figure 6, no extra states or proper prefixes, the test suite generated by the HSI method is equals to $T_{HSI} = \{aaa, aba, abb, baa, bba, bbb\}$, and $|T_{HSI}| = 6$.

2.3.1.2.5 SPY method

The SPY method (SIMÃO; PETRENKO; YEVTUSHENKO, 2009) is a recent method proposed to generate m-complete test suites by concatenating test cases *on-the-fly*. First, all sequences of the Q set are concatenated to their respective H_i identifiers based on sufficient conditions. Thus, test tree branching can be avoided as much as possible, and test suite length and the number of resets can be reduced. Experimental studies have shown that SPY can generate test suites on average 40% shorter, and longer test cases compared to traditional methods, such as W and HSI. Moreover, SPY has achieved higher effectiveness even if the number of extra states is underestimated (ENDO; SIMAO, 2013). Assuming the FSM in Figure 6, no extra states or proper prefixes, the test suite generated by the SPY method is equals to $T_{SPY} = \{aaaba, abbb, baa, bba\}$, and $|T_{SPY}| = 4$.

2.3.1.3 Related formalisms

Transition-based notations aim at describing SUTs based on the changes between their states (UTTING; PRETSCHNER; LEGEARD, 2012). In this section, we briefly present two formalisms related to Mealy machines that can also be used for testing purposes.

2.3.1.3.1 Deterministic Finite Automata - DFA

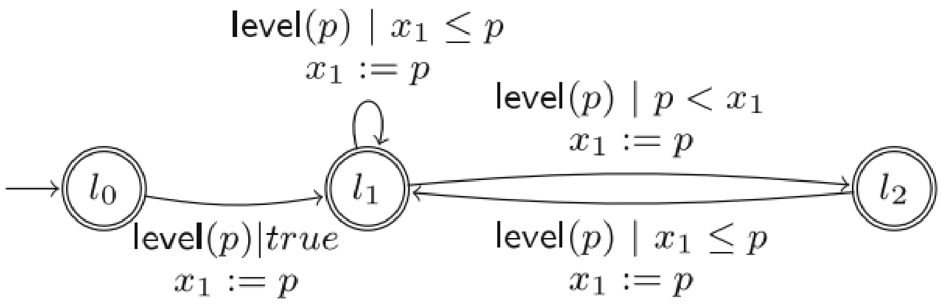
A deterministic finite-state automaton (DFA) is a 5-tuple $M = (\Sigma, Q, \delta, q_0, F)$, where Σ is the finite set of input symbols (i.e., letters) called alphabet, Q is a non-empty set of states, $\delta : Q \times \Sigma \rightarrow Q$ is the transition function, $q_0 \in Q$ is the initial state of M, and $F \subset Q$ is a set of accepting states. Initially, the machine M starts in the initial state q_0 and reads an input $w = a_1 a_2 ... a_n \in \Sigma^*$. It uses the transition function δ to change to the next state, given its current state and an input symbol read.

A finite run of a DFA M on an input $w = a_0\, a_1 \ldots a_n \in \Sigma^*$ is a sequence of states $q_0\, q_1 \ldots q_n$, such that $\delta(q_i, a_i) = q_{i+1},\ 0 \leq i < n$. An input is accepting if $\delta(q_0, w) = q_{n+1} \in F$. The set of accepting inputs (i.e., language) that a DFA M recognizes is denoted by $L(M)$. Among many applications, DFAs have been used for recognizing behavioral patterns (WENDEHALS; ORSO, 2006), representing regular expressions (HOPCROFT, 2007), and testing XML schemas (AMMANN; OFFUTT, 2008).

2.3.1.3.2 Extended Finite State Machines - EFSM

An EFSM can be described as a compressed representation of an FSM for modeling control-flow and data-flow aspects of SUTs (EL-FAKIH *et al.*, 2017). Essentially, it extends the Mealy machine model with input and output parameters, (context) variables, (update or assignment) operations, and predicates (or guards) defined over variables and input parameters. Figure 8 shows an example of EFSM named *water pump* (CASSEL *et al.*, 2016).

Figure 8 – The water pump EFSM



Source: Cassel *et al.* (2016).

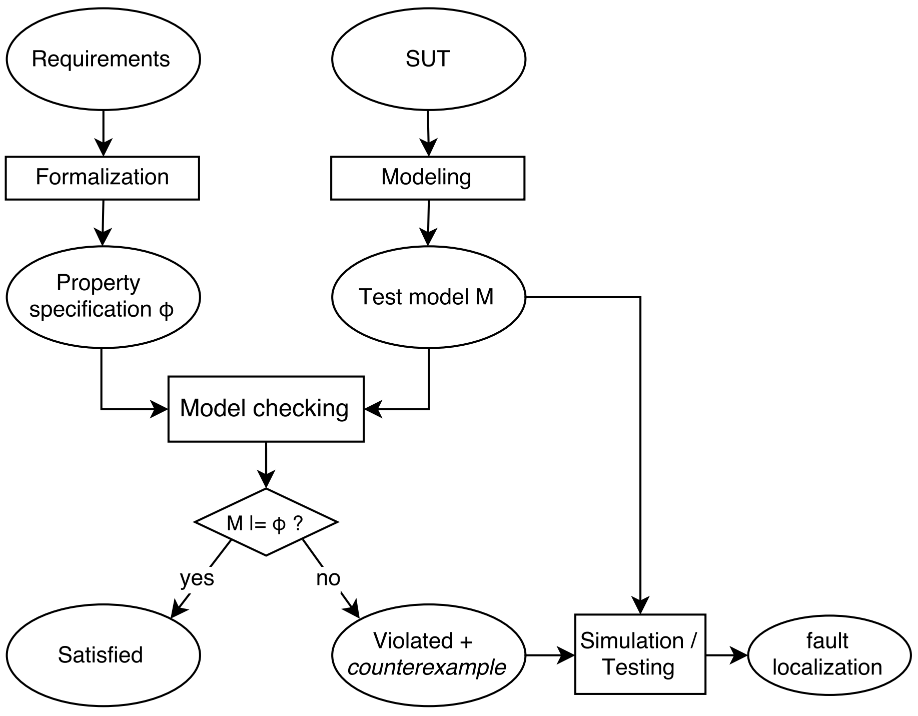
The water pump EFSM has three accepting states: l_0 (which is the initial state), l_1 (where the water level is above minimum), and l_2 (where the water level is below the minimum). There is one variable x_1 that stores the current water level. Transitions are denoted by arrows and labeled with an input parameter `level`(p), a guard to compare p to x_1, and an assignment to x_1. The variable x_1 is initialized by the transition from l_0 to l_1. Input symbols that do not match any transition lead to a sink state that we omitted. During operation, the EFSM moves between states l_1 and l_2 depending on the current water level x_1 and the threshold p.

Assuming that all the domains are finite, an EFSM can be unfolded into a flat FSM by expanding the values of the parameters and variables. Thus, traditional FSM-based testing criteria become applicable, but realistic, mainly because of state/test explosion problem (PETRENKO; BORODAY; GROZ, 2004). Test case generation for EFSMs can be carried out using traditional criteria for control-flow (ALLEN, 1970) and data-flow (RAPPS; WEYUKER, 1985), graph-coverage (AMMANN; OFFUTT, 2008). El-Fakih *et al.* (2017) present a comprehensive assessment of the most known types of EFSM test selection criteria.

2.3.2 Model Checking

Model checking is a formal verification technique that explores all possible states of a test model M in a systematic way to determine whether a temporal logic property φ is satisfied ($M \models \varphi$) or not ($M \not\models \varphi$) (BAIER; KATOEN, 2008). Figure 9 shows an schematic view of the model checking approach.

Figure 9 – Schematic view of the model checking approach



Source: Adapted from Baier and Katoen (2008).

If no property violations are detected, then the correctness of the SUT is proved. Otherwise, a counterexample is returned to identify and fix faults. The main examples of temporal logics are Linear Temporal Logic (LTL) (PNUELI, 1977), and Computational Tree Logic (CTL) (CLARKE; EMERSON, 1982).

In model checking, properties are specified using logical operators from propositional (i.e., not ($\neg$), and ($\wedge$), or ($\vee$), and implication ($\implies$)) and temporal logics. Temporal logic operators allow reasoning about computation paths and trees (FRASER; WOTAWA; AMMANN, 2009). The main temporal logic operators are: *neXt* (X or $\circ$), *Globally* (G or $\Box$), *Finally* (F or $\Diamond$). *Until* (U or $\mathscr{U}$), *Always* (A or $\forall$), and *Exists* (E or $\exists$).

The *linear* word of LTL stands for its path-based notion of time (i.e., at each moment, there is only one possible successor state and a unique possible future). The *tree* word of CTL refers to the infinite, directed tree of states that express the several different possible futures of an SUT (e.g., possibly the system never goes down, the system always eventually goes down and is operational until going down).

Since Pnueli (1977), model checking has evolved to a practical solution for real-world problems, capable of finding errors early in the development process for many classes of models (MILLER; WHALEN; COFER, 2010) and verifying test models with over 10^{120} reachable states (GRUMBERG; VEITH, 2008).

2.4 Test Case Prioritization

MBT tends to generate a large number of test cases that is reused as the SUT evolves. Due to time and resources constraints, only a part of the test suites is often used. To cope with these limitations, different techniques can be used to improve the cost-effectiveness of test suites.

According to Yoo and Harman (2012), these techniques can be classified into three groups:

Test Suite Reduction: Techniques for removing redundant test cases permanently;

Test Case Selection: Techniques for selecting some of the test cases and focus on changed parts of an SUT; and

Test Case Prioritization: Techniques for identifying an efficient ordering of the test cases to maximize certain properties.

Test suite reduction and test case selection can reduce the time for testing, but may omit important test cases able to detect certain faults (OURIQUES, 2015). Test case prioritization aims at finding an ideal ordering for executing test cases so that maximum benefits can be obtained, even if the execution is prematurely halted at some arbitrary point (YOO; HARMAN, 2012).

Formally, the test prioritization problem is defined as follows:

Definition 1. (Test Case Prioritization Problem)

Given: a test suite T, the set PT of all possible permutations of T, and a function $f : PT \to \mathbb{R}$ that describes a quantitative test criterion.

Problem: find a permutation $T' \in PT$ such that $(\forall T'')$ $(T'' \in PT)$ $(T'' \neq T')$ $[f(T') \geq f(T'')]$

The goal of test case prioritization is to maximize (or minimize) the function f which describes a *test criterion* (e.g., cumulative effectiveness).

The quality of order is described by the Average Percentage Faults Detected (APFD) metric (ELBAUM *et al.*, 2004). The APFD is a metric commonly used in test prioritization research and is defined as follows:

$$\text{APFD} = \frac{\sum_{i=1}^{n-1} F_i}{n \times l} + \frac{1}{2n} \tag{2.2}$$

In Equation 2.2, n denotes the total number of test cases, l defines the number of faults under consideration, and F_i specifies the number of faults detected by a test case i. The APFD value depicts the cumulative effectiveness of a test suite ordering as a value ranging from 0 to 1. The APFD metric can also be described as the area below the curve describing the cumulative effectiveness of a test suite. The greater the APFD and the area below the curve are, the better is the ordering of the test cases.

Besides the cumulative effectiveness, other test criteria can be considered, such as state coverage, transition coverage, code coverage, and reliability.

Example: Let an hypothetical set of five test cases A, B, C, D, E able to detect ten faults, as shown in Table 7.

Table 7 – Example of test cases with fault-detection capability

	Fault revealed by test case									
Test case	1	2	3	4	5	6	7	8	9	10
A	●				●					
B	●				●	●	●			
C	●	●	●	●	●	●	●			
D					●					
E								●	●	●

Source: Adapted from Elbaum *et al.* (2004).

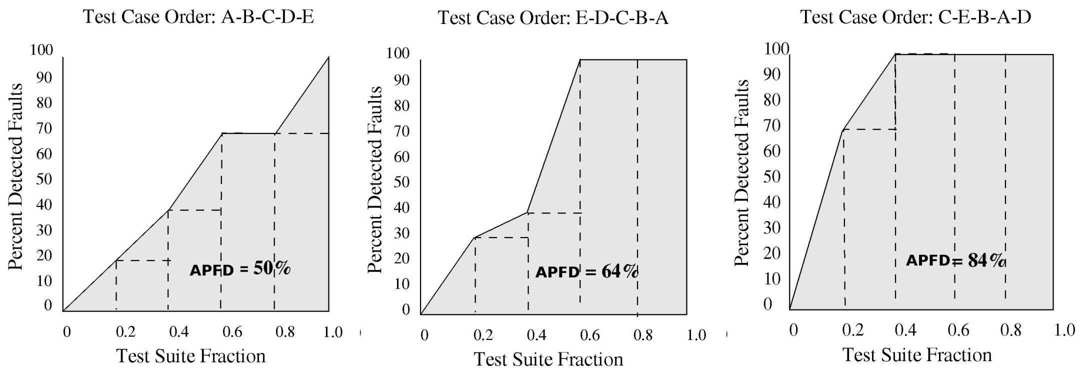
In this example, test cases C, E and A can respectively detect 70%, 30% and 20% of all faults. Thus, depending on the order of execution, it is possible to increase the cumulative effectiveness to 84% (e.g., by placing C and E). Table 8 and Figure 10 show the APFD of three test orderings. The APFD metric says that test suite $T3$ performs better than $T2$ and $T1$.

Table 8 – APFD value for the test cases example

Test suite id	Test cases ordering	APFD
$T1$	A,B,C,D,E	0.5
$T2$	E,D,C,B,A	0.64
$T3$	C,E,B,A,D	0.84

Source: Elbaum *et al.* (2004).

Figure 10 – Examples of prioritizations



(a) Prioritization T1 (b) Prioritization T2 (c) Prioritization T3

Source: Elbaum *et al.* (2004).

2.5 Tooling Support

Manual software testing can be costly and error-prone. Thus, test automation is a highly important attribute for software testing, especially MBT. In this section, we discuss a list of tools for software testing.

2.5.1 Plavis/FSM & JPLAVIS/FSM

Plavis is the acronym for *PLAtform for software Validation & Integration on Space systems*. The Plavis/FSM is an MBT tool that automates test case generation from FSM models (SIMAO *et al.*, 2005) which integrates the Proteum/FSM tool, an FSM-based testing web platform that supports test generation and mutation analysis using FSM models (FABBRI; MALDONADO; DELAMARO, 1999).

The JPlavis/FSM is a Java-based desktop tool for FSM-based testing developed to be an evolution of the Proteum/FSM (PINHEIRO, 2012). It has a more intuitive graphical user interface to model FSMs and supports a larger number of FSM testing methods, such as W (CHOW, 1978), HSI (PETRENKO; BOCHMANN, 1995) and SPY (SIMÃO; PETRENKO; YEVTUSHENKO, 2009).

2.5.2 NuSMV: a new symbolic model checker

The *new Symbolic Model Verifier* (NuSMV) (FBK, 2015) is the result of the reengineering and reimplementation of the CMU SMV model checking tool (CLARKE *et al.*, 1996). The NuSMV tool allows the representation of synchronous and asynchronous finite state systems using different data types, such as bounded integer, boolean, sets, arrays, and enumerates; and supports the verification of formulae specified using Computation Tree Logic (CTL) and Linear Temporal Logic (LTL) (BAIER; KATOEN, 2008).

User interaction can be carried using an interactive shell, batch mode and a GUI interface (CAVADA *et al.*, 2010). Regarding the performance of the NuSMV tool, by using optimized data structures for storing and manipulating large sets of states, such as Binary Decision Diagrams (BDD) (BRYANT, 1986; MCMILLAN, 1993; CLARKE *et al.*, 1996), it has been capable of analyzing models with over 10^{120} reachable states (COFER; WHALEN; MILLER, 2008).

2.5.3 Z3 Theorem Prover

The Z3 theorem prover is a high-performance satisfiability modulo theory (SMT) solver from Microsoft Research (2017). The Z3 tool is targeted at solving problems of software verification and analysis, such as extended static checking, predicate abstraction, test case generation, and bounded model checking over infinite domains (MOURA; BJØRNER, 2008). The Z3 tool has been used in many other Microsoft projects, such as Spec# (BARNETT; LEINO; SCHULTE, 2005; DELINE; LEINO, 2005), Pex (TILLMANN; HALLEUX, 2008), and VCC (COHEN *et al.*, 2009).

Spec# is an extension of the C# programming language that supports distinguishing non-null object references from possibly-null object references, method specifications like pre- and postconditions, managing exceptions, and constraining data fields of objects (BARNETT; LEINO; SCHULTE, 2005). The Spec# system is fully integrated into the Microsoft Visual Studio environment and includes not only a language and compiler, but an automatic program verifier, called Boogie, which checks specifications statically (DELINE; LEINO, 2005).

Pex is an automatic white-box testing tool for .NET programs. It used model-checking to steer an SUT along different execution paths and determine test inputs. Pex uses the Z3 tool as constraint solver to produce new test inputs which exercise different program behavior. Pex has been integrated to Visual Studio as a plugin and various unit testing frameworks (TILLMANN; SCHULTE, 2006; TILLMANN; HALLEUX, 2008).

The VCC tool is an industrial-strength environment for verifying concurrent systems written in C. It VCC takes an SUT annotated with function contracts, state assertions, and type invariants and attempts to prove the correctness of such annotations. This tool has been used to verify Microsoft Hyper-V hypervisor (COHEN *et al.*, 2009).

2.6 Final remarks

In this chapter, we introduced the main concepts of software testing. Software testing aims at finding errors by executing SUTs under specified conditions, observing and evaluating the obtained results. To that, testing techniques and criteria support test case generation and evaluation. However, traditional software testing is often ad hoc and heavily relies on testers' expertise. To tackle this issue, researchers have proposed a model-centric approach which relies on explicit models to support test case selection, generation, and execution. This is named MBT.

MBT has been successfully applied to different domains and complexities. However, building and maintaining models for MBT is a very time-consuming task. To overcome this issue, researchers have explored ways to extract test models from SUTs using model inference algorithms. In the next chapter, we present the main concepts, applications, and challenges of model inference.

CHAPTER

3

MODEL INFERENCE

As mentioned before, software testing is supposed to be *model-based* (BINDER, 1999). In traditional testing, test models are found stuck to engineers' minds and informally sketched on papers (MARINESCU *et al.*, 2015). By contrast, model-based testing (MBT) relies on explicit test models as a central artifact (UTTING; PRETSCHNER; LEGEARD, 2012). The problem of MBT is that building useful test models is time-consuming and highly dependent on how familiar test engineers are with an SUT (BERG; RAFFELT, 2005). Moreover, component-based software engineering and service-oriented architectures impose hurdles to the adoption of MBT due to missing or incomplete specification models (ISBERNER; HOWAR; STEFFEN, 2014). To tackle these issues, researchers have proposed to infer behavioral models from software and hardware systems (MARIANI; PEZZÈ; ZUDDAS, 2015).

According to Weyuker (1983), MBT and model inference can be thought as being inverse processes. The MBT process begins with an SUT and specification (i.e., the test model), and looks for test cases that characterize a conformance relation (BROY *et al.*, 2005). Model inference starts with a set of queries (i.e., test cases), and derives a behavioral specification (i.e., test model) that fits the behavior of a given System Under Learning (SUL) (IRFAN; ORIAT; GROZ, 2013).

Model inference has been studied under different names, such as system identification, grammatical inference, regular inference, regular extrapolation, active automata learning, or protocol state fuzzing (VAANDRAGER, 2017). It has been applied to different problems, such as automated quality assurance at Springer Verlag (NEUBAUER *et al.*, 2012), testing brake-by-wire systems at Volvo (FENG *et al.*, 2013), integration testing at Orange (SHAHBAZ; GROZ, 2014), and software refactoring at Phillips (SCHUTS; HOOMAN; VAANDRAGER, 2016).

In this chapter, we introduce the model inference procedure using the L_M^* algorithm, an extension of the classic L^* to mealy machines. Optimizations to reduce the cost of model inference and examples of application domains are also briefly discussed. At last, we present a list of tools to support model inference and close this chapter with the final remarks.

3.1 The Minimally Adequate Teacher framework

Coined by Angluin (1987), model inference is an *active learning* procedure which aims at extracting the behavioral model of an SUL by pursuing inputs and observing outputs based on previous information (IRFAN; ORIAT; GROZ, 2013). It differs from *passive learning* where models are synthesized from a finite set of logs or traces without further interaction with the SUL (TRETMANS, 2011). The model inference procedure can be described as the Minimally Adequate Teacher (MAT) framework (ANGLUIN, 1987), depicted in Figure 11.

