Automated test generation for production systems with a model-based testing approach

William Durand

N° d'ordre : D.U. 2691
EDSPIC : 753

UNIVERSITÉ BLAISE PASCAL - CLERMONT-FD II
ECOLE DOCTORALE
SCIENCES POUR L'INGÉNIEUR DE CLERMONT-FD

Thèse

Présentée par

William Durand

pour obtenir le grade de

Docteur d'Université

Spécialité : Informatique

Automated Test Generation for production systems with a Model-based Testing approach

---

Soutenue publiquement le 04 mai 2016, devant le jury :

PRÉSIDENTE	
Hélène Waeselynck	Directrice de Recherche, LAAS/CNRS, Toulouse
RAPPORTEURS	
Ana Rosa Cavalli	Professeur à Télécom SudParis
Roland Groz	Professeur à l'Institut Polytechnique de Grenoble
EXAMINATEURS	
Farouk Toumani	Professeur des Universités, UBP/LIMOS, Clermont-Ferrand
Pascal Lafourcade	Maître de Conférences HDR, UBP/LIMOS, Clermont-Ferrand
DIRECTEUR DE THÈSE	
Sébastien Salva	Professeur des Universités, UDA/LIMOS, Clermont-Ferrand
INVITÉ	
Stéphane Simonnet	Architecte, Manufacture Française des Pneumatiques Michelin

Abstract

This thesis tackles the problem of testing (legacy) production systems such as those of our industrial partner Michelin, one of the three largest tire manufacturers in the world, by means of Model-based Testing. A production system is defined as a set of production machines controlled by a software, in a factory. Despite the large body of work within the field of Model-based Testing, a common issue remains the writing of models describing either the system under test or its specification. It is a tedious task that should be performed regularly in order to keep the models up to date (which is often also true for any documentation in the Industry). A second point to take into account is that production systems often run continuously and should not be disrupted, which limits the use of most of the existing classical testing techniques.

We present an approach to infer exact models from traces, *i.e.* sequences of events observed in a production environment, to address the first issue. We leverage the data exchanged among the devices and software in a black-box perspective to construct behavioral models using different techniques such as expert systems, model inference, and machine learning. It results in large, yet partial, models gathering the behaviors recorded from a system under analysis. We introduce a context-specific algorithm to reduce such models in order to make them more usable while preserving trace equivalence between the original inferred models and the reduced ones. These models can serve different purposes, *e.g.*, generating documentation, data mining, but also testing.

To address the problem of testing production systems without disturbing them, this thesis introduces an *offline passive Model-based Testing technique*, allowing to detect differences between two production systems. This technique leverages the inferred models, and relies on two implementation relations: a slightly modified version of the existing trace preorder relation, and a weaker implementation proposed to overcome the partialness of the inferred models.

Overall, the thesis presents *Autofunk*, a modular framework for model inference and testing of production systems, gathering the previous notions. Its Java implementation has been applied to different applications and production systems at Michelin, and this thesis gives results from different case studies. The prototype developed during this thesis should become a standard tool at Michelin.

Keywords. Model inference, expert system, production system, offline passive testing, conformance.

Résumé

Ce manuscrit de thèse porte sur le problème du test *basé modèle* de systèmes de production existants, tels ceux de notre partenaire industriel Michelin, l'un des trois plus grands fabricants de pneumatiques au monde. Un système de production est composé d'un ensemble de machines de production contrôlées par un ou plusieurs logiciels au sein d'un atelier dans une usine. Malgré les nombreux travaux dans le domaine du test basé modèle, l'écriture de modèles permettant de décrire un système sous test ou sa spécification reste un problème récurrent, en partie à cause de la complexité d'une telle tâche. De plus, un modèle est utile lorsqu'il est à jour par rapport à ce qu'il décrit, ce qui implique de le maintenir dans le temps. Pour autant, conserver une documentation à jour reste compliqué puisqu'il faut souvent le faire manuellement. Dans notre contexte, il est important de souligner le fait qu'un système de production fonctionne en continu et ne doit être ni arrêté ni perturbé, ce qui limite l'usage des techniques de test classiques.