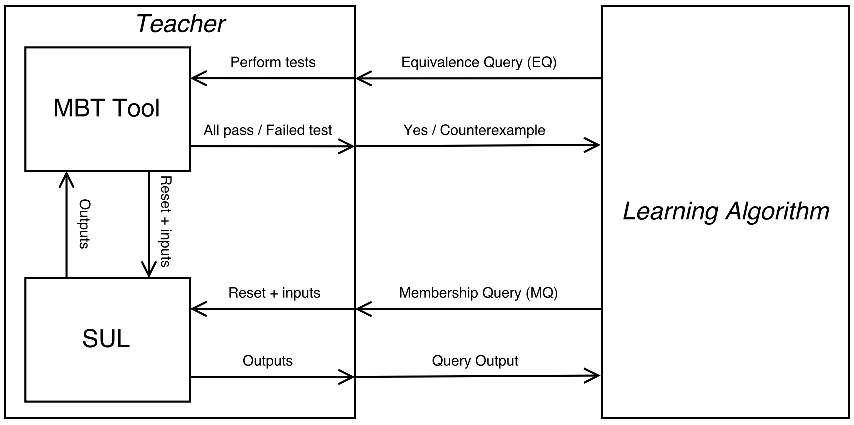


Figure 11 – The Minimally Adequate Teacher framework

A `learning algorithm` iteratively performs `queries` (i.e., tests) and observe `outputs` to build a `hypothesis` (i.e., test model) supported by a `teacher` (i.e., oracle) that can answer if such hypothesis is correct or not, by returning `counterexamples`. The MAT framework has two phases: *hypothesis construction*, when a model is built by posing `Membership Queries (MQ)`; and *hypothesis validation*, when the hypothesis is subjected to `Equivalence Queries (EQ)`.

An `Equivalence Query (EQ)` checks whether an `hypothesized model` denotes the real behavior of an SUL. The result of an `EQ` is `yes`, if the hypothesis is correct (i.e., `all pass`), otherwise a `counterexample` points non-conformances (i.e., `failed test`). Often, teachers use `MBT` to compute `inputs` and `outputs` and `reset` the SUL to its initial state.

A `Membership Query (MQ)` consists of `inputs` and `reset` performed to gain knowledge (i.e., `query output`). If a teacher replies with a `counterexample`, other `MQ`s are asked to improve the `hypothesized model` until the hypothesis becomes correct. The outputs are organized in an *observation table* that is incrementally refined to build a hypothesis.

Since Angluin (1987) seminal paper, the MAT framework has been extended to several other formalisms, such as nondeterministic finite automata (NFA) (BOLLIG *et al.*, 2009), mealy machines (NIESE, 2003; LI; GROZ; SHAHBAZ, 2006; SHAHBAZ; GROZ, 2009), register automata (AARTS *et al.*, 2012; CASSEL FALK HOWAR, 2015; AARTS *et al.*, 2015; CASSEL *et al.*, 2016), and input-output transition systems (IOTS) (VOLPATO; TRETMANS, 2015).

3.2 Inferring mealy machines

By using the MAT framework, Angluin (1987) proposed the L^* algorithm and proved that it can learn an unknown regular language (i.e., deterministic finite automata - DFA) in polynomial time, given an *a priori* fixed input alphabet I. After that, several variants of the L^* algorithm have been proposed to infer richer models such as Mealy machines.

Groce, Peled and Yannakakis (2002) proposed to model the input alphabet of SULs as a collection of inputs and outputs (i.e., $\Sigma = I \cup O$), but this increment at the size of the input domain directly affects the time complexity. Recently, Shahbaz and Groz (2009) extended Angluin (1987) work to optimize the inference of mealy machines and proposed the L_M^* algorithm. In this section, we will introduce the L_M^* algorithm and take the mealy machine depicted in Figure 12 as SUL.

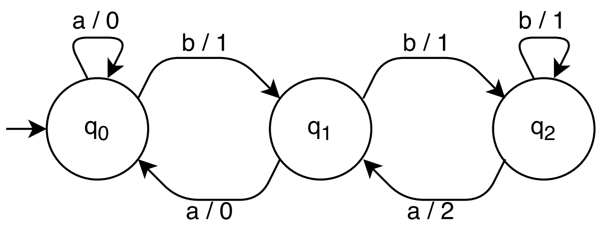


Figure 12 – Example of mealy machine taken as SUL

3.2.1 Observation Table

The L_M^* builds a hypothesis by iteratively asking `MQ` until a correct model is constructed from an observation table (SHAHBAZ; GROZ, 2009). In this case, the `MQ` is called output queries (`OQ`) as mealy machines deal with outputs, but acceptance of words. An observation table (OT) is defined as a triple $OT = (S_M, E_M, T_M)$, where $S_M \subseteq I^*$ is a prefix-closed set of inputs labeling the rows of OT, $E_M \subseteq I^+$ is a suffix-closed set of inputs labeling the columns of OT, and $T_M : (S_M \cup S_M \cdot I) \times E_M \rightarrow O^+$ maps inputs to outputs.

Each cell of an OT keeps the output symbols of an `OQ` composed by a prefix $pre \in S_M$ and a suffix $suf \in E_M$ which generates an output $\lambda(q_0, pre \cdot suf) = T_M(pre, suf)$. Two rows $pre_1, pre_2 \in (S_M \cup S_M \cdot I)$ of an OT are equivalent iff $\forall suf \in E_M$, $T_M(pre_1, suf) = T_M(pre_2, suf)$, and denoted as $pre_1 \cong pre_2$. The equivalence class of a row r is denoted $[r]$.

There are two properties that must hold to construct an FSM: *closeness* and *consistency*. An OT is *closed* if for all $pre_1 \in (S_M \cup I)$, there is a $pre_2 \in S_M$ where $pre_1 \cong pre_2$. An OT is *consistent* whenever two rows $pre_1, pre_2 \in S_M$, for $pre_1 \cong pre_2$, then $\forall\ \alpha \in I\ pre_1 \cdot \alpha \cong pre_2 \cdot \alpha$. When an OT becomes closed and consistent and no `counterexample` is found, a hypothesis is built as $\mathscr{H} = (Q_M, q_{0_M}, I, O, D, \delta_M, \lambda_M)$ where $Q_M = \{[pre] | pre \in S_M\}$, $q_{0_M} = [\varepsilon]$, $\delta_M([pre], i) = [pre \cdot i], \forall pre \in S_M, i \in I\}$, and $\lambda_M([pre], i) = T_M(pre, i), \forall pre \in S_M, i \in I\}$.

3.2.2 The Algorithm L_M^*

The L_M^* (SHAHBAZ; GROZ, 2009) is presented in Algorithm 1. As input it receives the SUL and an *a priori* fixed input alphabet I and, as output, it returns a hypothesized model $\mathscr{H}$.

Algorithm 1 – The L_M^* Algorithm

```
Input: The SUL and the input alphabet $I$
Output: Hypothesized mealy machine $\mathscr{H}$
 1: Initialize the rows $S_M = \{\varepsilon\}$, columns $E_M = I$ and $S_M \cdot I = \{\varepsilon \cdot i\}$, for all $i \in I$
 2: Complete $OT = (S_M, E_M, T_M)$ by asking OQs, for all $s \in (S_M \cup S_M \cdot I)$ and $e \in E_M$
 3: repeat
 4:     while $(S_M, E_M, T_M)$ is not closed or not consistent do
 5:         if $(S_M, E_M, T_M)$ is not consistent then
 6:             Find $s_1, s_2 \in S_M$, $e \in E_M$, $i \in I$ such that $s_1 \cong s_2$, but $T_M(s_1 \cdot i, e) \neq T_M(s_2 \cdot i, e)$
 7:             $E_M = E_M \cup \{i \cdot e\}$                               ▷ add $i \cdot e$ to $E_M$
 8:             Complete $(S_M, E_M, T_M)$ by asking OQ for the new column $i \cdot e$
 9:         end if
10:         if $(S_M, E_M, T_M)$ is not closed then
11:             Find $s_1 \in S_M \cdot I$, such that $s_1 \ncong s_2$, for all $s_2 \in S_M$
12:             $S_M \cdot I = S_M \cdot I \setminus s_1$;   $S_M = S_M \cup \{s_1\}$          ▷ move $s_1$ to $S_M$
13:             $S_M \cdot I = S_M \cdot I \cup \{s_1 \cdot i\}$, for all $i \in I$     ▷ add $s_1 \cdot i$ to $S_M \cdot I$, for all $i \in I$
14:             Complete $(S_M, E_M, T_M)$ by asking OQ for the new added rows
15:         end if
16:     end while
17:     Build an hypotesized model $\mathscr{H}$ from $OT = (S_M, E_M, T_M)$
18:     Ask an EQ to the MAT
19:     if MAT replies counterexample to EQ then
20:         Process counterexample                      ▷ e.g., add all the prefixes to $S_M$
21:         Complete $(S_M, E_M, T_M)$ by asking OQ for the missing elements
22:     end if
23: until MAT replies Yes to an EQ
24: return Hypothesis $\mathscr{H}$ built from $OT = (S_M, E_M, T_M)$
```

Source: Adapted from Shahbaz and Groz (2009).

The L_M^* algorithm starts with $S_M = \{\varepsilon\}$, $E_M = I$ and $S_M \cdot I = \{\varepsilon \cdot i\}$, $\forall\, i \in I$ and output queries $s \cdot e$ are built by concatenating all the elements $s \in S_M \cup S_M \cdot I$ and $e \in E_M$. The outputs obtained for each *OQ* are used to complete the *OT* (Line 2). The main loop (Lines 4-23) keeps testing if the *OT* is complete and consistent until a correct hypothesis is found. If the *OT* is *not consistent*, the algorithm finds two equivalent rows $s_1, s_2 \in S_M$, one column $e \in E_M$, and an input $i \in I$ that $T_M(s_1 \cdot i, e) \neq T_M(s_2 \cdot i, e)$ (Line 6). After, an input $i \cdot e$ is added as a new column (Line 7) and new `OQ`s are asked. If the *OT* is *not closed*, the algorithm finds a row $s_1 \in S_M \cdot I$ not equivalent to any row in S_M (Line 11). The s_1 row is moved from $S_M \cdot I$ to S_M (Line 12), new rows $s_1 \cdot i, \forall\, i \in I$, are added to $S_M \cdot I$ (Line 13) and new `OQ`s are asked. After finding a consistent and closed *OT*, a hypothesized model is built and an `EQ` is asked (Line 18). If a `counterexample` is found, it is processed and `OQ`s are asked for the missing elements. The algorithm ends if a correct hypothesis $\mathscr{H}$ is found. An example is presented using the FSM in Figure 12.

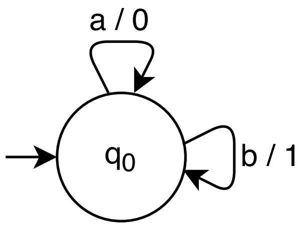
Example

Assuming the FSM in Figure 12 as SUL, the L_M^* algorithm starts filling the OT as $S_M = \{\varepsilon\}$, $E_M = \{I\} = \{a, b\}$, and $S_M \cdot I = \{\varepsilon \cdot i\}, \forall i \in I$, thus $S_M \cdot I = \{a, b\}$. The first OT_1, in Table 9, is closed and consistent. The hypotesized model in Figure 13.

Table 9 – First observation table

E_M / $S_M \cup S_M \cdot I$	a	b
ε	0	1
a	0	1
b	0	1

Figure 13 – A hypothesized model



Now, it asks an EQ which has to reply a `counterexample`. For simplification purposes, we assume the teacher returns *bba* as `counterexample` to show that $\mathscr{H} \not\equiv \mathscr{M}$ and Angluin (1987) approach to process counterexamples. Thus, the learner adds all its prefixes to S_M (i.e., $\{\varepsilon, b, bb, bba\}$) and constructs a second OT_2, shown in Table 10.

This second OT_2 is still not consistent as rows ε and b are equivalent but $T_M(\varepsilon \cdot b, a) \neq T_M(b \cdot b, a)$. Thus, input *ba* is added to E_M. At this point, the learner generates a third OT_3, shown in Table 11 that is closed and consistent.

Table 10 – Second observation table

E_M / $S_M \cup S_M \cdot I$	a	b
ε	0	1
b	0	1
bb	2	1
bba	0	1
a	0	1
ba	0	1
bbb	2	1
bbaa	0	1
bbab	2	1

Table 11 – Third observation table

E_M / $S_M \cup S_M \cdot I$	a	b	ba
ε	0	1	0
b	0	1	2
bb	2	1	2
bba	0	1	2
a	0	1	1
ba	0	1	1
bbb	2	1	2
bbaa	0	1	1
bbab	2	1	2

In the worst case, the L_M^* has polynomial complexity $O(|I|^2mn + |I|mn^2)$ on the size $|I|$ of the input domain, the length m of the longest counterexamples, and the minimal number n of states of the SUL (SHAHBAZ; GROZ, 2009).

3.2.3 Searching and Processing Counterexamples

In model inference of black box systems, there is often no oracle to answer EQ. To tackle this, different approaches to search counterexamples have been used, such as random sampling the input domain (ANGLUIN, 1987), or inputs not previously used (HOWAR; STEFFEN; MERTEN, 2010). After finding a counterexample, another step with significant impact on the time complexity is processing counterexamples.

Angluin (1987) requires adding all prefixes to the observation table, as shown in the previous example. Rivest and Schapire (1993) improve the worst case number of queries required to learn mealy machines by adding only a single distinguishing sequence to E_M. Maler and Pnueli (1995) add to E_M all the suffixes of the counterexample not yet included. Shahbaz and Groz (2009) drop the longest prefix of a counterexample that matches any element of $S_M \cup S_M \cdot I$, before adding the suffixes of its suffix to E_M. Irfan, Oriat and Groz (2010) propose the Suffix1by1 method which takes the qualities of all processing methods presented previously.

Irfan, Oriat and Groz (2013) performed an experiment comparing each of the methods aforementioned on random FSMs with different numbers of states. Figure 14 shows the number of queries for different numbers of states. At this experiment, Shahbaz and Groz (2009) performed better than the other methods, followed by Suffix1by1 in the second place. The Suffix1by1 got the second place as the models conjectured are always consistent with the observation table.

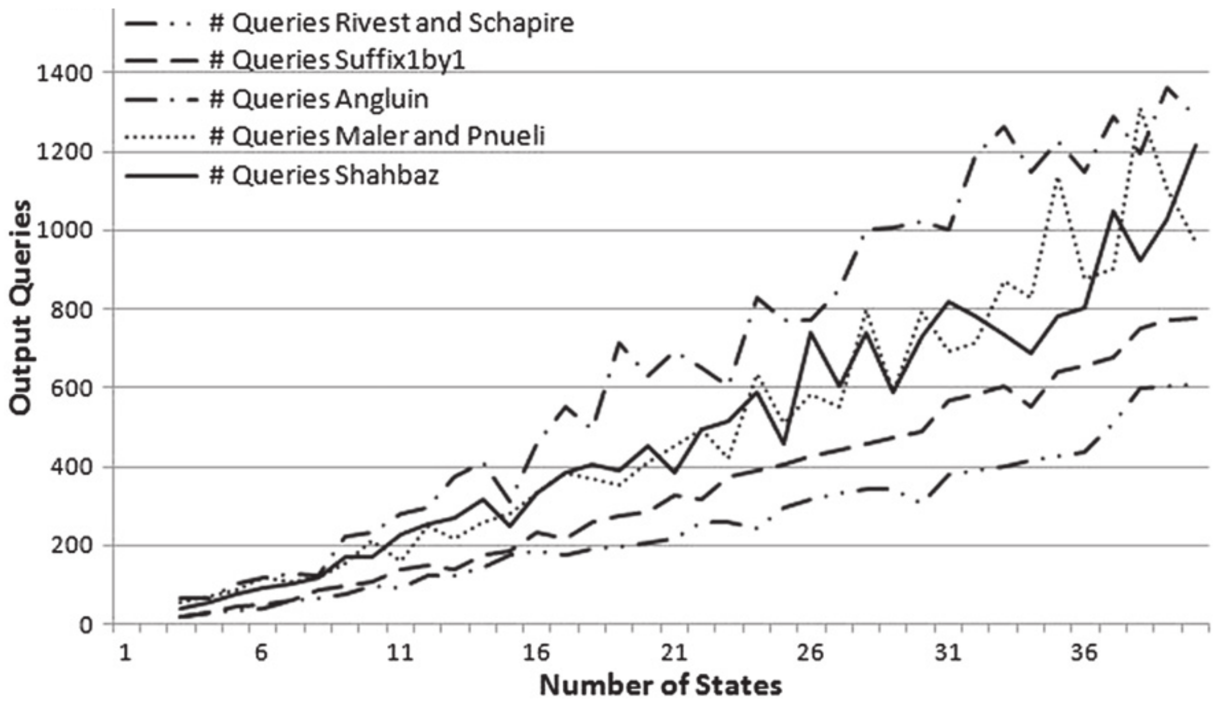


Figure 14 – Comparison of counterexample processing methods

Source: Irfan, Oriat and Groz (2013).

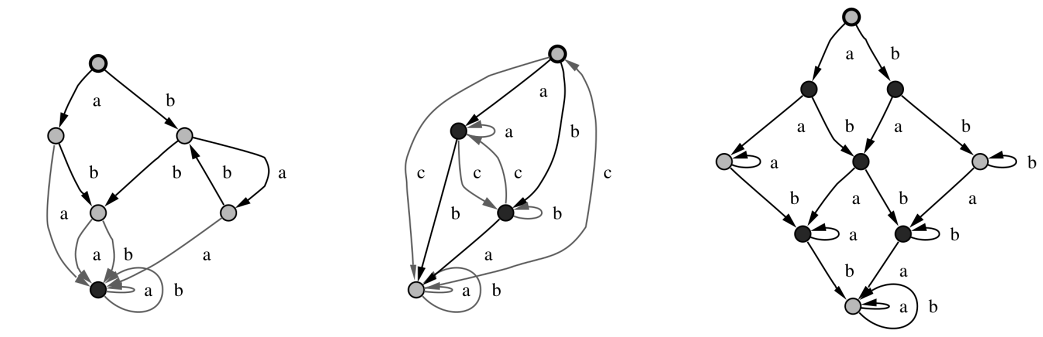
3.2.4 Possible Optimizations

For large systems, model inference may require an impractically large number of queries, or several seconds, if not minutes, to perform tests. Thus, an intelligent reuse of queries is required to make model inference more feasible to real-world problems. To tackle these issues, domain-specific knowledge has been used to reduce the number of required queries, verify evolvable systems, and avoid reset operations.

3.2.4.1 Filters

Margaria, Raffelt and Steffen (2005) present an experimental analysis comparing four different filters to reduce the number of required queries. The following filters were investigated and combined in ten different ways: Redundancy (*C-filter*); Prefix-closure (*P-filter*); Independence of actions (*I-filter*); and Symmetry (*S-filter*). Figure 15 depicts the P, I and S filters as DFAs with accepting states in gray.

Figure 15 – Filters as DFAs with different characteristics

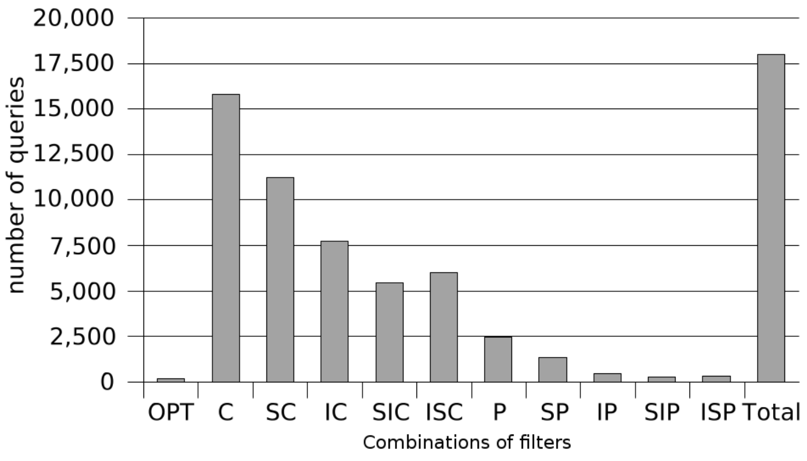


(a) Prefix-closure filter (b) Independent actions {a,b} (c) Symmetric actions {a,b}

Source: Margaria, Raffelt and Steffen (2005).

The *C-filter* prevents redundant queries by caching results of queries in a hash table and reports inputs that are part or not of a language or were never asked before. The *P-filter* avoids inputs that are prefix of queries previously asked. The *I-filter* filters out queries iff there is a reordering of the inputs conforming to independence relations specifying which events can be shuffled in any order. The *S-filter* identifies actors (e.g., processors, memories, phones) appearing in a particular query and frees them whenever they reach an initial state again. Figure 16 shows the number of queries for ten different combinations of the aforementioned filters.