Pour pallier le problème de l'écriture de modèles, nous proposons une approche pour construire automatiquement des modèles depuis des séquences d'événements observés (*traces*) dans un environnement de production. Pour se faire, nous utilisons les informations fournies par les données échangées entre les éléments qui composent un système de production. Nous adoptons une approche boîte noire et combinons les notions de système expert, inférence de modèles et *machine learning*, afin de créer des modèles comportementaux. Ces modèles inférés décrivent des comportements complets, enregistrés sur un système analysé. Ces modèles sont partiels, mais également très grands (en terme de taille), ce qui les rend difficilement utilisable par la suite. Nous proposons une technique de réduction spécifique à notre contexte qui conserve l'équivalence de traces entre les modèles de base et les modèles fortement réduits. Grâce à cela, ces modèles inférés deviennent intéressant pour la génération de documentation, la fouille de données, mais également le test.

Nous proposons une méthode passive de test basé modèle pour répondre au problème du test de systèmes de production sans interférer sur leur bon fonctionnement. Cette technique permet d'identifier des différences entre deux systèmes de production et réutilise l'inférence de modèles décrite précédemment. Nous introduisons deux relations d'implantation : une relation basée sur l'inclusion de traces, et une seconde

relation plus faible proposée, pour remédier au fait que les modèles inférés soient partiels.

Enfin, ce manuscrit de thèse présente *Autofunk*, un *framework* modulaire pour l'inférence de modèles et le test de systèmes de production qui aggrège les notions mentionnées précédemment. Son implémentation en Java a été appliquée sur différentes applications et systèmes de production chez Michelin dont les résultats sont donnés dans ce manuscrit. Le prototype développé lors de la thèse a pour vocation de devenir un outil standard chez Michelin.

Mots-clés. Inférence de modèles, système expert, système de production, test passif, conformité.

Acknowledgement

Je tiens tout d'abord à remercier Ana Rosa Cavalli et Roland Groz pour avoir pris le temps de rapporter ce manuscrit de thèse, pour leurs remarques et conseils, mais également leur présence lors de ma soutenance. Je souhaite ensuite remercier Hélène Waeselynck, Farouk Toumani, Pascal Lafourcade et Stéphane Simonnet d'avoir bien voulu examiner mon travail et assister à ma soutenance.

Merci également à Sébastien, mon directeur de thèse, pour cette collaboration pendant trois belles années. Je ne saurais jamais assez exprimer l'immense reconnaissance, respect et humilité que j'ai envers lui.

Bien entendu, cette thèse n'aurait pu se faire sans le concours de l'entreprise Michelin. Je tiens à remercier Thierry D., Ludovic L., Etienne S., Stéphane S., Yann G., Thomas B., Jean D., Pierre P., Gabriel Q., Alain L., Isabelle A., Stephen N. et l'ensemble des équipes "AI". Dans le même esprit, je tiens à remercier le laboratoire LIMOS et son directeur, Farouk Toumani, de m'avoir accueilli.

Mère, père, vous avez forcément votre place ici. Merci de m'avoir donné goût au savoir et à la réflexion, de m'avoir rendu si responsable et autonome, et de m'avoir permis d'apprendre de mes erreurs. Vous m'avez également donné cette petite soeur, Camille, qui m'a toujours supporté et que je remercie énormément ici. De peur d'oublier certain(e)s, je ne me risquerais pas à lister toutes les personnes, famille ou ami(e)s, qui font partie de ma vie mais à qui je pense forcément.

Ce document n'aurait probablement pas été celui que vous commencez à parcourir sans l'aide précieuse de Bérénice B., Pascal L., Pascal B. et Jonathan P. Un grand merci à vous !

Une dernière pensée pour mes collègues, camarades et/ou amis de l'IUT de Clermont-Ferrand, du laboratoire LIMOS, de l'école ISIMA et, plus généralement, à toutes les personnes que j'ai pu rencontré et qui m'ont, d'une manière ou d'une autre, sciemment ou non, façonné.

Contents

List of Definitions

List of Figures

List of Tables

1 Introduction

Contents

1.1 General context and motivations

Almost a decade ago, *quality assurance* (QA) was not a common practice in most software companies, and researchers had to prove, for instance, what the benefits of testing could be. Generally speaking, quality assurance is a way of preventing faults in manufactured products, which is defined in ISO 9000 as “part of quality management focused on providing confidence that quality requirements will be fulfilled” [ISO05]. *Software testing* provides information about the quality of software. For instance, “testing can prove confidence of a software system by executing a program with the intent of finding errors” [Mye79], also known as "bugs".