Figure 16 – Comparative filter efficiency by number of queries



Source: Margaria, Raffelt and Steffen (2005).

The scenarios are identified by the combination of the letters *C*, *P*, *I*, and *S*, *OPT* denotes the optimal case, and *Total* presents total number of required queries. The results show that filtering queries can be an extremely powerful approach for the practicality of model inference. Additionally, *P-filter* was more efficient at eliminating redundant queries than *C-filter*, even combined with other filters. In particular, the combination of the *S*, *I*, and *P* filters lead to a reduction in the learning effort close to the optimal (*OPT*) filter. At last, it was detected that permutations on the filters (i.e., SIC vs. ISC) can affect the efficiency.

3.2.4.2 Evolving Systems

Chaki *et al.* (2008) proposed an approach to the substitutability problem for evolving software systems. The substitutability problem consists on verifying if (i) any updated part of an SUT still provides all services offered by its earlier versions, and (ii) previously satisfied properties remain valid for newer versions. Central to their approach is the *dynamic* L^*, a novel dynamic algorithm to learn appropriate models for new components by reusing the models of older versions.

Originally, the L^* algorithm starts with $S_M = E_M = \{\varepsilon\}$ and this can be a drawback when there is a need for restart learning and previously inferred models (and hence an OT) exist. The *dynamic* L^* algorithm is able to terminate with a correct model given any valid OT. The previous *OT* is revalidated and a subset of S_M and E_M is reused to focus only on updated parts. An experiment showed that the dynamic approach required 20% the effort of a complete re-validation of the SUL after an upgrade (CHAKI; SHARYGINA; SINHA, 2004).

3.2.4.3 Inferring FSMs Without Reset

Recently, Groz *et al.* (2015) proposed a method for inferring models of non-resettable, minimal and strongly connected mealy machines. Traditional inference algorithms assume that the SUL can be reliably reset to an initial state, making inference easier, as each new experiment can start from a known state, and examine neighboring states. However, sometimes the reset operation can be inexistent, unreliable, or require a lot of time (e.g., rebooting a system with many components to reconfigure and/or reinitialize). The method relies on an upper bound n on the number of states, a characterizing set W, and a localizer procedure $L(\omega, W)$ to ensure that learning continues from states visited before.

The key idea is that since the number of states is bounded by n, by repeatedly applying an input sequence α we can observe at most n different output sequences. Thus, in the state reached by applying n times α, the SUL must have reached a cycle, coming to one of the states visited. Experiments with hundreds of random models show that for $n = 15$, $|I| = 10$, inference is achieved in around 500 times (i.e., 3 orders of magnitude) less steps than its theoretical bound.

3.3 Examples of Application

MBT relies on explicit test models of SUTs and/or its environment to automatically derive test cases (UTTING; PRETSCHNER; LEGEARD, 2012). Unfortunately, building and maintaining test models is time-consuming and dependent on how familiar test engineers are with the SUT (BERG; RAFFELT, 2005). To address this issue, researchers have proposed MBT from inferred models. In this section, we present three examples of this approach.

3.3.1 MBT from ripped models

One of the primary requirements for MBT is the availability of explicit test models specifying the behavior of the SUT, e.g., *input-only*, or expected *input-output* domains (UTTING; PRETSCHNER; LEGEARD, 2012). However, in practice, test models are often unavailable, lack completeness, consistency or are outdated and this may hinder the applicability of MBT (MARIANI; PEZZÈ; ZUDDAS, 2015).

To overcome these problems, researchers have proposed the MBT from ripped models. The MBT from ripped models works by first extracting test models of SUTs using static or dynamic analysis and then applying standard MBT at the hypothesized test model (MARIANI; PEZZÈ; ZUDDAS, 2015). Ripped models are not real specifications but derived oracles as they describe system traces rather than the intended behavior. Thus, they are not suitable to test unimplemented features.

MBT from ripped model has been used for security testing (RUITER; POLL, 2015), verifying evolvable systems (CHAKI; SHARYGINA; SINHA, 2004; CHAKI *et al.*, 2008), integration testing (LI; GROZ; SHAHBAZ, 2006; SHAHBAZ; PARREAUX; KLAY, 2007; SHAHBAZ; LI; GROZ, 2007; SHAHBAZ; GROZ, 2014), regression testing (HAGERER *et al.*, 2002; HUNGAR; NIESE; STEFFEN, 2003), performance evaluation (NEUBAUER *et al.*, 2012; ADAMIS; KOVÁCS; RÉTHY, 2015), and GUI testing (CHOI; NECULA; SEN, 2013).

3.3.2 Learning-based testing

Real systems are often too big to be completely inferred and tested (MEINKE; SINDHU, 2011). To tackle this issue, Meinke and Sindhu (2011) proposed learning-based testing (LBT), an iterative approach that aims at automatically generating a large number of test cases by combining a model checking algorithm with optimized model inference algorithms.

Previous works have combined model inference and model checking for testing and verifying systems (PELED; VARDI; YANNAKAKIS, 1999; GROCE; PELED; YANNAKAKIS, 2002; RAFFELT *et al.*, 2009). LBT distinguishes from previous approaches by emphasizing testing rather than verification, and by using iterative learning algorithms specifically tailored to make testing more effective and scalable (MEINKE; SINDHU, 2011).

The general process of LBT is as follows: (1) *run an initial test suite* on the SUL; (2) *refine hypothesis* based on the resulting behavior of the SUL; (3) *model check the hypothesis* using temporal requirements (e.g., LTL) and a model checking tool (e.g., NuSMV) where counterexamples are used as test cases (i.e., tests as queries); (4) *Tests are executed* until either faults are revealed or a correct hypothesis is found. If more time is available, retake step (2).

LBT has been used for unit testing of numerical software and significantly outperformed random testing (MEINKE; NIU, 2010). For reactive systems, the iterative nature of LBT has also enabled a more scalable and efficient testing compared to complete learning (i.e., MBT from ripped models) (MEINKE; SINDHU, 2011). Meinke and Nycander (2015) evaluated and confirmed the viability of LBT for testing distributed microservice architectures. Meinke (2017) showed that LBT can find faults of cyber-physical systems by learning large but incomplete models. Khosrowjerdi, Meinke and Rasmusson (2017) evaluated the feasibility and effectiveness of LBT for testing automotive systems and found that simple propositional LTL formulae are quite viable for modeling behavioral requirements and that the high throughput of LBT is well suited to build large black-box test suites on-the-fly. A list of case studies in LBT can be found in (FENG *et al.*, 2013).

3.3.3 Assertion Violations and Failure Models

Testing with model checking allows to determine if properties hold for a given SUT and take counterexamples as test cases (FRASER; WOTAWA; AMMANN, 2009). However, counterexamples are often useless to guide programmers in software analysis (CHAPMAN *et al.*, 2015). To overcome this problem, model inference has been used to describe failures.

Chapman *et al.* (2015) proposed a learning scheme to describe possible scenarios under which assertion violations occur. Their learning scheme is based on code instrumentation and user-defined events describing the behaviors (e.g., entry to functions and branches) that lead to an assertion violation. Events are identified by instrumenting the code with two instructions: $Learn(uid)$, at a desired position, where *uid* is a unique identifier; and *Learn_Assert*, at the location of the assertion that is being investigated. The set of identifiers constitutes the alphabet of the model they construct. Beyond assertion violations, their approach can be used in program explanation and for merging branches in version control systems.

Kunze *et al.* (2016) designed a similar approach to build *generalized failure models*. These models start from a single failing test case and produce a model which describes under what other circumstances (e.g., other interaction scenarios with possibly different parameters) similar failures can be observed.

3.4 Tooling Support

A major challenge of model inference is to perform this task in a rigorous, scalable and efficient manner. To that, there is a need for providing researchers and practitioners with reusable components and tools. In this section, we summarize five tools that support model inference of behavioral models.

3.4.1 *libalf: The Automata Learning Framework*

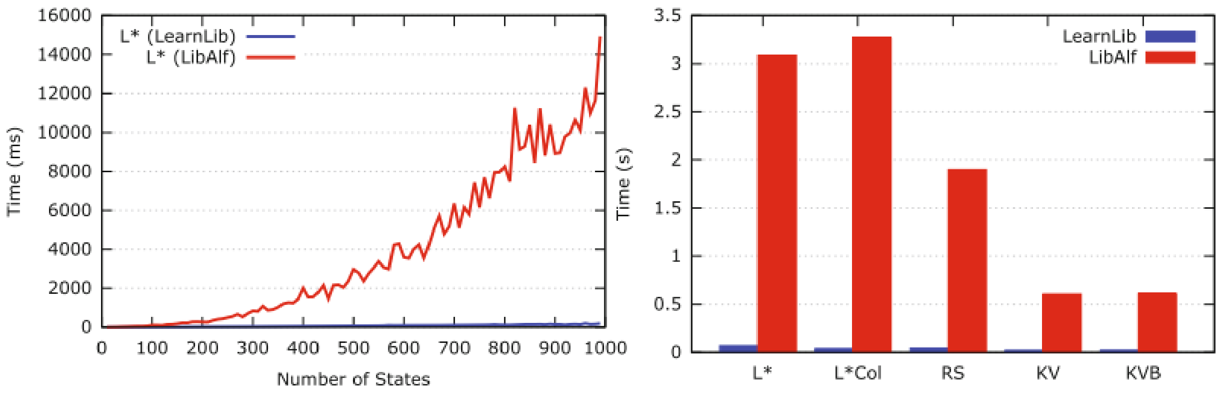
The libalf automata learning framework was a comprehensive, open-source automata learning library developed by RWTH Aachen University (BOLLIG *et al.*, 2010). It supported various inference algorithms for deterministic and nondeterministic finite automata and was written in C++. The last version was released in April 2011 (libalf, 2017).

3.4.2 *LearnLib: a framework for automata learning*

The LearnLib tool (LearnLib, 2017) is an open-source project developed in Java to support automata learning in practical settings, and to develop and analyze automata learning algorithms (RAFFELT; STEFFEN, 2006; ISBERNER; STEFFEN; HOWAR, 2015). It provides the majority of learning algorithms for DFA and mealy machines which have been published in a modular, extensible and parameterized fashion (ISBERNER; HOWAR; STEFFEN, 2015).

Experiments using randomly generated automata and the classic L^* algorithm have shown that LearnLib outperforms the libalf tool in more than an order of magnitude. Figure 17 shows some of the results comparing LearnLib and libalf with random automata with states counts between 10 and 1000 and five comparable algorithms on a DFA with 500 states.

Figure 17 – Performance comparison between LearnLib and libalf



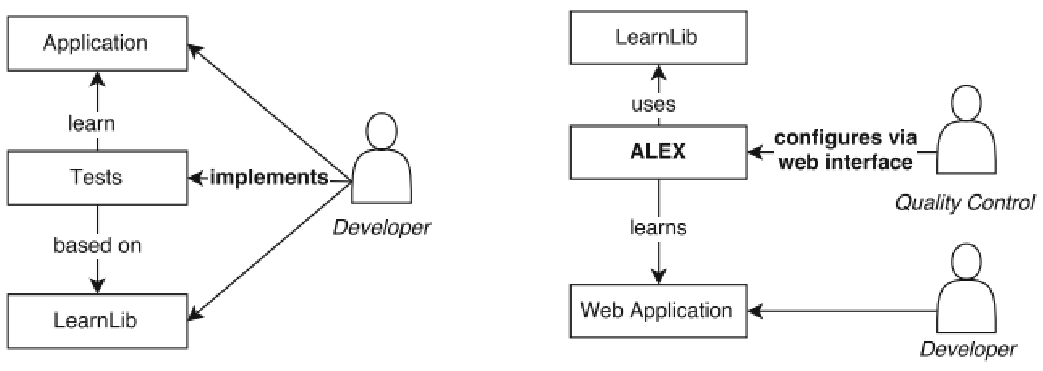
Source: LearnLib (2017).

The LearnLib tool provides a graphical user interface (GUI) to setup learning scenarios as executable process models and domain-specific libraries in a service-oriented manner (BAUER; NEUBAUER; ISBERNER, 2016). In 2010, the LearnLib tool won the 2010 Zulu active learning from queries competition (HOWAR; STEFFEN; MERTEN, 2010).

3.4.3 ALEX: Automata Learning EXperience

The Automata Learning EXperience (ALEX) tool (ALEX, 2017) is a recent project developed by TU Dortmund University in cooperation with the Irish Software Research Center (LERO). It is a web application that enables non-programmers to use the LearnLib tool functionalities via a RESTful API for inferring models of web applications (BAINCZYK *et al.*, 2016). Figure 18 depicts two ways to setup an environment for inferring models of web applications.

Figure 18 – Setting up a model learning environment



(a) Manual scenario with LearnLib (b) ALEX: Automata Learning EXperience

Source: Bainczyk *et al.* (2016).

In the first case, depicted in Figure 18a, there is a need for a significant technical knowledge to define the semantics of the learning alphabet and writing Java code. In the second case, depicted in Figure 18b, non-programmers can setup learning scenarios in a guided way and reduce the cost of writing code to intermediate the communication between the SUL and the LearnLib tool.

3.4.4 RALib: A LearnLib extension for inferring EFSMs

The RALib (RALib, 2017) is another extension to the LearnLib framework for active learning of register automata. A register automata (RA) is a class of extended finite state machine (EFSM) with a finite control structure, extended with variables (registers), assignments, and guards (CASSEL FALK HOWAR, 2015; CASSEL *et al.*, 2016). The RALib tool has been applied to TCP implementations of different operating systems and has found specification violations, further confirmed by Linux developers (FITERĂU-BROŞTEAN; HOWAR, 2017).

3.4.5 Tomte Tool

The Tomte tool (Tomte, 2017), developed at the Radboud University of Nijmegen, uses counterexample-guided abstraction refinement (CEGAR) to automatically generate a mapper and learn a specific class of RA called scalarset Mealy machines, in which one can test for equality of data parameters, but no operations on data are allowed (AARTS *et al.*, 2012). This mapper is placed between the MAT (i.e., LearnLib) and the SUL to associate actions of the MAT into a small set of abstract actions. Thus, it can effectively learn a (symbolically represented) state machine that is equivalent to the SUL.

By using tomte, Aarts *et al.* (2012) were able to automatically learn models of realistic software components, such as a biometric passport and the Session Initiation Protocol (SIP). The performance of RALib and Tomte is comparable, but superior to LearnLib. In some cases, RALib outperforms Tomte, but there are some RA that RALib cannot handle, such as a FIFO-set, i.e., a queue that only stores different values, with capacity 40 (VAANDRAGER, 2017).

3.5 Final remarks

Model inference is an active learning procedure which aims at extracting the behavioral model of an SUL by pursuing inputs and observing outputs. Researchers have used model inference to support MBT, describe assertion violations, and generalize failure models to describe under what other circumstances similar failures can be observed. There are several tools available for active automata learning, and LearnLib has been part of most of these investigations. In the next chapter, we discuss the main concepts of software product lines and the challenges of specifying families of products of SPLs.

CHAPTER

4

SOFTWARE PRODUCT LINE

Large technology companies such as ABB, Boeing, Philips, and Siemens have confronted with an increasing demand for mass production and customization of hardware and software products (POHL; BÖCKLE; LINDEN, 2005). To cope with this, these companies have invested in establishing common platforms to build families of software products efficiently. These platforms are named software product lines.

Software product line (SPL) is a family of software products sharing a common and managed set of assets that satisfy the specific needs of a particular market segment, and that is developed in a prescribed way (CLEMENTS; NORTHROP, 2001). Unlike traditional systems which are independent and self-contained, SPLs are developed for reuse and with reuse (LINDEN; SCHMID; ROMMES, 2007). To achieve these goals, commonalities, and variabilities support the development of reusable assets and derivation of families of products.

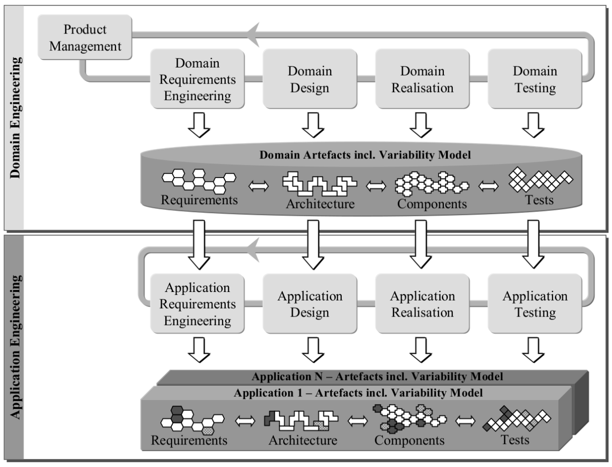
Software product line engineering (SPLE) is a framework to develop families of products using common platforms and mass customization (POHL; BÖCKLE; LINDEN, 2005). Due to the systematic reuse, the cost and time-to-market of SPLs tend to decrease, while the quality of individual products increases (OSTER *et al.*, 2011a). However, SPLE challenges traditional analysis techniques, such as MBT, as they have to guarantee that reused assets work as intended, regardless combinations. Moreover, exhaustively analyzing families of products is usually not feasible, due to the potentially exponential number of valid products and redundant computations performed on artifacts, such as test models and source-code (THÜM *et al.*, 2014a).

In this chapter, we introduce the main concepts of SPLE, domain engineering, application engineering, and the Feature-Oriented Domain Analysis (FODA), a method to support variability modeling. Afterwards, we discuss the two extremes of analysis strategies for SPL: Product-based analysis and family-based analysis. Examples of notations for family-based analysis and tooling support to implement SPLs are also presented.

4.1 Software Product Line Engineering Framework

According to Pohl, Böckle and Linden (2005), the SPLE framework aims at supporting the development of multiple software applications (called *products*) from a common set of assets in a systematic way. Products are differentiated by the features they implement, where features consist of increments to their functionalities (BENAVIDES; SEGURA; RUIZ-CORTÉS, 2010). The SPLE framework is depicted in Figure 19.

Figure 19 – The Software Product Line Framework



Source: Pohl, Böckle and Linden (2005).

The SPLE framework is composed of two processes (i.e., domain engineering and application engineering). Each process contains a set of iterative activities, as in traditional software engineering (i.e., requirements specification, design, implementation and testing), and encompass different kinds of software artifacts (e.g., requirements specifications, design models, software components, and test artifacts). This separation of concerns enables to build robust platforms to develop customer-specific applications in a shorter time, at lower cost, and with improved quality. The activities and resulting artifacts of domain engineering and application engineering are discussed in the next sections.

4.1.1 Domain Engineering

The domain engineering is the process where the common and variable artifacts (i.e., commonality and variability), and the scope of an SPL are defined, managed and constructed. In domain engineering, there are five activities: Product management, domain requirements engineering, domain design, domain realization, and domain testing. Each one of these activities contributes to the application engineering process by generating specific sets of reusable artifacts. In Table 12, we present the goals and main resulting artifacts of the activities from domain engineering.

Table 12 – Activities of the domain engineering

Activity	Aims at	Main resulting artifacts
Product management	Economic aspects, marketing strategies, and the scope of an SPL	Product roadmap and major common and variable features
Domain requirements engineering	Elicitation and documentation of common and variable requirements	Variability model, and textual and/or model-based requirements
Domain design	Defines a common, high-level structure for all products of an SPL	Core architecture, internal variability model, and list of reusable artifacts
Domain realisation	Design and implementation of reusable software components	Loosely coupled and configurable components
Domain testing	Verification and validation of reusable components	Test results from domain-level artifacts and reusable test artefacts

Source: Adapted from Pohl, Böckle and Linden (2005).

Among the outputs of the activities depicted in Table 12, the variability model is one of the most important artifacts of domain engineering.

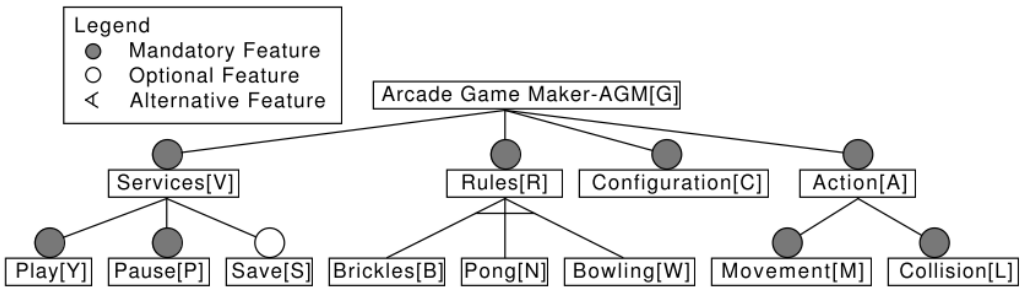
4.1.1.1 *Variability Modeling*

Variability models provide an overview of the capabilities of an SPL (BENAVIDES; SEGURA; RUIZ-CORTÉS, 2010). The feature model (or feature diagram) is one of the most widely used artifacts for variability modeling (SCHAEFER *et al.*, 2012). The term feature model was coined by Kang *et al.* (1990) in the Feature-Oriented Domain Analysis (FODA) technical report. There are different feature modeling languages, and for a detailed survey on feature modeling languages, we refer to Schobbens *et al.* (2007).

Feature-Oriented Domain Analysis (FODA)

A feature model captures all information about common and variant features of an SPL as a hierarchically arranged set of features interconnected by different kinds of relationships. Figure 20 depicts a simplified feature model named Arcade Game Maker (AGM).

Figure 20 – An example of feature model



Source: Fragal, Simao and Mousavi (2017).

Features are interconnected by four kinds of basic relationships: *Mandatory*, if a child feature is included in all products in which its parent appears; *Optional*, if a child can be optionally included; *Alternative*, when only one child feature can be selected; and *Or*, when one or more of features can be included in the products. Additionally, feature models can also present cross-tree constraints between features. These are two kinds of cross-tree constraints: *Requires*, if a feature A requires a feature B; and *Excludes*, if a feature A excludes a feature B.

Example: In Figure 20, it is *mandatory* to every AGM to support play (Y) and pause (P) services (S), configuration (C) and the actions (A) of movement (M) and collision (L) detection. At last, the following Rules (R) are alternatively supported: the Brickles game (B), the poNg game (N), and the boWling game (W). In this case, there are six possible products that can be derived from the AGM feature model.

Propositional logic formulae and constraint programming rules can be used to represent and automate the analysis of feature models (BENAVIDES; SEGURA; RUIZ-CORTÉS, 2010). Thus, it can detect invalid feature models and product configurations, dead features, redundancies, and enumerate and quantify all valid products of an SPL.

Besides the aforementioned relationships, recent studies have proposed that feature models have to include arbitrary propositional formulae as constraints (KNÜPPEL *et al.*, 2017). Large feature models, such as the Linux kernel (BERGER *et al.*, 2013), give evidence that requires and excludes constraints are not sufficient to express real-world feature models.

4.1.2 Application Engineering

The application engineering is the process where commonalities and variabilities of an SPL are exploited to achieve the highest possible reuse of domain artifacts. Artifacts generated at the domain engineering are used to support the development of products. Thus, most of the application artifacts are not developed anew but reused from domain engineering. In Table 13, we present the goals and main resulting artifacts of the activities from application engineering.

Table 13 – Activities of the application engineering

Activity	Aims at	Main resulting artifacts
Application requirements engineering	Detecting deltas between application and domain requirements and capturing product-specific requirements	Requirements specification of one particular product
Application design	Selecting and configuring the required parts of domain design artifacts and incorporating product-specific adaptations	Product architecture and realization effort for required adaptations (i.e., develop with reuse or from scratch?)
Application realisation	Selecting and configuring reusable software components and developing product-specific artifacts	Running software product, and detailed design artifacts
Application testing	Verification and validation of a product against its specification	Test report (e.g., test coverage) and detected defects

Source: Adapted from Pohl, Böckle and Linden (2005).

Application artifacts are interrelated by traceability links. This connection ensures the correct binding of variability and a consistent evolution of the variability in time (i.e., different versions at different times) and in space (i.e., an artifact in different shapes at the same time).

4.2 Analysis Strategies for SPLs

Quality assurance for SPL is more challenging than to single-systems as it has to guarantee that features work as intended regardless combinations. According to Thüm *et al.* (2014a), there are five categories of specification and analysis strategies for SPLs: product-based, domain-independent, family-wide, feature-based, and family based. Table 14 presents the main characteristics, disadvantages, and challenges of each analysis strategy.

Table 14 – Description of analysis strategies for SPLs

Strategy	Specification characteristics	Disadvantages and Challenges
Product-based	One specification model for every valid product of an SPL	Scales only for small SPLs, and involves redundant effort
Domain-independent	Specification independent but valid across SPLs	Only describes properties common across SPLs
Family-wide	One specification model that holds for all products of an SPL	Cannot express varying behavior common to some but not all products
Feature-based	Specify the behavior of isolated features instead of products	There is no explicit reference to other features
Family-based	Specify properties of individual features and feature combinations	Traditional methods cannot be used as they are, maintenance of artifacts

Source: Adapted from Thüm *et al.* (2014a).

The two extremes of these specification strategies, i.e., product-based, and family-based analysis, are discussed in the next sections.

4.2.1 Product-Based Analysis

In a product-based analysis, specifications of all products of SPLs are generated and analyzed individually using standard techniques. A product-based analysis strategy is *optimized*, or *sample-based*, if it takes a subset of all products; or *unoptimized*, if it analyzes all products in an exhaustive (i.e., brute-force, feature-oblivious) way.

Although theoretically possible, it is infeasible due to the often exponential number of valid products of an SPL, and inefficient due to redundant computations over parts of specifications shared among products. Moreover, the product-based analysis refers to generated application artifacts, but to domain artifacts. Thus, changes to application artifacts are not automatically shared among themselves. The unoptimized product-based analysis serves as a baseline for other strategies regarding soundness, completeness, and efficiency (THÜM *et al.*, 2014a).

4.2.2 Family-Based Analysis

Family-based analysis operates on domain artifacts and incorporates knowledge about valid feature combinations, given a feature model. Thus, not every individual product has to be analyzed, and redundant computations are avoided. As an advantage, the effort of family-based analysis strategies is mainly influenced by the number and size of features and the amount of sharing during combinations, while to product-based strategies it is proportional to the number of valid feature combinations (BRABRAND *et al.*, 2012).

Schaefer *et al.* (2012) categorize family-based specifications in three groups: (i) *compositional*, when product variants are described through the combination of model fragments; (ii) *annotative*, when a single *family model* (or *150% model*) is annotated with presence conditions to represent families of products; and (iii) *transformational*, where products of an SPL are represented as a core model and a set of deltas specifying changes to the core model.

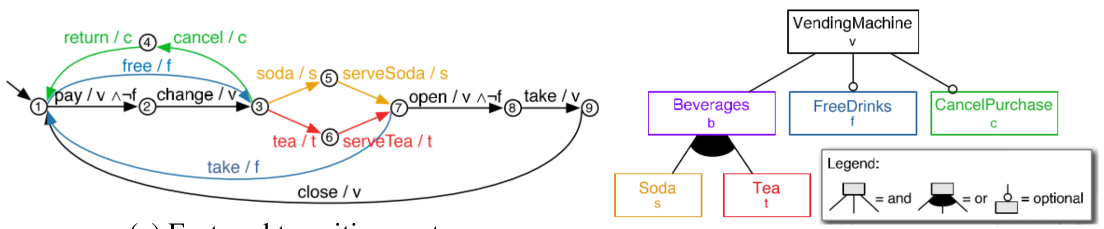
Compositional approaches improve modularity by encapsulating individual features into separate modules (BENDUHN *et al.*, 2015), but lack expressiveness for crosscutting features and removal of behavior (SCHAEFER *et al.*, 2010). Annotative approaches enable the fine-grained representation of variability among products (e.g., crosscutting features), but suffer scalability issues (SCHAEFER *et al.*, 2010) and can become too complex to be handled efficiently, and hard to be maintained. Transformational approaches can take advantage of both compositional and annotative models (SABOURI; KHOSRAVI, 2013), but may require conflict-resolving rules for every pair of conflicting deltas (CLARKE; HELVENSTEIJN; SCHAEFER, 2010).

Three examples of family-based notations are presented in the following sections: Featured transition systems (FTS), Featured finite state machines (FFSM), and delta modeling.

4.2.2.1 Featured Transition Systems (FTS)

The featured transition system (FTS) model was proposed by Classen *et al.* (2013) as a concise mathematical model for representing the behavior of families of products. It is an annotative notation where a transition system is extended with presence conditions based on a feature model. Figure 21 depicts an example of FTS of a vending machine.

Figure 21 – Example of FTS - Vending machine SPL



(a) Featured transition system

(b) Feature model

Source: Classen *et al.* (2013).

Formally, an FTS is a tuple $\langle \mathrm{S}, \mathrm{Act}, \mathrm{trans}, \mathrm{I}, d, \gamma \rangle$ where S is a finite set of states, Act a finite set of actions, *trans* $\subseteq \mathrm{S} \times \mathrm{Act} \times \mathrm{S}$ is the transition relation with $(s1, \alpha, s2) \in \mathrm{trans}$, $\mathrm{I} \in \mathrm{S}$ is a set of initial states, d is a feature model, and $\gamma : \mathrm{trans} \rightarrow \mathbb{B}(N)$ is a function labeling each transition with a boolean feature expression (i.e., boolean formula $\mathbb{B}$ over a set of features N, whose solutions represent a set of products that a transition is present).

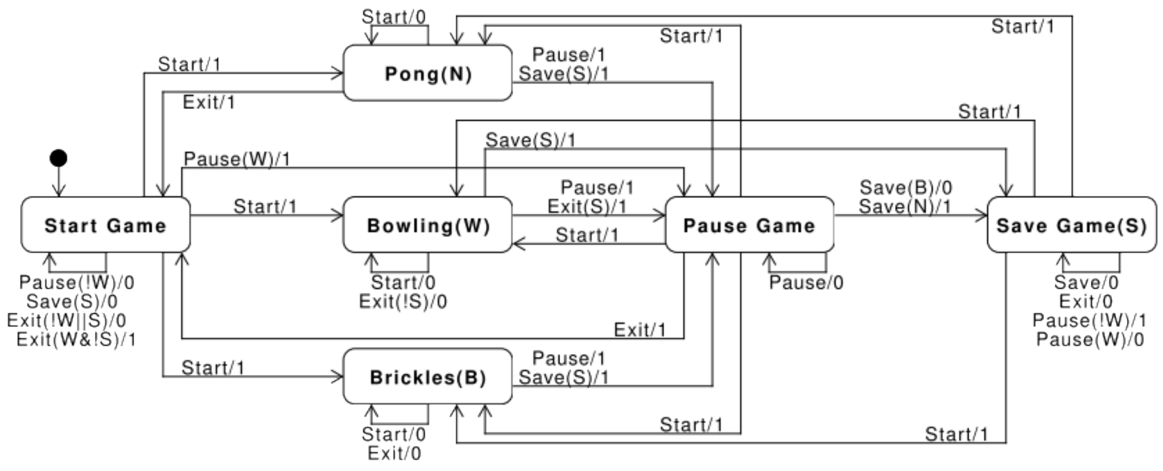
Recently, Beohar and Mousavi (2016) presented the FTS model as the most expressive formalism compared to other popular family-based notations, such as modal transition systems (MTS) (LARSEN; THOMSEN, 1988), and product line calculus of communicating systems (PL-CSS) (GRULER; LEUCKER; SCHEIDEMANN, 2008).

Several extensions to the FTS model have been proposed, such as weighted FTS (WFTS) to support the analysis of performance and energy consumption (OLAECHEA *et al.*, 2016), *feature merge expressions* (FME) to enable the modular specification of FTS models (ATLEE *et al.*, 2015), and and Input-Output FTS (IOFTS) to enable the input-output conformance testing (IOCO) of families of products (BEOHAR; MOUSAVI, 2016).

4.2.2.2 Featured Finite State Machines (FFSM)

Fragal, Simao and Mousavi (2017) proposed a specification language named Featured Finite State Machine (FFSM). An FFSM is an extended Mealy machine model where states and transitions are annotated with feature constraints to denote presence conditions given a feature model. Figure 22 depicts an example of FFSM for the Arcade Game Maker (AGM) SPL presented in Figure 20.

Figure 22 – Example of FFSM - AGM SPL



Source: Fragal, Simao and Mousavi (2017).

Formally, an FFSM is a 7-tuple $\langle F, \Lambda, C, c_0, Y, O, \Gamma \rangle$, where: **(1)** F is a finite set of features, **(2)** Λ is the set of product configurations, **(3)** $C \subseteq S \times B(F)$ is a finite set of conditional states, where S is a finite set of state labels, $B(F)$ is the set of all feature constraints, and C satisfies the following condition:

$$\forall (s, \phi) \in C,\ \exists \rho \in \Lambda, \text{ such that } \rho \models \phi \tag{4.1}$$

(4) $c_0 = (s_0, true) \in C$ is the initial conditional state, **(5)** $Y \subseteq I \times B(F)$ is a finite set of conditional inputs, where I is the set of input labels, **(6)** O is a finite set of outputs, and **(7)** $\Gamma \subseteq C \times Y \times O \times C$ is the set of conditional transitions satisfying the following condition:

$$\forall ((s, \phi), (x, \phi''), o, (s', \phi')) \in \Gamma,\ \exists \rho \in \Gamma, \text{ such that } \rho \models (\phi \wedge \phi' \wedge \phi'') \tag{4.2}$$

As a first step to support FSM-based testing of families of products, Fragal, Simao and Mousavi (2017) proposed a family-based validation criterion for FFSM models and proved that an FFSM coincides with all test models derivable from it. The proposed validation criterion was implemented using Java and Z3 (MOURA; BJØRNER, 2008) and evaluated in a large set of synthetic models and in the Body Comfort System (BCS) SPL (LITY *et al.*, 2013).

Specification languages as FTS and FFSM are useful for *family-based testing*, where families of products are checked in a single run whether they satisfy a family model (CICHOS *et al.*, 2011).

4.2.2.3 Delta modeling

Delta modeling (or Δ-modeling) is a variability modeling approach orthogonal to model refinement (SCHAEFER, 2010). In delta modeling, families of products are represented by a core model, specifying a valid product of an SPL, and a set of delta models denoting changes (i.e., additions, modifications and removals of model fragments) to the core model.

The delta modeling concept has been applied to several artifacts, such as UML component and class diagrams (SCHAEFER, 2010), behavioral specifications in Rebeca language (SIRJANI; JAGHOORI, 2011), an actor-based language for modeling concurrent and distributed systems (SABOURI; KHOSRAVI, 2013), FSM models (VARSHOSAZ; BEOHAR; MOUSAVI, 2015), MontiArc (HABER *et al.*, 2011), programs in java (SCHAEFER *et al.*, 2010), and Matlab/Simulink models (HABER *et al.*, 2013b). Recently, Haber *et al.* (2013a) proposed a systematic approach for deriving delta languages from a given base language.

Regardless the benefits of delta modeling, the additional expressiveness gained by using delta modules may cause problems on software maintenance and program comprehension (APEL *et al.*, 2013a). Thus, it may be harder to take an overview whether delta modules will delete an element that is needed.

4.3 Tooling Support

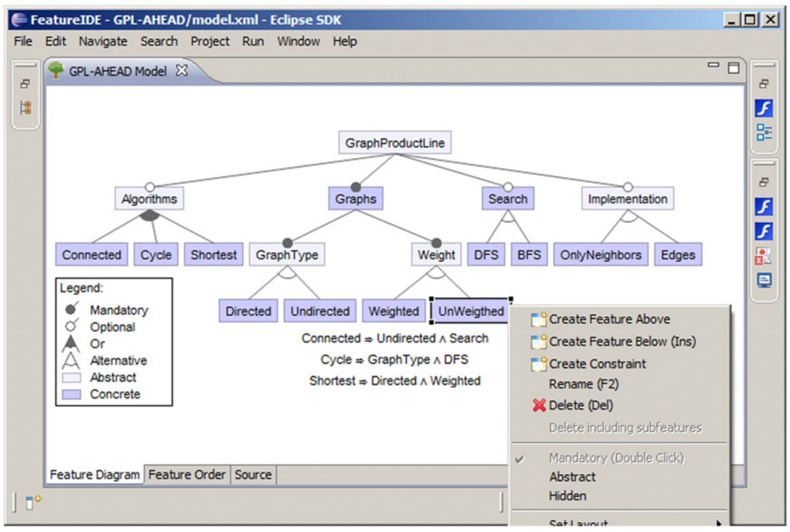
In this section, we present examples of tools to support the implementation of SPLs. The first aims at supporting feature-oriented software development (FOSD) (APEL *et al.*, 2013b). The latter consists of a java dialect for delta-oriented programming (SCHAEFER *et al.*, 2010).

4.3.1 FeatureIDE: An Eclipse plug-in for FOSD

Feature-oriented software development (FOSD) is a compositional strategy for building SPLs that relies directly on features. In FOSD, the structure of a system is decomposed into the features it provides with, ideally, one module or component per feature (APEL *et al.*, 2013a). Thus, there is a direct mapping from features to feature modules that implement them.

FeatureIDE is an extensible framework for FOSD based on eclipse (THÜM *et al.*, 2014b). It supports different SPL implementation techniques, such as feature-oriented programming (FOP), delta-oriented programming (DOP), aspect-oriented programming (AOP), preprocessors, and plug-ins. It also includes a graphical feature model editor with several analysis techniques incorporated. The graphical feature model editor of FeatureIDE is shown in Figure 23.

Figure 23 – The graphical feature model editor of FeatureIDE



Source: Meinicke *et al.* (2017).

Meinicke *et al.* (2017) present a self-contained, practical introduction to FeatureIDE as an integrated development environment (IDE) for implementing variable systems. Recent updates, published papers and the full documentation and source code of FeatureIDE is available online at <https://featureide.github.io/>.

4.3.2 DeltaJava: a delta oriented programming language

In DeltaJava, a delta model specifies which classes, interfaces, methods or fields are added, removed, and modified. They also specify application conditions and conflict-resolving rules to resolve simultaneous changes to similar parts specified by different delta models (CLARKE; HELVENSTEIJN; SCHAEFER, 2010). Figure 24 depicts the syntax of the DeltaJava, a delta modeling approach for java programs.

Figure 24 – The DeltaJava syntax

```
delta <name> [after <delta names>] when <application condition> {
    removes <class or interface name>
    adds class <name> <standard Java class>
    adds interface <name> <standard Java interface>
    modifies interface <name> { <remove, add, rename method header clauses> }
    modifies class <name> { <remove, add, rename field clauses> <remove, add, rename method clauses> }
}
```

Source: Schaefer *et al.* (2010).

In Figure 25, we present an example of program in DeltaJava (VARSHOSAZ; BEOHAR; MOUSAVI, 2015). The *Bridge* can be or not available for crossing depending on the value of the boolean variable `Avl`. Added to this, there are two delta modules: `DPedLight`, which modifies the `Bridge` class by adding a boolean variable `Psig` to denote the status of a pedestrian light; and `DController`, which includes a new class `Controller` to control the status of the lights on both sides of the bridge.

Figure 25 – The Bridge example in DeltaJava

```
Core Bridge{
        Class Bridge{
          private boolean Avl;
          public Bridge() {Avl=true;}
          public void SetAvl(){Avl=true;}
          public void ResetAvl(){Avl=false;}
          public boolean CheckAvl(){return(Avl);}
        }
}
delta DPedLight when pedestrian Light {
        modifies Class Bridge{
          adds boolean Psig
          adds boolean CheckPsig(){return(Psig);}
          adds void SetPsig(){Psig=true;}
          adds void ResetPsig(){Psig=false;}
        }
}
```

```
delta DController when controller {
   adds Class Controller{
     private boolean Lsig,Rsig;
     public bridge b;
     public controller(){
     Lsig=false; Rsig=false;}
     public int CheckLsig(){return(Lsig);}
     public int CheckRsig(){return(Rsig);}
     public void GetReq(int id){
       if(b.CheckAvl()==true){
         if(id==0){
         Lsig=true;Rsig=false;}
       else{
         Rsig=true;Lsig=false;
       }
       b. ResetAvl();}
     }
     public void SetPassed(){
       Lsig=false;Rsig=false;
       b.SetAvl();}
   }
}
```

Source: Varshosaz, Beohar and Mousavi (2015).

4.4 Final Remarks

SPLE is a paradigm for developing families of software products using common platforms and mass customization. Thus, the cost and time-to-market of SPLs tend to decrease, while software quality increases. Since SPLE has to guarantee that a potentially exponential number of valid configurations work as intended, regardless combinations; traditional analysis and exhaustive testing techniques are not suitable or cost-effective for SPL analysis.

The family-based analysis incorporates knowledge about valid feature combinations to avoid redundant computations. Based on a specification of a family of products of an SPL (i.e., family model), MBT can be used to generate test cases at reduced cost and with the same effectiveness of exhaustive testing of families of products. However, the maintenance of family models can become hard to be handled manually. To support the maintenance and evolution of family models of SPLs, automated approaches and Computer-Aided Software Engineering (CASE) tools are required.