Nowadays, quality assurance and software testing are well-known in the Industry, and everyone understands the need for them. Yet, testing is often performed by hand, which is complicated and far from perfect. Software testing is often seen as a sequence of three major steps: “(i) the design of test cases that are good at revealing faults, according to a certain level of requirement [Kan03], (ii) the execution of these test cases, and (iii) the determination of whether the produced output is correct [Wey82]” [Lak09].

Sadly, the test case execution is often the only fully automated aspect of this activity in the Industry. *Continuous Integration* (CI) [Boo91] is now associated with the automation of the execution of test cases and quick feedback, often received by email. There are countless tools and services [1] to automate this process, but too few tools have emerged to tackle the problem of the automatic test case generation. Fortunately, in Academia, researchers have studied such a problem for decades [Ber07].

[1] For example, Travis CI: `https://travis-ci.org/`.

A relatively recent field to automate and improve testing is *Model-based Testing* (MbT). While the original idea has been around for decades [Moo56], there has been a growing interest over the last years. MbT is application of (formal) Model-based design for designing and optionally also executing artifacts to perform software testing [Jor95]. The use of a model allows to formally describe the expected behaviors of a software, from which it is possible to automatically generate test cases, and then to execute them. Nonetheless, writing such models is tedious and error-prone, which is a drawback and can also explain the slow adoption of MbT in the Industry.

Model inference is a research field that aims at automatically deriving models, expressing functional behaviors of existing software. These models, even if incomplete, help understand how a software behaves. That is why model inference is interesting to solve the limitation of MbT mentioned previously. Most of the model inference techniques that are used for testing purpose are not designed to infer large models or to test large software systems though. Anti-Model-based Testing [Ber+04] is somehow related to this idea, although Anti-Model-based Testing is more about using testing to reverse-engineer a model, and then check such a model to detect whether the software behaves correctly.

Michelin, one of the three largest tire manufacturers in the world, has been our industrial partner for the last three years. The company designs most of its factories, production machines, and software itself. In a factory, there are several workshops for the different parts of the manufacturing process. From our point of view, a workshop is seen as a set of production machines controlled by a software. That is what we call a *production system*. Such systems deal with many physical devices, databases, and also human interactions. Most of these systems run for years, up to 20 years according to our partner. Maintaining and updating such legacy software is complicated as documentation is often outdated, and the developers who wrote these software are not available anymore. In this thesis, we propose a solution to the problem of testing such systems called *Autofunk*, a framework (and a tool) combining different techniques to: (i) infer formal models from large (legacy) production systems, and (ii) perform Model-based Testing (using the inferred models) in order to detect potential regressions.

The next section discusses in detail the problems this thesis addresses. Section 1.3 presents the contributions of this thesis. Finally, Section 1.4 gives the overview of this thesis.

1.2 Problems and objectives

Michelin is a tire manufacturer since 1889. With a strong but old culture of secret, the company uses to design most of its factories, production machines, and software itself. The engineering department is dedicated to the construction of these items, and gathers embedded software engineers, control engineers, mechanical engineers, and also software engineers. The computing sub-department deals with software that have been built for decades, and we can count about 50 different applications, and even more when we consider the different versions with customization made for the different factories. For the record, Michelin owns factories in more than 70 countries.

Different programming languages have been used for building such software, as well as different development frameworks, and various paradigms. This context leads to a large and disparate set of legacy applications that have to be maintained, but also updated with new features. As most of these software are built for controlling and supervising a set of production machines in a factory, they are considered critical in the sense that they can break the production flow. The main issue faced by the software engineers is the introduction of regressions. A *regression* is a software fault that makes a feature stop functioning as intended after a certain event, for instance, the deployment of a new version of a software. Regressions have to be revealed as soon as possible in order to avoid production downtimes.

The goal of this thesis is to propose technical solutions to Michelin's engineers to prevent such regressions with testing, *i.e.* performing *regression testing* on Michelin's production systems based on behavioral models of such systems. Such models have to be up to date, hence we cannot rely on existing documentation. We determine, by means of Model-based Testing, whether a change in one part of a software affects other parts of it. To avoid disturbing the system, we use a *passive testing* approach, which only observes the production system under test and does not interact with it. The main assumption in this work is that we choose to consider a running production system in a production environment (that is, a factory) as a fully operational system, *i.e.* a system that behaves correctly. Such a hypothesis has been expressed by our partner Michelin. The inferred models can be employed for different purposes, *e.g.*, documentation and testing, but it is manifest that they should not be used for conformance testing in general. Last but not least, we propose scalable techniques because production systems exchange thousands *production events* a day.