Model inference has been applied to single-systems for model checking evolving systems, program comprehension, MBT without test models specified *a priori*, describing assertion violations and generating failure models. But, to the best of our knowledge, there are no studies applying model inference on SPLs for software analysis (e.g., MBT, program comprehension, model checking). Thus, we propose to investigate how model inference can be applied to extract test models from families of products (i.e., family models). In the next chapter, we present the objectives and methods, research questions, work plan and expected outcomes of this project proposal.

Part II

Research Proposal and Methodology

CHAPTER

5

RESEARCH PROPOSAL

In MBT, test models are expressed in formal languages to support automated test generation (UTTING; PRETSCHNER; LEGEARD, 2012). Standard MBT is suitable to SPLs by taking the test models of products exhaustively, but it becomes inefficient due to redundant computations performed over shared assets, and the exponential number of valid products. To tackle this, novel approaches have extended software analysis and testing to the domain level.

Family-based testing aims at generating test cases from family models (THÜM *et al.*, 2014a). Family models (also named *150% model*) use extended traditional notations, such as FSMs, annotated with presence conditions represent a family of products. These notations have been exploited as theoretical foundation to perform more efficient test case generation (BEOHAR; VARSHOSAZ; MOUSAVI, 2016) and model checking (SABOURI; KHOSRAVI, 2013; ter Beek; VINK; WILLEMSE, 2017), automatically generate specifications of individual products (ASIRELLI *et al.*, 2012), support the automatic validation of families of products (FRAGAL; SIMAO; MOUSAVI, 2017), and specify fine-grained differences among product variants (SCHAEFER *et al.*, 2010). Despite these possibilities, family-based testing has shortcomings, such as the required investment for creating and maintaining test models (MLYNARSKI *et al.*, 2012), especially to large SPLs and crosscutting features (OSTER, 2012).

Model inference, also called *model learning*, aims at constructing models describing the actual behavior of software and hardware systems through tests (VAANDRAGER, 2017). Model inference has been widely investigated at the level of single systems for black box model checking (PELED; VARDI; YANNAKAKIS, 1999), detection of feature interactions (SHAHBAZ; PARREAUX; KLAY, 2007), automated test generation (RAFFELT *et al.*, 2009), formal description of failure scenarios (CHAPMAN *et al.*, 2015; KUNZE *et al.*, 2016), and extracting Mealy machines (SHAHBAZ; GROZ, 2009; MERTEN *et al.*, 2012; GROZ *et al.*, 2015). To the best of our knowledge, it has never been investigated for inferring family models of SPLs. Thus, we propose to investigate ways to lift model inference to the family-based level.

This chapter is organized as follows: The materials and methods are discussed in section 5.1. The analysis of results is discussed in section 5.2. The work plan is presented in section 5.3. The expected outcomes are shown in section 5.5.

5.1 Objectives and methods

The main objective of this Ph.D. research project is to investigate how model inference can be lifted to the family-based level and enable an efficient and effective extraction of family models of SPLs. To the best of our knowledge, there are no model inference approaches to support the construction of family models of SLPs. Thus, we propose to investigate six research questions (RQ). The three first are:

RQ1: How can we effectively infer Mealy machines from SPLs?

RQ2: How can we merge Mealy machines to generate FFSMs?

RQ3: How can we efficiently infer FFSMs from products of SPLs?

Family-based testing and analysis are known for avoiding redundant analysis across multiple products (THÜM *et al.*, 2014a). We believe that model inference can be harnessed by exploiting variability, as in FFSMs (FRAGAL; SIMAO; MOUSAVI, 2017), be more efficient than a product-by-product strategy, and ease the application of MBT on SPLs even if family models are unavailable, outdated, or incomplete. In particular, in **RQ1**, we want to adapt the L_M^* algorithm (SHAHBAZ; GROZ, 2009) to effectively infer Mealy machines from families of products of SPLs. Model inference (IRFAN; ORIAT; GROZ, 2013) and testing (BROY *et al.*, 2005) for Mealy machines are well-consolidated research topics, and we believe that we can take advantage of these studies in this research.

After effectively extracting Mealy machines from families of products, we intend to investigate approaches to combine the inferred models in a single FFSM model (FRAGAL; SIMAO; MOUSAVI, 2017). This is where we investigate the **RQ2**. Languages of feature merge expressions (ATLEE *et al.*, 2015) have been proposed to enable the modular specification and combination of featured transition systems (FTS) (CLASSEN *et al.*, 2013) and we believe that adapting these expressions to FFSMs can help the inclusion of new behavior into existing family models. Along a family model inference procedure, states of different products equivalent to each other will have to be merged for representing a broader range of the possible behavior of its family. State merging algorithms have been used for merging pairs of states with similar subsequent behavior (WALKINSHAW, 2013) and we also intend to study how these algorithms can support family model inference.

At last, in **RQ3**, we want to investigate ways to perform family model inference (i.e., FFSMs) in an efficient way. We are considering to reuse models inferred of products from a same

SPL. Filters for caching queries (MARGARIA; RAFFELT; STEFFEN, 2005) and algorithms for inferring models of evolving systems (CHAKI *et al.*, 2008) could be redesigned based on the FFSM theory for making family model inference more efficient (FRAGAL; SIMAO; MOUSAVI, 2017). Configuration prioritization (HENARD *et al.*, 2014) can also be used to prioritize products to achieve t-wise coverage and optimize family model inference.

After investigating family model inference in the setting of Mealy machines and FFSMs, we also want to investigate learning-based testing (LBT), feature interaction detection, and the inference of EFSMs from SPLs. Thus, we define three other RQs, presented bellow:

RQ4: How can we take advantage of LBT for testing SPLs?

RQ5: Can we use family model inference to detect feature interaction problems?

RQ6: How can we perform family model inference in a setting of EFSMs?

LBT applies *tests as queries*, where queries are generated either by model learning algorithms or by model checking inferred models (MEINKE; SINDHU, 2013). Several model checking tools have been adapted to analyze whether families of products satisfy temporal requirements, such as `NuSMV` (CLASSEN *et al.*, 2011), `Rebeca` (SIRJANI; JAGHOORI, 2011) and `mcrl2` (CRANEN *et al.*, 2013), and we plan to use them to investigate the **RQ4**.

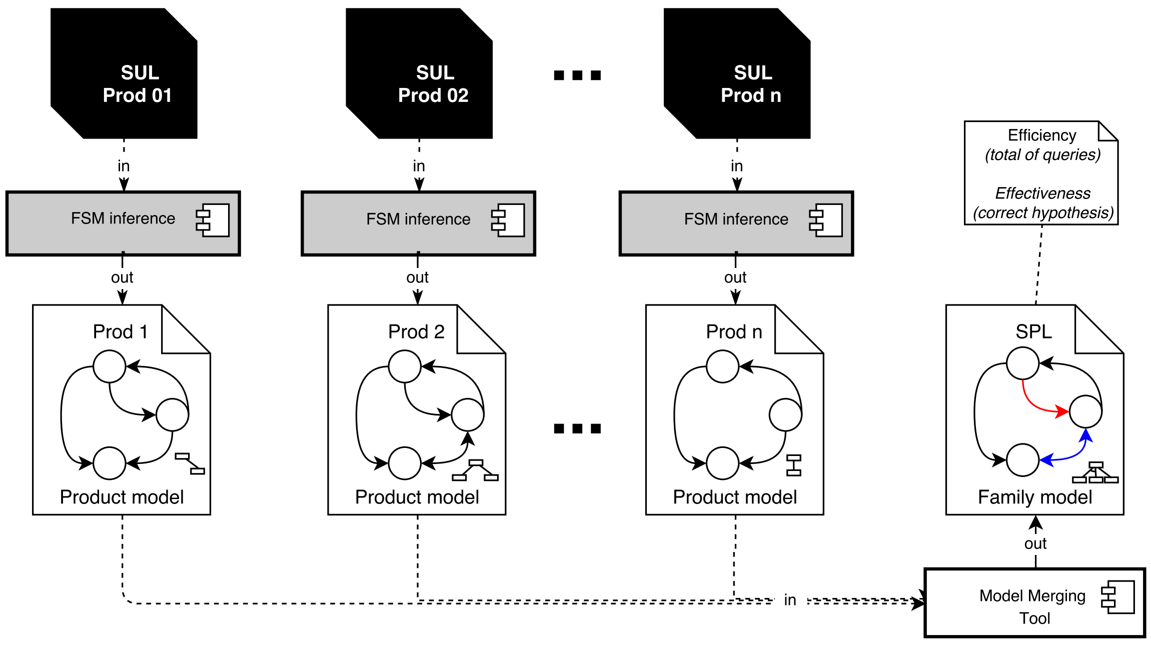
To the **RQ5**, we want to investigate if, by using model checking on inferred family models, we can detect and formally describe scenarios where feature interactions and inadvertent behavioral deviations occur, in a similar way to Chapman *et al.* (2015) and Kunze *et al.* (2016).

At last, to the **RQ6**, we also want to investigate the inference of richer behavioral models, such as EFSMs (PETRENKO; BORODAY; GROZ, 2004), from SPLs. EFSMs have been investigated in the context of MBT (EL-FAKIH *et al.*, 2017) and SPLs (WEIẞLEDER; SOKENOU; SCHLINGLOFF, 2008; GONZALEZ; LUNA, 2008; OSTER *et al.*, 2011b), but as isolated problems. Recently, Cassel *et al.* (2016), Cassel Falk Howar (2015), Aarts *et al.* (2015) investigated approaches for inferring a specific class of EFSMs named register automata (RA). Studies have shown that model inference tools specific for RAs, e.g., RALib (2017), Tomte (2017), outperform traditional model inference tools, such as LearnLib (2017). Thus, we also want to investigate the problem of *inferring EFSMs from SPLs*.

5.2 Analysis of results

In order to evaluate our proposed approaches, we will perform a set of experiments to measure their efficiency and effectiveness. Essentially, we plan to compare our family model inference approach to an exhaustive model inference procedure by taking the *number of queries* as efficiency metric and the *number of correct hypothesis* as the effectiveness. Figures 26 and 27 depict a schematic view of the experiments we designed to evaluate our proposals.

Figure 26 – Evaluation of exhaustive family model inference



Source: Elaborated by the author.

The exhaustive model inference, shown in Figure 26, consists of inferring the behavioral models, that we refer to as *product model*, of each product of an SPL using a standard Mealy inference, such as the L^*_M (SHAHBAZ; GROZ, 2009). Afterwards, we will use a model merging tool to combine the product models into a single *family model*.

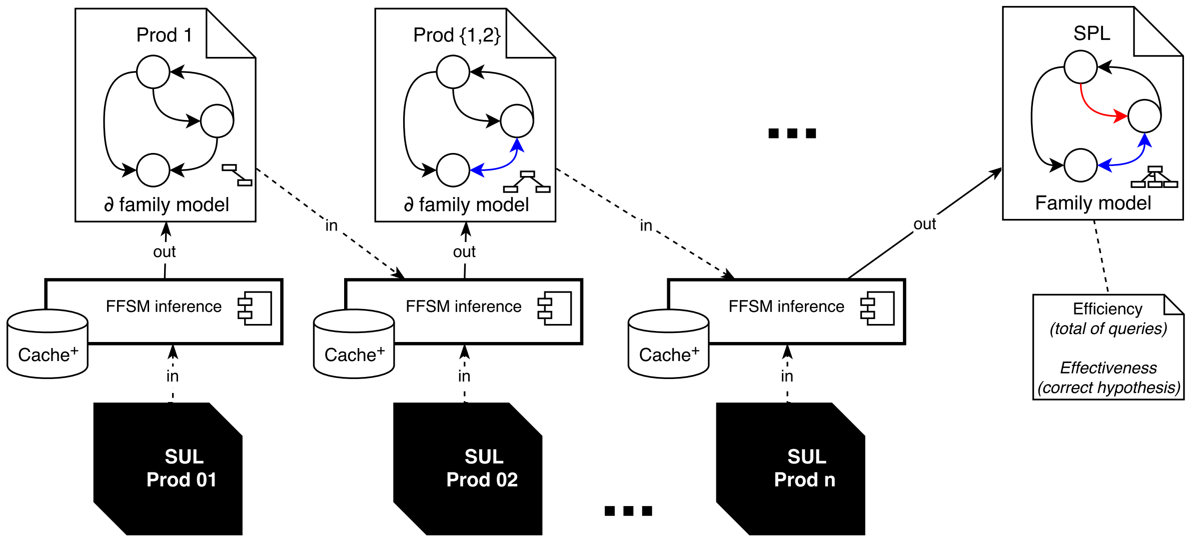
The cost and the effectiveness of inferring product models will be calculated based on the total *number of queries* required to infer the models of a family of products and as the *number of correct hypotheses* (i.e., the FFSM is valid and coincides with all the FSMs derivable from it). The results of the exhaustive model inference procedure will be used as a baseline to the optimized family model inference.

By contrast, we are initially considering to design our family model inference as an incremental procedure, as shown in Figure 27.

All queries performed to infer FSMs of products will be cached for reuse. Filters based on the FFSM theory and previous membership queries will be considered to avoid redundant queries and make family model inference more efficient. Thus, we expect to build an FFSM describing families of products at reduced cost compared to the exhaustive family model inference procedure. We name the intermediate family models as the ∂ `family model`. The cache$^+$ denotes the dataset of queries previously asked and that may potentially filter out unnecessary `MQ` and `EQ`.

We also intend to evaluate the impact of different product configurations orderings. To that, we will use configuration prioritization to find an ordering capable of reducing the queries required to infer a specific FFSM model.

Figure 27 – Evaluation of family model inference



Source: Elaborated by the author.

Configuration selection and reduction (YOO; HARMAN, 2012) are further topics we may investigate depending on the results obtained by using configuration prioritization. A similar experimental setup will be used for investigating approaches for inferring EFSMs from families of products of SPLs.

5.2.1 Experiment artifacts - Tools and SULs

Initially, we plan to use the LearnLib framework (RAFFELT; STEFFEN, 2006) as tool support. The LearnLib is an open-source tool, written in Java that provides a rich set of automata learning algorithms, methods for finding and processing counterexamples, and filters (ISBERNER; HOWAR; STEFFEN, 2015). The RALib (CASSEL FALK HOWAR, 2015) is another tool we pretend to use to support the investigations on EFSM model inference from SPLs.

For equivalence queries and counterexamples, the LearnLib tool supports from random walks to classical conformance testing algorithms, such as W (CHOW, 1978) and Wp (FUJIWARA *et al.*, 1991) methods (ISBERNER; STEFFEN; HOWAR, 2015), specifically tailored for automata learning scenario. It also allows to collect different metrics and implement novel algorithms.

As SULs, we are considering to use (i) random FSM/FFSM/EFSM models (FRAGAL; SIMAO; MOUSAVI, 2017), (ii) the body comfort system (BCS) specification (LITY *et al.*, 2013), (iii) examples from the automotive domain that our collaborators have access, (iv) SPLs from the SPL2go catalog (OvGU, 2011), and (v) other SPLs we eventually find during our studies.

5.3 Work plan

The work plan of this Ph.D. research project started at *23rd may 2016*, date of enrollment, and is bounded to *25th jan 2021*, the limit date to deposit the thesis. Table 15 shows our work plan.

1. Literature review;
2. Studies on model inference, LBT, FFSM, EFSM, and tools;
3. Writing the monograph to the qualification exam;
4. Proficiency exam in the English language;
5. Evaluation of techniques to support family model inference;
6. Design and evaluation of novel model inference and LBT approaches for SPLs
7. Research internship abroad
8. Experiment design
9. Experiment execution
10. Ph.D. coursework requirements
11. Diffusion of results; and
12. Writing and defense of the thesis

Table 15 – Activities schedule

	2016	2017				2018				2019				2020			
Activ.	05-12	01-03	04-06	07-09	10-12	01-03	04-06	07-09	10-12	01-03	04-06	07-09	10-12	01-03	04-06	07-09	10-12
1	■	■	■	■	■	■	■										
2		■	■	■	■	■											
3			■	■	■												
4					■	■	■										
5					■	■	■	■	■								
6						■	■	■	■	■	■						
7								■	■	■	■						
8								■	■	■	■	■	■				
9									■	■	■	■	■	■	■		
10	■																
11					■	■	■	■	■	■	■	■	■	■	■	■	■
12												■	■	■	■	■	■

Source: Elaborated by the author.

5.4 Research internship abroad

Several activities proposed to this Ph.D. research project will be developed in collaboration with Mohammad Reza Mousavi[1] and Neil Walkinshaw[2], both professors at the University of Leicester - United Kingdom (UK). Professor Mousavi has prior successful collaboration with Adenilso Simão, the advisor of this project, and two former Ph.D. students (FRAGAL; SIMAO; MOUSAVI, 2017; FRAGAL *et al.*, 2017; PAIVA *et al.*, 2016). He also has agreed to host the Ph.D. student for an extended period during his project. The university of Leicester is ranked in the top 1% of universities worldwide and have an international reputation for excellence in teaching and research. With this collaboration, we believe that we will be able to undertake more realistic scenarios.

5.5 Expected outcomes

As the main goal, this Ph.D. research project aims at contributing to the SPL area by enabling model inference of families of products of SPLs and support MBT of SPLs even if models are not specified a priori, complete, or updated. In details, the expected results are:

1. An overview on model inference and where it has been applied;
2. An efficient and effective model inference approach to extract FFSMs from SPLs;
3. An approach for merging multiple Mealy machines in a single FFSM model;
4. Guidelines for using configuration prioritization in family model inference;
5. Guidelines to perform LBT on families of products;
6. An approach to formally describe scenarios where feature interactions occur;
7. An approach to extract EFSMs from families of products;
8. Experimental comparisons of all proposed techniques;
9. Strengthen the coolaboration between USP and University of Leicester;
10. Tool support to execute and replicate of the proposed techniques;

The outcomes of this Ph.D. research will be disseminated through scientific journals, workshops, and conferences on software testing, SPL, and software engineering. We wil also maintain a laboratory package with all tools, models, SUTs/SULs, and results of experiments created along this project. This lab package will be hosted as an open-source project in github.

[1] <http://www2.le.ac.uk/departments/informatics/people/mohammad-mousavi>
[2] <http://www2.le.ac.uk/departments/informatics/people/neil-walkinshaw>

BIBLIOGRAPHY

AARTS, F.; FITERAU-BROSTEAN, P.; KUPPENS, H.; VAANDRAGER, F. Learning register automata with fresh value generation. In: ____. **Theoretical Aspects of Computing - ICTAC 2015: 12th International Colloquium, Cali, Colombia, October 29-31, 2015, Proceedings**. Cham: Springer International Publishing, 2015. p. 165–183. ISBN 978-3-319-25150-9. Available: <http://dx.doi.org/10.1007/978-3-319-25150-9_11>. Citations on pages 56 and 85.

AARTS, F.; HEIDARIAN, F.; KUPPENS, H.; OLSEN, P.; VAANDRAGER, F. Automata learning through counterexample guided abstraction refinement. In: ____. **FM 2012: Formal Methods: 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. p. 10–27. ISBN 978-3-642-32759-9. Available: <https://doi.org/10.1007/978-3-642-32759-9_4>. Citations on pages 23, 56, and 67.

ADAMIS, G.; KOVÁCS, G.; RÉTHY, G. Generating performance test model from conformance test logs. In: **Proceedings of the 17th International SDL Forum on SDL 2015: Model-Driven Engineering for Smart Cities - Volume 9369**. New York, NY, USA: Springer-Verlag New York, Inc., 2015. p. 268–284. ISBN 978-3-319-24911-7. Available: <http://dx.doi.org/10.1007/978-3-319-24912-4_19>. Citation on page 63.

AFSHAN, S.; MCMINN, P.; STEVENSON, M. Evolving readable string test inputs using a natural language model to reduce human oracle cost. In: **2013 IEEE Sixth International Conference on Software Testing, Verification and Validation**. [S.l.: s.n.], 2013. p. 352–361. ISSN 2159-4848. Citation on page 32.

ALEX. **ALEX: Automata Learning EXperience**. 2017. <https://learnlib.github.io/alex/>. [Online; accessed 17-Out-2017]. Citation on page 66.

ALLEN, F. E. Control flow analysis. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 5, n. 7, p. 1–19, Jul. 1970. ISSN 0362-1340. Available: <http://doi.acm.org/10.1145/390013.808479>. Citations on pages 37 and 48.

AMMANN, P.; OFFUTT, J. **Introduction to Software Testing**. 1. ed. New York, NY, USA: Cambridge University Press, 2008. ISBN 0521880386, 9780521880381. Citations on pages 31, 33, 35, 38, 39, 47, and 48.

ANGLUIN, D. Learning regular sets from queries and counterexamples. **Information and Computation**, v. 75, n. 2, p. 87 – 106, 1987. ISSN 0890-5401. Available: <http://www.sciencedirect.com/science/article/pii/0890540187900526>. Citations on pages 22, 41, 56, 57, 59, and 60.

APEL, S.; BATORY, D.; KÄSTNER, C.; SAAKE, G. Advanced, language-based variability mechanisms. In: ____. **Feature-Oriented Software Product Lines: Concepts and Implementation**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. p. 129–174. ISBN 978-3-642-37521-7. Available: <https://doi.org/10.1007/978-3-642-37521-7_6>. Citations on pages 76 and 77.

____. A development process for feature-oriented product lines. In: ____. **Feature-Oriented Software Product Lines: Concepts and Implementation**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. p. 17–44. ISBN 978-3-642-37521-7. Available: <https://doi.org/10.1007/978-3-642-37521-7_2>. Citation on page 77.

APEL, S.; KOLESNIKOV, S.; SIEGMUND, N.; KäSTNER, C.; GARVIN, B. Exploring feature interactions in the wild: The new feature-interaction challenge. In: **Proceedings of the 5th International Workshop on Feature-Oriented Software Development**. New York, NY, USA: ACM, 2013. (FOSD '13), p. 1–8. ISBN 978-1-4503-2168-6. Available: <http://doi.acm.org/10.1145/2528265.2528267>. Citation on page 25.

ASIRELLI, P.; BEEK, M. H. ter; FANTECHI, A.; GNESI, S. A compositional framework to derive product line behavioural descriptions. In: ____. **Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change: 5th International Symposium, ISoLA 2012, Heraklion, Crete, Greece, October 15-18, 2012, Proceedings, Part I**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. p. 146–161. ISBN 978-3-642-34026-0. Available: <http://dx.doi.org/10.1007/978-3-642-34026-0_12>. Citations on pages 22 and 83.

ATLEE, J. M.; BEIDU, S.; FAHRENBERG, U.; LEGAY, A. Merging Features in Featured Transition Systems. In: **Proceedings of the 12th Workshop on Model-Driven Engineering, Verification and Validation co-located with ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems**. Ottawa, Canada: [s.n.], 2015. (CEUR Workshop Proceedings, v. 1514), p. 38–43. Available: <https://hal.inria.fr/hal-01237661>. Citations on pages 24, 75, and 84.

BAIER, C.; KATOEN, J.-P. **Principles of Model Checking (Representation and Mind Series)**. [S.l.]: The MIT Press, 2008. ISBN 026202649X, 9780262026499. Citations on pages 41, 48, and 52.

BAINCZYK, A.; SCHIEWECK, A.; ISBERNER, M.; MARGARIA, T.; NEUBAUER, J.; STEFFEN, B. Alex: Mixed-mode learning of web applications at ease. In: ____. **Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications: 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part II**. Cham: Springer International Publishing, 2016. p. 655–671. ISBN 978-3-319-47169-3. Available: <https://doi.org/10.1007/978-3-319-47169-3_51>. Citations on pages 23 and 66.

BARNETT, M.; LEINO, K. R. M.; SCHULTE, W. The spec# programming system: An overview. In: ____. **Construction and Analysis of Safe, Secure, and Interoperable Smart Devices: International Workshop, CASSIS 2004, Marseille, France, March 10-14, 2004, Revised Selected Papers**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005. p. 49–69. ISBN 978-3-540-30569-9. Available: <https://doi.org/10.1007/978-3-540-30569-9_3>. Citation on page 52.

BARR, E. T.; HARMAN, M.; MCMINN, P.; SHAHBAZ, M.; YOO, S. The oracle problem in software testing: A survey. **IEEE Transactions on Software Engineering**, n. 5, p. 507–525, May 2015. ISSN 0098-5589. Citation on page 32.

BAUER, O.; NEUBAUER, J.; ISBERNER, M. Model-driven active automata learning with learnlib studio. In: ____. **Leveraging Applications of Formal Methods, Verification, and**

Validation : 6th International Symposium, ISoLA 2014, Corfu, Greece, October 8-11, 2014, and 5th International Symposium, ISoLA 2012, Heraklion, Crete, Greece, October 15-18, 2012, Revised Selected Papers. Cham: Springer International Publishing, 2016. p. 128–142. ISBN 978-3-319-51641-7. Available: <https://doi.org/10.1007/978-3-319-51641-7_8>. Citation on page 66.

BEKRAR, S.; BEKRAR, C.; GROZ, R.; MOUNIER, L. Finding software vulnerabilities by smart fuzzing. In: **Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation**. Washington, DC, USA: IEEE Computer Society, 2011. (ICST '11), p. 427–430. ISBN 978-0-7695-4342-0. Available: <http://dx.doi.org/10.1109/ICST.2011.48>. Citation on page 32.

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. Available: <http://www.sciencedirect.com/science/article/pii/S0306437910000025>. Citations on pages 21, 70, 71, and 72.

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**. New York, NY, USA: ACM, 2015. (VaMoS '15), p. 80:80–80:87. ISBN 978-1-4503-3273-6. Available: <http://doi.acm.org/10.1145/2701319.2701332>. Citation on page 74.

BEOHAR, H.; MOUSAVI, M. R. Input–output conformance testing for software product lines. **Journal of Logical and Algebraic Methods in Programming**, v. 85, n. 6, p. 1131 – 1153, 2016. ISSN 2352-2208. {NWPT} 2013. Available: <http://www.sciencedirect.com/science/article/pii/S2352220816301171>. Citation on page 75.

BEOHAR, H.; VARSHOSAZ, M.; MOUSAVI, M. R. Basic behavioral models for software product lines: Expressiveness and testing pre-orders. **Science of Computer Programming**, v. 123, p. 42 – 60, 2016. ISSN 0167-6423. Available: <http://www.sciencedirect.com/science/article/pii/S0167642315001288>. Citations on pages 22 and 83.

BERG, T.; RAFFELT, H. 19 model checking. Springer Berlin Heidelberg, Berlin, Heidelberg, p. 557–603, 2005. Available: <http://dx.doi.org/10.1007/11498490_25>. Citations on pages 22, 41, 55, and 63.

BERGER, T.; SHE, S.; LOTUFO, R.; WASOWSKI, A.; CZARNECKI, K. A study of variability models and languages in the systems software domain. **IEEE Transactions on Software Engineering**, v. 39, n. 12, p. 1611–1640, Dec 2013. ISSN 0098-5589. Citation on page 72.

BINDER, R. V. **Testing Object-oriented Systems: Models, Patterns, and Tools**. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN 0-201-80938-9. Citations on pages 41 and 55.

BOLLIG, B.; HABERMEHL, P.; KERN, C.; LEUCKER, M. Angluin-style learning of nfa. In: **Proceedings of the 21st International Jont Conference on Artifical Intelligence**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2009. (IJCAI'09), p. 1004–1009. Available: <http://dl.acm.org/citation.cfm?id=1661445.1661605>. Citation on page 56.

BOLLIG, B.; KATOEN, J.-P.; KERN, C.; LEUCKER, M.; NEIDER, D.; PIEGDON, D. R. libalf: The automata learning framework. In: ____. **Computer Aided Verification: 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. p. 360–364. ISBN 978-3-642-14295-6. Available: <https://doi.org/10.1007/978-3-642-14295-6_32>. Citation on page 65.

BRABRAND, C.; RIBEIRO, M.; TOLêDO, T.; BORBA, P. Intraprocedural dataflow analysis for software product lines. In: **Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development**. New York, NY, USA: ACM, 2012. (AOSD '12), p. 13–24. ISBN 978-1-4503-1092-5. Available: <http://doi.acm.org/10.1145/2162049.2162052>. Citation on page 74.

BROY, M.; JONSSON, B.; KATOEN, J.-P.; LEUCKER, M.; PRETSCHNER, A. **Model-Based Testing of Reactive Systems: Advanced Lectures**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005. 557–603 p. ISBN 978-3-540-32037-1. Available: <http://dx.doi.org/10.1007/11498490_25>. Citations on pages 22, 23, 32, 55, and 84.

BRYANT, R. E. Graph-based algorithms for boolean function manipulation. **IEEE Trans. Comput.**, IEEE Computer Society, Washington, DC, USA, v. 35, n. 8, p. 677–691, Aug. 1986. ISSN 0018-9340. Available: <http://dx.doi.org/10.1109/TC.1986.1676819>. Citation on page 52.

BURDY, L.; CHEON, Y.; COK, D. R.; ERNST, M. D.; KINIRY, J. R.; LEAVENS, G. T.; LEINO, K. R. M.; POLL, E. An overview of jml tools and applications. **International Journal on Software Tools for Technology Transfer**, v. 7, n. 3, p. 212–232, Jun 2005. ISSN 1433-2787. Available: <https://doi.org/10.1007/s10009-004-0167-4>. Citation on page 32.

CASSEL FALK HOWAR, B. J. S. Ralib: A learnlib extension for inferring efsms. **DIFTS 2015**, 2015. Available: <http://www.faculty.ece.vt.edu/chaowang/difts2015/papers/paper_5.pdf>. Citations on pages 56, 66, 85, and 87.

CASSEL, S.; HOWAR, F.; JONSSON, B.; STEFFEN, B. Active learning for extended finite state machines. **Formal Aspects of Computing**, v. 28, n. 2, p. 233–263, Apr 2016. ISSN 1433-299X. Available: <https://doi.org/10.1007/s00165-016-0355-5>. Citations on pages 26, 47, 56, 66, and 85.

CAVADA, R.; CIMATTI, A.; JOCHIM, C. A.; KEIGHREN, G.; OLIVETTI, E.; PISTORE, M.; ROVERI, M.; TCHALTSEV, A. **Nusmv 2.6 user manual**. [S.l.], 2010. Citation on page 52.

CHAKI, S.; CLARKE, E.; SHARYGINA, N.; SINHA, N. Verification of evolving software via component substitutability analysis. **Formal Methods in System Design**, v. 32, n. 3, p. 235–266, Jun 2008. ISSN 1572-8102. Available: <https://doi.org/10.1007/s10703-008-0053-x>. Citations on pages 24, 62, 63, and 85.

CHAKI, S.; SHARYGINA, N.; SINHA, N. Verification of evolving software. In: **Proceedings of the third workshop on specification and verification of component based systems**. Newport Beach, CA, USA: ACM, 2004. (SAVCBS '04), p. 55–61. Citations on pages 62 and 63.

CHAPMAN, M.; CHOCKLER, H.; KESSELI, P.; KROENING, D.; STRICHMAN, O.; TAUTSCHNIG, M. Learning the language of error. In: ____. **Automated Technology for Verification and Analysis: 13th International Symposium, ATVA 2015, Shanghai, China, October 12-15, 2015, Proceedings**. Cham: Springer International Publishing, 2015. p. 114–130. ISBN 978-3-319-24953-7. Available: <http://dx.doi.org/10.1007/978-3-319-24953-7_9>. Citations on pages 23, 25, 64, 83, and 85.

CHOI, W.; NECULA, G.; SEN, K. Guided gui testing of android apps with minimal restart and approximate learning. In: **Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications**. New York, NY, USA: ACM, 2013. (OOPSLA '13), p. 623–640. ISBN 978-1-4503-2374-1. Available: <http://doi.acm.org/10.1145/2509136.2509552>. Citation on page 63.

CHOW, T. S. Testing software design modeled by finite-state machines. **IEEE Transactions on Software Engineering**, IEEE Press, Piscataway, NJ, USA, v. 4, n. 3, p. 178–187, May 1978. ISSN 0098-5589. Citations on pages 22, 44, 45, 51, and 87.

CICHOS, H.; OSTER, S.; LOCHAU, M.; SCHÜRR, A. Model-based coverage-driven test suite generation for software product lines. In: ____. **Model Driven Engineering Languages and Systems: 14th International Conference, MODELS 2011, Wellington, New Zealand, October 16-21, 2011. Proceedings**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. p. 425–439. ISBN 978-3-642-24485-8. Available: <http://dx.doi.org/10.1007/978-3-642-24485-8_31>. Citation on page 76.

CLARKE, D.; HELVENSTEIJN, M.; SCHAEFER, I. Abstract delta modeling. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 46, n. 2, p. 13–22, Oct. 2010. ISSN 0362-1340. Available: <http://doi.acm.org/10.1145/1942788.1868298>. Citations on pages 74 and 78.

CLARKE, E.; MCMILLAN, K.; CAMPOS, S.; HARTONAS-GARMHAUSEN, V. Symbolic model checking. In: ____. **Computer Aided Verification: 8th International Conference, CAV '96 New Brunswick, NJ, USA, July 31– August 3, 1996 Proceedings**. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996. p. 419–422. ISBN 978-3-540-68599-9. Available: <https://doi.org/10.1007/3-540-61474-5_93>. Citation on page 52.

CLARKE, E. M.; EMERSON, E. A. Design and synthesis of synchronization skeletons using branching-time temporal logic. In: **Logic of Programs, Workshop**. London, UK, UK: Springer-Verlag, 1982. p. 52–71. ISBN 3-540-11212-X. Available: <http://dl.acm.org/citation.cfm?id=648063.747438>. Citation on page 49.

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 Transactions on Software Engineering**, v. 39, n. 8, p. 1069–1089, Aug 2013. ISSN 0098-5589. Citations on pages 24, 74, and 84.

CLASSEN, A.; HEYMANS, P.; SCHOBBENS, P.-Y.; LEGAY, A. Symbolic model checking of software product lines. In: **Proceedings of the 33rd International Conference on Software Engineering**. New York, NY, USA: ACM, 2011. (ICSE '11), p. 321–330. ISBN 978-1-4503-0445-0. Available: <http://doi.acm.org/10.1145/1985793.1985838>. Citations on pages 25 and 85.

CLEMENTS, P. C.; NORTHROP, L. **Software Product Lines: Practices and Patterns**. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001. (SEI Series in Software Engineering). ISBN 0-201-70332-7. Citations on pages 21 and 69.

COFER, D.; WHALEN, M.; MILLER, S. Software model checking for avionics systems. In: **2008 IEEE/AIAA 27th Digital Avionics Systems Conference**. [S.l.: s.n.], 2008. p. 5.D.5–1–5.D.5–8. ISSN 2155-7195. Citation on page 52.

COHEN, E.; DAHLWEID, M.; HILLEBRAND, M.; LEINENBACH, D.; MOSKAL, M.; SANTEN, T.; SCHULTE, W.; TOBIES, S. Vcc: A practical system for verifying concurrent c. In: ____. **Theorem Proving in Higher Order Logics: 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. p. 23–42. ISBN 978-3-642-03359-9. Available: <https://doi.org/10.1007/978-3-642-03359-9_2>. Citation on page 52.

CRANEN, S.; GROOTE, J. F.; KEIREN, J. J. A.; STAPPERS, F. P. M.; VINK, E. P. de; WESSELINK, W.; WILLEMSE, T. A. C. An overview of the mcrl2 toolset and its recent advances. In: ____. **Tools and Algorithms for the Construction and Analysis of Systems: 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. p. 199–213. ISBN 978-3-642-36742-7. Available: <http://dx.doi.org/10.1007/978-3-642-36742-7_15>. Citation on page 85.

DAMASCENO, C. D. N.; MASIERO, P. C.; SIMAO, A. Evaluating test characteristics and effectiveness of fsm-based testing methods on rbac systems. In: **Proceedings of the 30th Brazilian Symposium on Software Engineering**. New York, NY, USA: ACM, 2016. (SBES '16), p. 83–92. ISBN 978-1-4503-4201-8. Available: <http://doi.acm.org/10.1145/2973839.2973849>. Citation on page 22.

DELINE, R.; LEINO, K. R. M. **BoogiePL: A typed procedural language for checking object-oriented programs**. [S.l.], 2005. Citation on page 52.

DEUTSCH, M. S. Tutorial series 7 software project verification and validation. **Computer**, v. 14, n. 4, p. 54–70, April 1981. ISSN 0018-9162. Citation on page 31.

DOROFEEVA, R.; EL-FAKIH, K.; MAAG, S.; CAVALLI, A. R.; YEVTUSHENKO, N. Fsm-based conformance testing methods: A survey annotated with experimental evaluation. **Information and Software Technology**, v. 52, n. 12, p. 1286 – 1297, 2010. ISSN 0950-5849. Available: <http://www.sciencedirect.com/science/article/pii/S0950584910001278>. Citation on page 22.

EL-FAKIH, K.; SIMAO, A.; JADOON, N.; MALDONADO, J. C. An assessment of extended finite state machine test selection criteria. **Journal of Systems and Software**, v. 123, n. Supplement C, p. 106 – 118, 2017. ISSN 0164-1212. Available: <http://www.sciencedirect.com/science/article/pii/S0164121216301923>. Citations on pages 26, 47, 48, and 85.

ELBAUM, S.; ROTHERMEL, G.; KANDURI, S.; MALISHEVSKY, A. G. Selecting a cost-effective test case prioritization technique. **Software Quality Journal**, v. 12, n. 3, p. 185–210, 2004. ISSN 1573-1367. Available: <http://dx.doi.org/10.1023/B:SQJO.0000034708.84524.22>. Citations on pages 32, 50, and 51.

ENDO, A. T.; SIMAO, A. Evaluating test suite characteristics, cost, and effectiveness of fsm-based testing methods. **Information and Software Technology**, v. 55, n. 6, p. 1045 – 1062, 2013. ISSN 0950-5849. Available: <http://www.sciencedirect.com/science/article/pii/S0950584913000128>. Citations on pages 22, 24, and 46.

ERNST, M. D.; PERKINS, J. H.; GUO, P. J.; MCCAMANT, S.; PACHECO, C.; TSCHANTZ, M. S.; XIAO, C. The daikon system for dynamic detection of likely invariants. **Science of Computer Programming**, v. 69, n. 1, p. 35 – 45, 2007. ISSN 0167-6423. Special issue on Experimental Software and Toolkits. Available: <http://www.sciencedirect.com/science/article/pii/S016764230700161X>. Citation on page 32.

FABBRI, S. C. P. F.; DELAMARO, M. E.; MALDONADO, J. C.; MASIERO, P. C. Mutation analysis testing for finite state machines. In: **Software Reliability Engineering, 1994. Proceedings., 5th International Symposium on**. [S.l.: s.n.], 1994. p. 220–229. Citation on page 43.

FABBRI, S. C. P. F.; MALDONADO, J. C.; DELAMARO, M. Proteum/fsm: a tool to support finite state machine validation based on mutation testing. In: IEEE. **Computer Science Society, 1999. Proceedings. SCCC'99. XIX International Conference of the Chilean**. [S.l.], 1999. p. 96–104. Citation on page 51.

FBK. **NuSMV: a new symbolic model checker**. 2015. <http://nusmv.fbk.eu/>. [Online; accessed 20-Set-2017]. Citation on page 52.

FENG, L.; LUNDMARK, S.; MEINKE, K.; NIU, F.; SINDHU, M. A.; WONG, P. Y. H. Case studies in learning-based testing. In: ____. **Testing Software and Systems: 25th IFIP WG 6.1 International Conference, ICTSS 2013, Istanbul, Turkey, November 13-15, 2013, Proceedings**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. p. 164–179. ISBN 978-3-642-41707-8. Available: <http://dx.doi.org/10.1007/978-3-642-41707-8_11>. Citations on pages 23, 55, and 64.

FITERĂU-BROŞTEAN, P.; HOWAR, F. Learning-based testing the sliding window behavior of tcp implementations. In: ____. **Critical Systems: Formal Methods and Automated Verification: Joint 22nd International Workshop on Formal Methods for Industrial Critical Systems and 17th International Workshop on Automated Verification of Critical Systems, FMICS-AVoCS 2017, Turin, Italy, September 18–20, 2017, Proceedings**. Cham: Springer International Publishing, 2017. p. 185–200. ISBN 978-3-319-67113-0. Available: <https://doi.org/10.1007/978-3-319-67113-0_12>. Citations on pages 23 and 66.

FRAGAL, V. H.; SIMAO, A.; ENDO, A. T.; MOUSAVI, M. R. Reducing the concretization effort in fsm-based testing of software product lines. In: **2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)**. [S.l.: s.n.], 2017. p. 329–336. Citation on page 89.

FRAGAL, V. H.; SIMAO, A.; MOUSAVI, M. R. Validated test models for software product lines: Featured finite state machines. In: ____. **Formal Aspects of Component Software: 13th International Conference, FACS 2016, Besançon, France, October 19-21, 2016, Revised Selected Papers**. Cham: Springer International Publishing, 2017. p. 210–227. ISBN 978-3-319-57666-4. Available: <http://dx.doi.org/10.1007/978-3-319-57666-4_13>. Citations on pages 22, 23, 24, 26, 71, 75, 76, 83, 84, 85, 87, and 89.