In this thesis, we provide solutions for making Michelin's production systems more reliable by means of testing, to ease software engineers' work, thanks to two main lines:

- The inference of partial yet exact models of production systems in a fast and efficient manner, based on the data exchanged in a (production) environment. A method to extract knowledge from data available in a production environment has to be defined before inferring models;

- The design of a conformance testing technique based on the inferred models, targeting production systems. It is worth mentioning that it is possible to perform conformance testing here thanks to the assumption introduced previously. Finally, due to the nature of these systems, the testing method should scale.

1.3 Contributions of the thesis

The contributions of this thesis are:

1. A preliminary study introducing our model inference framework, called *Autofunk (v1)*, on web applications. This framework is framed upon a rule-based expert system capturing knowledge of human business experts. *Autofunk* is also combined with an automatic testing technique to interact with web applications in order to improve the completeness of the inferred models. Several models can be built at different levels of abstraction, allowing to create, for instance, human-readable documentation. This preliminary work targeting web applications validates our ideas and concepts to infer models from data recorded in a (production) environment;

2. *Autofunk (v2)*, the enhanced version of our framework, now combining several techniques originating from different fields such as expert systems, machine learning and model inference, for inferring formal models of legacy software systems. Our main goal is to infer models of Michelin's production systems with a *black-box* approach. We choose to leverage the data exchanged among the devices and software in order to target most of the existing Michelin applications without having to consider the programming languages or frameworks used to develop any of these applications. *Autofunk*'s modular architecture allows multiple extensions so that we can take different data sources as input, and perform various tasks based on the inferred models, *e.g.*, testing.

3. An evaluation of the model inference of Michelin's production systems with *Autofunk*, proving that it is able to build exact models in an efficient manner, based on large sets of traces. For now, we define a set of traces as information collected from a production system, and describing its behaviors. A more formal definition is given in the sequel of this thesis. Exactness of our inferred models is defined by the trace-equivalence and trace-inclusion relations

proposed in [PY06]. Results show that our model inference technique scales well;

4. A reduction technique for symbolic transition systems that is both fast and efficient, keeping the exactness of the models, and targeting large models. This technique has successfully been applied to large Michelin's production systems. This context-specific reduction technique is a key element to enable testing of Michelin's production systems because it improves the usability of the inferred models, which also favors the adoption of *Autofunk* at Michelin;

5. An offline passive testing technique leveraging the inferred models for testing production systems, along with a case study. This work is an extension of *Autofunk* introducing two implementation relations: (i) the first relation refers to the trace preorder relation [DNH84], and (ii) the second relation is used to overcome a limitation caused by the partialness of the inferred models. This testing extension detects differences between two production systems, *i.e.* potential regressions.

Publications. Most of these contributions have been published in the following international conference proceedings:

- *Actes de la 13eme édition d'AFADL, atelier francophone sur les Approches Formelles dans l'Assistance au Développement de Logiciels* (AFADL'14) [DS14a];
- Proceedings of the Fifth Symposium on Information and Communication Technology (SoICT'14) [DS14b];
- Proceedings of Formal Methods 2015 (FM'15) [DS15a];
- Proceedings of the 9th International Conference on Distributed Event-Based Systems (DEBS'15) [SD15];
- Proceedings of the 13th International Conference on Formal Methods and Models for Co-Design (MEMOCODE'15) [DS15b].

1.4 Overview of the thesis

The work presented in this thesis deals with two main research realms: (i) (software) model inference, and (ii) software testing. Hence, Chapter 2 is split into two main sections: Section 2.1 introduces important notions of software testing, and Section 2.2 reviews the literature on software model inference techniques. Chapters 3 and 4 are dedicated to the model inference of software systems (web applications first,

and production systems then). Chapter 5 is dedicated to the testing of production systems. Chapter 6 ends this thesis with conclusions and perspectives for future work. We give more details about each chapter below.

Chapter 2 surveys the literature in software testing first, and then in model inference applied to software systems. The chapter starts by introducing what software testing means, mentioning some important notions as well as the different types of testing. It then presents what Model-based Testing (MbT) is, along with a few definitions and common terms employed in MbT. The software testing part ends with a review on some passive testing techniques, and why they are interesting in our case. The second part of this chapter presents what model inference is, from active inference, *i.e.* methods interacting with a software system to extract knowledge about it, to passive inference, *i.e.* techniques that infer models from a fixed set of knowledge, such as a set of execution traces, source code, or even documentation.