FRASER, G.; WOTAWA, F.; AMMANN, P. E. Testing with model checkers: A survey. **Softw. Test. Verif. Reliab.**, John Wiley and Sons Ltd., Chichester, UK, v. 19, n. 3, p. 215–261, Sep. 2009. ISSN 0960-0833. Available: <http://dx.doi.org/10.1002/stvr.v19:3>. Citations on pages 49 and 64.

FUJIWARA, S.; BOCHMANN, G. v.; KHENDEK, F.; AMALOU, M.; GHEDAMSI, A. Test selection based on finite state models. **IEEE Transactions on Software Engineering**, v. 17, n. 6, p. 591–603, Jun 1991. ISSN 0098-5589. Citations on pages 46 and 87.

GILL, A. **Introduction to the Theory of Finite State Machines**. New York: McGraw-Hill, 1962. Citation on page 42.

GONZALEZ, A.; LUNA, C. Behavior specification of product lines via feature models and uml statecharts with variabilities. In: **2008 International Conference of the Chilean Computer Science Society**. [S.l.: s.n.], 2008. p. 32–41. ISSN 1522-4902. Citations on pages 26 and 85.

GOODENOUGH, J. B.; GERHART, S. L. Toward a theory of test data selection. **IEEE Transactions on Software Engineering**, SE-1, n. 2, p. 156–173, June 1975. ISSN 0098-5589. Citation on page 33.

GRINDAL, M.; OFFUTT, J.; ANDLER, S. F. Combination testing strategies: a survey. **Software Testing, Verification and Reliability**, John Wiley & Sons, Ltd., v. 15, n. 3, p. 167–199, 2005. ISSN 1099-1689. Available: <http://dx.doi.org/10.1002/stvr.319>. Citation on page 24.

GROCE, A.; PELED, D.; YANNAKAKIS, M. Adaptive model checking. In: **Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems**. London, UK, UK: Springer-Verlag, 2002. (TACAS '02), p. 357–370. ISBN 3-540-43419-4. Available: <http://dl.acm.org/citation.cfm?id=646486.694482>. Citations on pages 57 and 63.

GROZ, R.; SIMAO, A.; PETRENKO, A.; ORIAT, C. Inferring finite state machines without reset using state identification sequences. In: ____. **Testing Software and Systems: 27th IFIP WG 6.1 International Conference, ICTSS 2015, Sharjah and Dubai, United Arab Emirates, November 23-25, 2015, Proceedings**. Cham: Springer International Publishing, 2015. p. 161–177. ISBN 978-3-319-25945-1. Available: <http://dx.doi.org/10.1007/978-3-319-25945-1_10>. Citations on pages 23, 62, and 83.

GRULER, A.; LEUCKER, M.; SCHEIDEMANN, K. Modeling and model checking software product lines. In: ____. **Formal Methods for Open Object-Based Distributed Systems: 10th IFIP WG 6.1 International Conference, FMOODS 2008, Oslo, Norway, June 4-6, 2008 Proceedings**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. p. 113–131. ISBN 978-3-540-68863-1. Available: <https://doi.org/10.1007/978-3-540-68863-1_8>. Citation on page 75.

GRUMBERG, O.; VEITH, H. (Ed.). **25 Years of Model Checking: History, Achievements, Perspectives**. Berlin, Heidelberg: Springer-Verlag, 2008. ISBN 978-3-540-69849-4. Citation on page 49.

HABER, A.; HöLLDOBLER, K.; KOLASSA, C.; LOOK, M.; RUMPE, B.; MüLLER, K.; SCHAEFER, I. Engineering delta modeling languages. In: **Proceedings of the 17th International Software Product Line Conference**. New York, NY, USA: ACM, 2013. (SPLC '13), p. 22–31. ISBN 978-1-4503-1968-3. Available: <http://doi.acm.org/10.1145/2491627.2491632>. Citation on page 76.

HABER, A.; KOLASSA, C.; MANHART, P.; NAZARI, P. M. S.; RUMPE, B.; SCHAEFER, I. First-class variability modeling in matlab/simulink. In: **Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems**. New York, NY, USA: ACM, 2013. (VaMoS '13), p. 4:1–4:8. ISBN 978-1-4503-1541-8. Available: <http://doi.acm.org/10.1145/2430502.2430508>. Citation on page 76.

HABER, A.; KUTZ, T.; RENDEL, H.; RUMPE, B.; SCHAEFER, I. Delta-oriented architectural variability using monticore. In: **Proceedings of the 5th European Conference on Software Architecture: Companion Volume**. New York, NY, USA: ACM, 2011. (ECSA '11), p. 6:1–6:10. ISBN 978-1-4503-0618-8. Available: <http://doi.acm.org/10.1145/2031759.2031767>. Citation on page 76.

HAGERER, A.; HUNGAR, H.; NIESE, O.; STEFFEN, B. Model generation by moderated regular extrapolation. In: ____. **Fundamental Approaches to Software Engineering: 5th International Conference, FASE 2002 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 8–12, 2002 Proceedings**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002. p. 80–95. ISBN 978-3-540-45923-1. Available: <http://dx.doi.org/10.1007/3-540-45923-5_6>. Citations on pages 23 and 63.

HENARD, C.; PAPADAKIS, M.; PERROUIN, G.; KLEIN, J.; HEYMANS, P.; TRAON, Y. L. Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines. **IEEE Transactions on Software Engineering**, IEEE, v. 40, n. 7, p. 650–670, 2014. Citations on pages 24 and 85.

HOPCROFT, J. E. **Introduction to Automata Theory, Languages, and Computation**. 3rd. ed. [S.l.]: Pearson Addison Wesley, 2007. ISBN 0321455371, 9780321455376. Citation on page 47.

HOWAR, F.; STEFFEN, B.; MERTEN, M. From zulu to rers. In: ____. **Leveraging Applications of Formal Methods, Verification, and Validation: 4th International Symposium on Leveraging Applications, ISoLA 2010, Heraklion, Crete, Greece, October 18-21, 2010, Proceedings, Part I**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. p. 687–704. ISBN 978-3-642-16558-0. Available: <http://dx.doi.org/10.1007/978-3-642-16558-0_55>. Citations on pages 59 and 66.

HUNGAR, H.; NIESE, O.; STEFFEN, B. Domain-specific optimization in automata learning. In: ____. **Computer Aided Verification: 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003. Proceedings**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003. p. 315–327. ISBN 978-3-540-45069-6. Available: <https://doi.org/10.1007/978-3-540-45069-6_31>. Citations on pages 23 and 63.

IEEE. Ieee standard classification for software anomalies. **IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993)**, p. 1–23, Jan 2010. Citation on page 32.

____. Systems and software engineering – vocabulary. **ISO/IEC/IEEE 24765:2010(E)**, p. 1–418, Dec 2010. Citations on pages 31 and 32.

____. Ieee standard for system and software verification and validation. **IEEE Std 1012-2012 (Revision of IEEE Std 1012-2004)**, p. 1–223, May 2012. Citation on page 31.

IRFAN, M. N.; ORIAT, C.; GROZ, R. Angluin style finite state machine inference with non-optimal counterexamples. In: **Proceedings of the First International wWorkshop on Model Inference In Testing**. New York, NY, USA: ACM, 2010. (MIIT '10), p. 11–19. ISBN 978-1-4503-0147-3. Available: <http://doi.acm.org/10.1145/1868044.1868046>. Citation on page 60.

____. Chapter 3 - model inference and testing. In: MEMON, A. (Ed.). Elsevier, 2013, (Advances in Computers, v. 89). p. 89 – 139. Available: <http://www.sciencedirect.com/science/article/pii/B9780124080942000035>. Citations on pages 22, 55, 56, 60, and 84.

ISBERNER, M.; HOWAR, F.; STEFFEN, B. Learning register automata: from languages to program structures. **Machine Learning**, v. 96, n. 1, p. 65–98, Jul 2014. ISSN 1573-0565. Available: <https://doi.org/10.1007/s10994-013-5419-7>. Citation on page 55.

____. The open-source learnlib. In: ____. **Computer Aided Verification: 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I**. Cham: Springer International Publishing, 2015. p. 487–495. ISBN 978-3-319-21690-4. Available: <https://doi.org/10.1007/978-3-319-21690-4_32>. Citations on pages 32, 65, and 87.

ISBERNER, M.; STEFFEN, B.; HOWAR, F. Learnlib tutorial. In: ____. **Runtime Verification: 6th International Conference, RV 2015, Vienna, Austria, September 22-25, 2015. Proceedings**. Cham: Springer International Publishing, 2015. p. 358–377. ISBN 978-3-319-23820-3. Available: <https://doi.org/10.1007/978-3-319-23820-3_25>. Citations on pages 65 and 87.

JIA, Y.; HARMAN, M. Higher order mutation testing. **Information and Software Technology**, v. 51, n. 10, p. 1379 – 1393, 2009. ISSN 0950-5849. Source Code Analysis and Manipulation, SCAM 2008. Available: <http://www.sciencedirect.com/science/article/pii/S0950584909000688>. Citation on page 39.

____. An analysis and survey of the development of mutation testing. **IEEE Transactions on Software Engineering**, v. 37, n. 5, p. 649–678, Sept 2011. ISSN 0098-5589. Citations on pages 39, 42, and 43.

JORGENSEN, P. **Software Testing: A Craftsman's Approach, Fourth Edition**. Taylor & Francis, 2013. (An Auerbach book). ISBN 9781466560680. Available: <https://books.google.com.br/books?id=6WlmAQAAQBAJ>. Citations on pages 32 and 37.

KANG, K.; COHEN, S.; HESS, J.; NOVAK, W.; PETERSON, A. **Feature-Oriented Domain Analysis (FODA) Feasibility Study**. Pittsburgh, PA, 1990. Available: <http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=11231>. Citation on page 71.

KHOSROWJERDI, H.; MEINKE, K.; RASMUSSON, A. Learning-based testing for safety critical automotive applications. In: ____. **Model-Based Safety and Assessment: 5th International Symposium, IMBSA 2017, Trento, Italy, September 11–13, 2017, Proceedings**. Cham: Springer International Publishing, 2017. p. 197–211. ISBN 978-3-319-64119-5. Available: <https://doi.org/10.1007/978-3-319-64119-5_13>. Citation on page 64.

KNÜPPEL, A.; THÜM, T.; MENNICKE, S.; MEINICKE, J.; SCHAEFER, I. Is there a mismatch between real-world feature models and product-line research? In: **Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering**. New York, NY, USA: ACM, 2017. (ESEC/FSE 2017), p. 291–302. ISBN 978-1-4503-5105-8. Available: <http://doi.acm.org/10.1145/3106237.3106252>. Citation on page 72.

KUNZE, S.; MOSTOWSKI, W.; MOUSAVI, M. R.; VARSHOSAZ, M. Generation of failure models through automata learning. In: **2016 Workshop on Automotive Systems/Software Architectures (WASA)**. [S.l.: s.n.], 2016. p. 22–25. Citations on pages 23, 25, 64, 83, and 85.

LARSEN, K. G.; THOMSEN, B. A modal process logic. In: **[1988] Proceedings. Third Annual Symposium on Logic in Computer Science**. [S.l.: s.n.], 1988. p. 203–210. Citation on page 75.

LearnLib. **LearnLib: a framework for automata learning**. 2017. <https://learnlib.de/>. [Online; accessed 17-Out-2017]. Citations on pages 65 and 85.

LI, K.; GROZ, R.; SHAHBAZ, M. Integration testing of components guided by incremental state machine learning. In: **Testing: Academic Industrial Conference - Practice And Research Techniques (TAIC PART'06)**. [S.l.: s.n.], 2006. p. 59–70. Citations on pages 56 and 63.

libalf. **libalf: The Automata Learning Framework**. 2017. <http://libalf.informatik.rwth-aachen.de/>. [Online; accessed 17-Out-2017]. Citation on page 65.

LINDEN, F. J. v. d.; SCHMID, K.; ROMMES, E. **Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering**. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2007. ISBN 3540714367. Citations on pages 21 and 69.

LITY, S.; LACHMANN, R.; LOCHAU, M.; SCHAEFER, I. **Delta-oriented Software Product Line Test Models - The Body Comfort System Case Study**. 2013. Available: <http://tubiblio.ulb.tu-darmstadt.de/73869/>. Citations on pages 76 and 87.

MACHADO, P.; VINCENZI, A.; MALDONADO, J. C. Software testing: An overview. In: ____. **Testing Techniques in Software Engineering: Second Pernambuco Summer School on Software Engineering, PSSE 2007, Recife, Brazil, December 3-7, 2007, Revised Lectures**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. p. 1–17. ISBN 978-3-642-14335-9. Available: <https://doi.org/10.1007/978-3-642-14335-9_1>. Citation on page 33.

MALER, O.; PNUELI, A. On the learnability of infinitary regular sets. **Information and Computation**, v. 118, n. 2, p. 316 – 326, 1995. ISSN 0890-5401. Available: <http://www.sciencedirect.com/science/article/pii/S089054018571070X>. Citation on page 60.

MARGARIA, T.; RAFFELT, H.; STEFFEN, B. Knowledge-based relevance filtering for efficient system-level test-based model generation. **Innovations in Systems and Software Engineering**, v. 1, n. 2, p. 147–156, Sep 2005. ISSN 1614-5054. Available: <https://doi.org/10.1007/s11334-005-0016-y>. Citations on pages 25, 61, and 85.

MARIANI, L.; PEZZÈ, M.; ZUDDAS, D. Chapter four - recent advances in automatic black-box testing. In: MEMON, A. (Ed.). Elsevier, 2015, (Advances in Computers, v. 99). p. 157 – 193. Available: <http://www.sciencedirect.com/science/article/pii/S0065245815000315>. Citations on pages 23, 41, 55, and 63.

MARINESCU, R.; SECELEANU, C.; GUEN, H. L.; PETTERSSON, P. Chapter three - a research overview of tool-supported model-based testing of requirements-based designs. In: HURSON, A. R. (Ed.). Elsevier, 2015, (Advances in Computers, v. 98). p. 89 – 140. Available: <http://www.sciencedirect.com/science/article/pii/S0065245815000297>. Citations on pages 41 and 55.

MASOOD, A.; BHATTI, R.; GHAFOOR, A.; MATHUR, A. P. Scalable and effective test generation for role-based access control systems. **IEEE Transactions on Software Engineering**, IEEE Press, Piscataway, NJ, USA, v. 35, n. 5, p. 654–668, Sep. 2009. ISSN 0098-5589. Citation on page 22.

MASSOL, V.; HUSTED, T. **JUnit in Action**. Greenwich, CT, USA: Manning Publications Co., 2003. ISBN 1930110995. Citation on page 32.

MCGREGOR, J. D. **Testing a Software Product Line**. [S.l.], 2001. Citation on page 21.

MCMILLAN, K. L. **Symbolic Model Checking**. Norwell, MA, USA: Kluwer Academic Publishers, 1993. ISBN 0792393805. Citation on page 52.

MEINICKE, J.; THÜM, T.; SCHRÖTER, R.; BENDUHN, F.; LEICH, T.; SAAKE, G. **Mastering Software Variability with FeatureIDE**. Cham: Springer International Publishing, 2017. 3–10 p. ISBN 978-3-319-61443-4. Available: <https://doi.org/10.1007/978-3-319-61443-4_1>. Citation on page 77.

MEINKE, K. Learning-based testing of cyber-physical systems-of-systems: A platooning study. In: ____. **Computer Performance Engineering: 14th European Workshop, EPEW 2017, Berlin, Germany, September 7-8, 2017, Proceedings**. Cham: Springer International Publishing, 2017. p. 135–151. ISBN 978-3-319-66583-2. Available: <https://doi.org/10.1007/978-3-319-66583-2_9>. Citation on page 64.

MEINKE, K.; NIU, F. A learning-based approach to unit testing of numerical software. In: ____. **Testing Software and Systems: 22nd IFIP WG 6.1 International Conference, ICTSS 2010, Natal, Brazil, November 8-10, 2010. Proceedings**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. p. 221–235. ISBN 978-3-642-16573-3. Available: <http://dx.doi.org/10.1007/978-3-642-16573-3_16>. Citation on page 64.

MEINKE, K.; NYCANDER, P. Learning-based testing of distributed microservice architectures: Correctness and fault injection. In: ____. **Software Engineering and Formal Methods: SEFM 2015 Collocated Workshops: ATSE, HOFM, MoKMaSD, and VERY*SCART, York, UK, September 7-8, 2015. Revised Selected Papers**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015. p. 3–10. ISBN 978-3-662-49224-6. Available: <https://doi.org/10.1007/978-3-662-49224-6_1>. Citation on page 64.

MEINKE, K.; SINDHU, M. A. Incremental learning-based testing for reactive systems. In: ____. **Tests and Proofs: 5th International Conference, TAP 2011, Zurich, Switzerland, June 30 – July 1, 2011. Proceedings**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. p. 134–151. ISBN 978-3-642-21768-5. Available: <http://dx.doi.org/10.1007/978-3-642-21768-5_11>. Citations on pages 25, 63, and 64.

____. LBTest: A learning-based testing tool for reactive systems. In: **2013 IEEE Sixth International Conference on Software Testing, Verification and Validation**. [S.l.: s.n.], 2013. p. 447–454. ISSN 2159-4848. Citations on pages 25 and 85.

MEINKE, K.; WALKINSHAW, N. Model-based testing and model inference. In: ____. **Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change: 5th International Symposium, ISoLA 2012, Heraklion, Crete, Greece, October 15-18, 2012, Proceedings, Part I**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. p. 440–443. ISBN 978-3-642-34026-0. Available: <http://dx.doi.org/10.1007/978-3-642-34026-0_32>. Citation on page 22.

MERTEN, M.; HOWAR, F.; STEFFEN, B.; MARGARIA, T. Automata learning with on-the-fly direct hypothesis construction. In: ____. **Leveraging Applications of Formal Methods, Verification, and Validation: International Workshops, SARS 2011 and MLSC 2011, Held Under the Auspices of ISoLA 2011 in Vienna, Austria, October 17-18, 2011. Revised Selected Papers**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. p. 248–260. ISBN 978-3-642-34781-8. Available: <http://dx.doi.org/10.1007/978-3-642-34781-8_19>. Citation on page 83.

Microsoft Research. **The Z3 Theorem Prover**. 2017. <https://github.com/Z3Prover/z3/>. [Online; accessed 20-Set-2017]. Citation on page 52.

MILLER, S. P.; WHALEN, M. W.; COFER, D. D. Software model checking takes off. **Commun. ACM**, ACM, New York, NY, USA, v. 53, n. 2, p. 58–64, Feb. 2010. ISSN 0001-0782. Available: <http://doi.acm.org/10.1145/1646353.1646372>. Citation on page 49.

MLYNARSKI, M.; GÜLDALI, B.; WEIßLEDER, S.; ENGELS, G. Model-based testing: Achievements and future challenges. In: HURSON, A.; MEMON, A. (Ed.). Elsevier, 2012, (Advances in Computers, v. 86). p. 1 – 39. Available: <http://www.sciencedirect.com/science/article/pii/B9780123965356000016>. Citation on page 83.

MOURA, L. D.; BJØRNER, N. Z3: An efficient smt solver. In: **Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems**. Berlin, Heidelberg: Springer-Verlag, 2008. (TACAS'08/ETAPS'08), p. 337–340. ISBN 3-540-78799-2, 978-3-540-78799-0. Available: <http://dl.acm.org/citation.cfm?id=1792734.1792766>. Citations on pages 52 and 76.

MYERS, G. J.; SANDLER, C.; BADGETT, T. **The Art of Software Testing**. 3rd. ed. John Wiley & Sons, Inc., 2012. i–xi p. ISBN 1118031962, 9781118031964. Available: <http://dx.doi.org/10.1002/9781119202486>. Citations on pages 31, 35, 36, and 38.

NEUBAUER, J.; STEFFEN, B.; BAUER, O.; WINDMÜLLER, S.; MERTEN, M.; MARGARIA, T.; HOWAR, F. Automated continuous quality assurance. In: **2012 First International Workshop on Formal Methods in Software Engineering: Rigorous and Agile Approaches (FormSERA)**. [S.l.: s.n.], 2012. p. 37–43. Citations on pages 23, 55, and 63.

NIE, C.; LEUNG, H. A survey of combinatorial testing. **ACM Comput. Surv.**, ACM, New York, NY, USA, v. 43, n. 2, p. 11:1–11:29, Feb. 2011. ISSN 0360-0300. Available: <http://doi.acm.org/10.1145/1883612.1883618>. Citation on page 24.

NIESE, O. **An integrated approach to testing complex systems**. Phd Thesis (PhD Thesis) — University of Dortmund, 2003. Available: <http://d-nb.info/969717474>. Citation on page 56.

OFFUTT, A. J.; PAN, J. Automatically detecting equivalent mutants and infeasible paths. **Software Testing, Verification and Reliability**, John Wiley & Sons, Ltd, v. 7, n. 3, p. 165–192, 1997. ISSN 1099-1689. Available: <http://dx.doi.org/10.1002/(SICI)1099-1689(199709)7:3<165::AID-STVR143>3.0.CO;2-U>. Citation on page 40.

OLAECHEA, R.; FAHRENBERG, U.; ATLEE, J. M.; LEGAY, A. Long-term average cost in featured transition systems. In: **Proceedings of the 20th International Systems and Software Product Line Conference**. New York, NY, USA: ACM, 2016. (SPLC '16), p. 109–118. ISBN 978-1-4503-4050-2. Available: <http://doi.acm.org/10.1145/2934466.2934473>. Citation on page 75.

OSTER, S. **Feature Model-based Software Product Line Testing**. Phd Thesis (PhD Thesis) — Technische Universität, 2012. Citations on pages 22 and 83.

OSTER, S.; WÜBBEKE, A.; ENGELS, G.; SCHÜRR, A. A survey of model-based software product lines testing. In: **Model-Based Testing for Embedded Systems**. [S.l.]: CRC Press, 2011. p. 338–381. Citations on pages 21 and 69.

OSTER, S.; ZORCIC, I.; MARKERT, F.; LOCHAU, M. Moso-polite: Tool support for pairwise and model-based software product line testing. In: **Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems**. New York, NY, USA: ACM, 2011. (VaMoS '11), p. 79–82. ISBN 978-1-4503-0570-9. Available: <http://doi.acm.org/10.1145/1944892.1944901>. Citations on pages 26 and 85.

OSTRAND, T. J.; BALCER, M. J. The category-partition method for specifying and generating fuctional tests. **Commun. ACM**, ACM, New York, NY, USA, v. 31, n. 6, p. 676–686, Jun. 1988. ISSN 0001-0782. Available: <http://doi.acm.org/10.1145/62959.62964>. Citation on page 35.

OURIQUES, J. a. F. S. Strategies for prioritizing test cases generated through model-based testing approaches. In: **Proceedings of the 37th International Conference on Software Engineering - Volume 2**. Piscataway, NJ, USA: IEEE Press, 2015. (ICSE '15), p. 879–882. Available: <http://dl.acm.org/citation.cfm?id=2819009.2819204>. Citation on page 49.

OvGU. **SPL2go: a catalog of software product lines @ Otto-von-Guericke University Magdeburg**. 2011. <http://spl2go.cs.ovgu.de/>. [Online; accessed 20-Set-2017]. Citation on page 87.

PAIVA, S. C.; SIMAO, A.; VARSHOSAZ, M.; MOUSAVI, M. R. Complete ioco test cases: A case study. In: **Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation**. New York, NY, USA: ACM, 2016. (A-TEST 2016), p. 38–44. ISBN 978-1-4503-4401-2. Available: <http://doi.acm.org/10.1145/2994291.2994297>. Citations on pages 26 and 89.

PELED, D.; VARDI, M. Y.; YANNAKAKIS, M. Black box checking. In: ____. **Formal Methods for Protocol Engineering and Distributed Systems (FORTE XII)**. Boston, MA: Springer US, 1999. p. 225–240. ISBN 978-0-387-35578-8. Available: <http://dx.doi.org/10.1007/978-0-387-35578-8_13>. Citations on pages 23, 63, and 83.

PETRENKO, A.; BOCHMANN, G. V. Selecting test sequences for partially-specified nondeterministic finite state machines. In: LUO, G. (Ed.). **7th IFIP WG 6.1 International Workshop on Protocol Test Systems**. London, UK, UK: Chapman and Hall, Ltd., 1995. (IWPTS '94), p. 95–110. ISBN 0-412-71160-5. Available: <http://dl.acm.org/citation.cfm?id=236187.233118>. Citations on pages 22, 46, and 51.

PETRENKO, A.; BORODAY, S.; GROZ, R. Confirming configurations in efsm testing. **IEEE Transactions on Software Engineering**, v. 30, n. 1, p. 29–42, Jan 2004. ISSN 0098-5589. Citations on pages 26, 48, and 85.

PINHEIRO, A. C. **Subsídios para a aplicação de métodos de geração de casos de testes baseados em máquinas de estados**. Master's Thesis (Master's Thesis) — Universidade de São Paulo, 2012. Citation on page 51.

PINHEIRO, A. C.; SIMÃO, A.; AMBROSIO, A. M. Fsm-based test case generation methods applied to test the communication software on board the itasat university satellite: A case study. **Journal of Aerospace Technology and Management**, SciELO Brasil, v. 6, n. 4, p. 447–461, 2014. Citation on page 22.

PNUELI, A. The temporal logic of programs. In: **Proceedings of the 18th Annual Symposium on Foundations of Computer Science**. Washington, DC, USA: IEEE Computer Society, 1977. (SFCS '77), p. 46–57. Available: <http://dx.doi.org/10.1109/SFCS.1977.32>. Citation on page 49.

POHL, K.; BÖCKLE, G.; LINDEN, F. J. v. d. **Software Product Line Engineering: Foundations, Principles and Techniques**. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005. ISBN 3540243720. Citations on pages 21, 69, 70, 71, and 72.

RAFFELT, H.; MERTEN, M.; STEFFEN, B.; MARGARIA, T. Dynamic testing via automata learning. **International Journal on Software Tools for Technology Transfer**, v. 11, n. 4, p. 307, 2009. ISSN 1433-2787. Available: <http://dx.doi.org/10.1007/s10009-009-0120-7>. Citations on pages 23, 63, and 83.

RAFFELT, H.; STEFFEN, B. Learnlib: A library for automata learning and experimentation. In: ____. **Fundamental Approaches to Software Engineering: 9th International Conference, FASE 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006. Proceedings**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006. p. 377–380. ISBN 978-3-540-33094-3. Available: <https://doi.org/10.1007/11693017_28>. Citations on pages 65 and 87.

RALib. **RALib: A LearnLib extension for inferring EFSMs**. 2017. <https://bitbucket.org/learnlib/ralib/>. [Online; accessed 17-Out-2017]. Citations on pages 66 and 85.

RAPPS, S.; WEYUKER, E. J. Selecting software test data using data flow information. **IEEE Trans. Softw. Eng.**, IEEE Press, Piscataway, NJ, USA, v. 11, n. 4, p. 367–375, Apr. 1985. ISSN 0098-5589. Available: <http://dx.doi.org/10.1109/TSE.1985.232226>. Citations on pages 38, 39, and 48.

RIVEST, R.; SCHAPIRE, R. Inference of finite automata using homing sequences. **Information and Computation**, v. 103, n. 2, p. 299 – 347, 1993. ISSN 0890-5401. Available: <http://www.sciencedirect.com/science/article/pii/S0890540183710217>. Citation on page 60.

RUITER, J. D.; POLL, E. Protocol state fuzzing of tls implementations. In: **Proceedings of the 24th USENIX Conference on Security Symposium**. Berkeley, CA, USA: USENIX Association, 2015. (SEC'15), p. 193–206. ISBN 978-1-931971-232. Available: <http://dl.acm.org/citation.cfm?id=2831143.2831156>. Citations on pages 23, 32, and 63.

SABOURI, H.; KHOSRAVI, R. Delta modeling and model checking of product families. In: ____. **Fundamentals of Software Engineering: 5th International Conference, FSEN 2013, Tehran, Iran, April 24-26, 2013, Revised Selected Papers**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. p. 51–65. ISBN 978-3-642-40213-5. Available: <http://dx.doi.org/10.1007/978-3-642-40213-5_4>. Citations on pages 22, 25, 74, 76, and 83.

SCHAEFER, I. Variability modelling for model-driven development of software product lines. In: **Fourth International Workshop on Variability Modelling of Software-Intensive Systems, Linz, Austria, January 27-29, 2010. Proceedings**. [s.n.], 2010. p. 85–92. Available: <http://www.vamos-workshop.net/proceedings/VaMoS_2010_Proceedings.pdf>. Citation on page 76.

SCHAEFER, I.; BETTINI, L.; BONO, V.; DAMIANI, F.; TANZARELLA, N. Delta-oriented programming of software product lines. In: ____. **Software Product Lines: Going Beyond: 14th International Conference, SPLC 2010, Jeju Island, South Korea, September 13-17, 2010. Proceedings**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. p. 77–91. ISBN 978-3-642-15579-6. Available: <http://dx.doi.org/10.1007/978-3-642-15579-6_6>. Citations on pages 22, 74, 76, 77, 78, and 83.

SCHAEFER, I.; RABISER, R.; CLARKE, D.; BETTINI, L.; BENAVIDES, D.; BOTTERWECK, G.; PATHAK, A.; TRUJILLO, S.; VILLELA, K. Software diversity: state of the art and perspectives. **International Journal on Software Tools for Technology Transfer**, v. 14, n. 5, p. 477–495, 2012. ISSN 1433-2787. Available: <http://dx.doi.org/10.1007/s10009-012-0253-y>. Citations on pages 21, 71, and 74.

SCHOBBENS, P.-Y.; HEYMANS, P.; TRIGAUX, J.-C.; BONTEMPS, Y. Generic semantics of feature diagrams. **Computer Networks**, v. 51, n. 2, p. 456 – 479, 2007. ISSN 1389-1286. Feature Interaction. Available: <http://www.sciencedirect.com/science/article/pii/S1389128606002179>. Citation on page 71.

SCHUTS, M.; HOOMAN, J.; VAANDRAGER, F. Refactoring of legacy software using model learning and equivalence checking: An industrial experience report. In: ____. **Integrated Formal Methods: 12th International Conference, IFM 2016, Reykjavik, Iceland, June 1-5, 2016, Proceedings**. Cham: Springer International Publishing, 2016. p. 311–325. ISBN 978-3-319-33693-0. Available: <http://dx.doi.org/10.1007/978-3-319-33693-0_20>. Citations on pages 23 and 55.

SHAHBAZ, M.; GROZ, R. Inferring mealy machines. In: ____. **FM 2009: Formal Methods: Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. p. 207–222. ISBN 978-3-642-05089-3. Available: <http://dx.doi.org/10.1007/978-3-642-05089-3_14>. Citations on pages 22, 24, 56, 57, 58, 59, 60, 83, 84, and 86.

____. Analysis and testing of black-box component-based systems by inferring partial models. **Softw. Test. Verif. Reliab.**, John Wiley and Sons Ltd., Chichester, UK, v. 24, n. 4, p. 253–288, Jun. 2014. ISSN 0960-0833. Available: <http://dx.doi.org/10.1002/stvr.1491>. Citations on pages 23, 55, and 63.

SHAHBAZ, M.; LI, K.; GROZ, R. Learning and integration of parameterized components through testing. In: PETRENKO, A.; VEANES, M.; TRETMANS, J.; GRIESKAMP, W. (Ed.). **Testing of Software and Communicating Systems: 19th IFIP TC6/WG6.1 International Conference, TestCom 2007, 7th International Workshop, FATES 2007, Tallinn, Estonia, June 26-29, 2007. Proceedings**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007. p. 319–334. ISBN 978-3-540-73066-8. Available: <http://dx.doi.org/10.1007/978-3-540-73066-8_22>. Citation on page 63.

SHAHBAZ, M.; PARREAUX, B.; KLAY, F. Model inference approach for detecting feature interactions in integrated systems. In: **Feature Interactions in Software and Communication Systems IX, International Conferenceserence on Feature Interactions in Software and Communication Systems, ICFI 2007, 3-5 September 2007, Grenoble, France**. [S.l.: s.n.], 2007. p. 161–171. Citations on pages 23, 63, and 83.

SHARYGINA, N.; CHAKI, S.; CLARKE, E.; SINHA, N. Dynamic component substitutability analysis. In: ____. **FM 2005: Formal Methods: International Symposium of Formal Methods Europe, Newcastle, UK, July 18-22, 2005. Proceedings**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005. p. 512–528. ISBN 978-3-540-31714-2. Available: <https://doi.org/10.1007/11526841_34>. Citation on page 24.

SIMÃO, A.; PETRENKO, A.; YEVTUSHENKO, N. Generating reduced tests for fsms with extra states. In: NUNEZ, M.; BAKER, P.; MERAYO, M. (Ed.). **Testing of Software and Communication Systems**. Springer Berlin Heidelberg, 2009, (Lecture Notes in Computer Science, v. 5826). p. 129–145. ISBN 978-3-642-05030-5. Available: <http://dx.doi.org/10.1007/978-3-642-05031-2_9>. Citations on pages 22, 46, and 51.

SIMAO, A. da S.; AMBRÓSIO, A. M.; FABBRI, S. C.; AMARAL, A. S. M. S. do; MARTINS, E.; MALDONADO, J. C. Plavis/fsm: an environment to integrate fsm-based testing tools. In:

Tool Session of XIX Brazilian Symposium on Software Engineering. [S.l.: s.n.], 2005. p. 1–6. Citation on page 51.

SIRJANI, M.; JAGHOORI, M. M. Ten years of analyzing actors: Rebeca experience. In: ____. **Formal Modeling: Actors, Open Systems, Biological Systems: Essays Dedicated to Carolyn Talcott on the Occasion of Her 70th Birthday**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. p. 20–56. ISBN 978-3-642-24933-4. Available: <http://dx.doi.org/10.1007/978-3-642-24933-4_3>. Citations on pages 76 and 85.

ter Beek, M. H.; VINK, E. P. de; WILLEMSE, T. A. C. Family-based model checking with mcrl2. In: ____. **Fundamental Approaches to Software Engineering: 20th International Conference, FASE 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017. p. 387–405. ISBN 978-3-662-54494-5. Available: <http://dx.doi.org/10.1007/978-3-662-54494-5_23>. Citations on pages 22, 25, and 83.

THÜM, T.; APEL, S.; KäSTNER, C.; SCHAEFER, I.; SAAKE, G. A classification and survey of analysis strategies for software product lines. **ACM Comput. Surv.**, ACM, New York, NY, USA, v. 47, n. 1, p. 6:1–6:45, Jun. 2014. ISSN 0360-0300. Available: <http://doi.acm.org/10.1145/2580950>. Citations on pages 22, 23, 24, 69, 73, 83, and 84.

THÜM, T.; KÄSTNER, C.; BENDUHN, F.; MEINICKE, J.; SAAKE, G.; LEICH, T. Featureide: An extensible framework for feature-oriented software development. **Science of Computer Programming**, v. 79, n. Supplement C, p. 70 – 85, 2014. ISSN 0167-6423. Experimental Software and Toolkits (EST 4): A special issue of the Workshop on Academic Software Development Tools and Techniques (WASDeTT-3 2010). Available: <http://www.sciencedirect.com/science/article/pii/S0167642312001128>. Citation on page 77.

TILLMANN, N.; HALLEUX, J. D. Pex: White box test generation for .net. In: **Proceedings of the 2Nd International Conference on Tests and Proofs**. Berlin, Heidelberg: Springer-Verlag, 2008. (TAP'08), p. 134–153. ISBN 3-540-79123-X, 978-3-540-79123-2. Available: <http://dl.acm.org/citation.cfm?id=1792786.1792798>. Citation on page 52.

TILLMANN, N.; SCHULTE, W. Unit tests reloaded: Parameterized unit testing with symbolic execution. **IEEE Softw.**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 23, n. 4, p. 38–47, Jul. 2006. ISSN 0740-7459. Available: <http://dx.doi.org/10.1109/MS.2006.117>. Citation on page 52.

Tomte. **Tomte: Bridging the Gap between Active Learning and Real-world Systems**. 2017. <http://tomte.cs.ru.nl>. [Online; accessed 17-Out-2017]. Citations on pages 67 and 85.

TRETMANS, J. Model-based testing and some steps towards test-based modelling. In: ____. **Formal Methods for Eternal Networked Software Systems: 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM 2011, Bertinoro, Italy, June 13-18, 2011. Advanced Lectures**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. p. 297–326. ISBN 978-3-642-21455-4. Available: <https://doi.org/10.1007/978-3-642-21455-4_9>. Citation on page 56.

UTTING, M.; PRETSCHNER, A.; LEGEARD, B. A taxonomy of model-based testing approaches. **Software Testing, Verification and Reliability**, John Wiley & Sons, Ltd, v. 22, n. 5, p. 297–312, 2012. ISSN 1099-1689. Citations on pages 22, 25, 31, 41, 47, 55, 63, and 83.

VAANDRAGER, F. Model learning. **Commun. ACM**, ACM, New York, NY, USA, v. 60, n. 2, p. 86–95, Jan. 2017. ISSN 0001-0782. Available: <http://doi.acm.org/10.1145/2967606>. Citations on pages 23, 55, 67, and 83.

VARSHOSAZ, M.; BEOHAR, H.; MOUSAVI, M. R. Delta-oriented fsm-based testing. In: ____. **Formal Methods and Software Engineering: 17th International Conference on Formal Engineering Methods, ICFEM 2015, Paris, France, November 3-5, 2015, Proceedings**. Cham: Springer International Publishing, 2015. p. 366–381. ISBN 978-3-319-25423-4. Available: <http://dx.doi.org/10.1007/978-3-319-25423-4_24>. Citations on pages 76 and 78.

VASILEVSKII, M. P. Failure diagnosis of automata. **Cybernetics**, v. 9, n. 4, p. 653–665, 1973. ISSN 1573-8337. Available: <http://dx.doi.org/10.1007/BF01068590>. Citations on pages 22 and 45.

VINCENZI, A.; DELAMARO, M.; HÖHN, E.; MALDONADO, J. C. Functional, control and data flow, and mutation testing: Theory and practice. In: ____. **Testing Techniques in Software Engineering: Second Pernambuco Summer School on Software Engineering, PSSE 2007, Recife, Brazil, December 3-7, 2007, Revised Lectures**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. p. 18–58. ISBN 978-3-642-14335-9. Available: <https://doi.org/10.1007/978-3-642-14335-9_2>. Citations on pages 33, 34, 35, 37, and 38.

VOLPATO, M.; TRETMANS, J. Approximate active learning of nondeterministic input output transition systems. **ECEASST**, v. 72, 2015. Available: <http://journal.ub.tu-berlin.de/eceasst/article/view/1008>. Citation on page 56.

WALKINSHAW, N. Chapter 1 - reverse-engineering software behavior. In: MEMON, A. (Ed.). **Advances in Computers**. Elsevier, 2013, (Advances in Computers, Supplement C). p. 1 – 58. Available: <http://www.sciencedirect.com/science/article/pii/B978012408089800001X>. Citations on pages 24 and 84.

WALKINSHAW, N.; TAYLOR, R.; DERRICK, J. Inferring extended finite state machine models from software executions. **Empirical Software Engineering**, v. 21, n. 3, p. 811–853, 2016. ISSN 1573-7616. Available: <http://dx.doi.org/10.1007/s10664-015-9367-7>. Citation on page 26.

WEIB̈LEDER, S.; SOKENOU, D.; SCHLINGLOFF, H. Reusing state machines for automatic test generation in product lines. In: Thomas Bauer, Hajo Eichler, Axel Rennoch (Ed.). **MoTiP '08: Model-Based Testing in Practice**. [S.l.]: Fraunhofer IRB Verlag, 2008. Citations on pages 26 and 85.

WENDEHALS, L.; ORSO, A. Recognizing behavioral patterns atruntime using finite automata. In: **Proceedings of the 2006 International Workshop on Dynamic Systems Analysis**. New York, NY, USA: ACM, 2006. (WODA '06), p. 33–40. ISBN 1-59593-400-6. Available: <http://doi.acm.org/10.1145/1138912.1138920>. Citation on page 47.

WEYUKER, E. J. On testing non-testable programs. **The Computer Journal**, v. 25, n. 4, p. 465–470, 1982. Available: <+http://dx.doi.org/10.1093/comjnl/25.4.465>. Citation on page 32.

____. Assessing test data adequacy through program inference. **ACM Trans. Program. Lang. Syst.**, ACM, New York, NY, USA, v. 5, n. 4, p. 641–655, Oct. 1983. ISSN 0164-0925. Available: <http://doi.acm.org/10.1145/69575.357231>. Citation on page 55.

YOO, S.; HARMAN, M. Regression testing minimization, selection and prioritization: A survey. **Softw. Test. Verif. Reliab.**, John Wiley and Sons Ltd., Chichester, UK, v. 22, n. 2, p. 67–120, Mar. 2012. ISSN 0960-0833. Available: <http://dx.doi.org/10.1002/stv.430>. Citations on pages 32, 49, and 87.

ICMC USP
SÃO CARLOS