Chapter 3 presents our work on model inference based on the remarks made in Chapter 2. This chapter introduces a preliminary work on model inference of web applications that gave birth to *Autofunk*, our modular framework for inferring models (and later, performing testing). This chapter gives an overview of *Autofunk*'s very first architecture, called *Autofunk v1*. It then presents how *Autofunk* relies on an automatic testing technique to extract more knowledge about the targeted web applications. This is important to infer more complete models. A note on its implementation is given, following by an experimentation. This work has been published in *Actes de la 13eme édition d'AFADL, atelier francophone sur les Approches Formelles dans l'Assistance au Développement de Logiciels* (AFADL'14) [DS14a], and in the Proceedings of the Fifth Symposium on Information and Communication Technology (SoICT'14) [DS14b].

Chapter 4 introduces our framework *Autofunk* revisited to target production systems. This chapter gives the context that led to our choices regarding the design of *Autofunk v2*. Our *reduction technique* that heavily reduces models is then presented, along with the results of experiments on Michelin's production systems. A whole section is dedicated to the implementation of *Autofunk* for Michelin. Last significant part of this chapter is the use of a machine learning technique to maximize one step of our model inference technique, which led to the creation of *Autofunk v3*. This work has been published in the Proceedings of Formal Methods 2015 (FM'15) [DS15a], and in the Proceedings of the 9th International Conference on Distributed Event-Based Systems (DEBS'15) [SD15].

Chapter 5 tackles the problem of passively testing production systems, without disturbing them, and without having any specification. It presents our work on offline passive testing, by extending *Autofunk v3*'s model inference framework. After

having presented the overall idea, our passive algorithm is given and explained. The results of experiments on Michelin's production systems are also given. This work has been published in the Proceedings of the 13th International Conference on Formal Methods and Models for Co-Design (MEMOCODE'15) [DS15b].

Chapter 6 closes the main body of the thesis with concluding comments and proposals for future work.

State of the art

2

In this chapter, we survey the literature in software testing first, and then in model inference applied to software systems. We start by introducing what software testing means, mentioning some important notions as well as the different types of testing. We then present what Model-based Testing (MbT) is, along with a few definitions and common terms employed in MbT. The software testing part ends with a review on some passive testing techniques, and why they are interesting in our case. The second part of this chapter presents what model inference is, from active inference, *i.e.* methods interacting with a software system to extract knowledge about it, to passive inference, *i.e.* techniques that infer models from a fixed set of knowledge, such as a set of execution traces, source code, or even documentation.

Contents

2.1 Software testing

"Software testing is the process of executing a program or system with the intent of finding errors" [Mye79]. Indeed, "testing shows the presence, not the absence of *bugs*", *i.e.* faults or errors, as Edsger Wybe Dijkstra used to say [BR70].

Testing is achieved by analyzing a software to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of this software. As Myers explained in [Mye79], testing is used to find faults, but it is also useful to provide confidence of reliability, correctness, and absence of particular faults on software we develop. This does not mean that the software is completely free of defects. Rather, it must be good enough for its intended use. Testing is a *verification and validation* (V&V) process [WF89]. One uses to (informally) explain both terms with the following questions [Boh79]:

- **Validation:** "are we building the right software?"
- **Verification:** "are we building the software right?"

In other words, formal verification is the act of proving or disproving the correctness of intended algorithms underlying a system with respect to a certain formal specification or property, using formal methods of mathematics. Model checking [Cla+99], runtime verification [LS09], theorem proving [Fit12], static analysis [LA+00], and simulation are all verification methods.

Validation is testing as the act of revealing bugs. That is what most people think testing is, and also the meaning we give to the word "testing" in the sequel of this thesis. Testing involves a (software) *System Under Test* (SUT). The prevailing intuition of testing is reflected in its operational characterization as an activity in which a *tester* first generates and sends stimuli, *i.e. test input* data, to a system under test in order to observe phenomena (mainly behaviors). Such phenomena can be represented by the existence of *test outputs* for instance. We then have to decide on a suitable *verdict*, which expresses the assessment made. The two most well-known verdicts are $Pass$ and $Fail$. We call *Test Case* (TC), a structure that is compound of *test data*, *i.e.* inputs which have been devised to test the system, an expected behavior, and an expected output [ISO10b]. A set of test cases is called a *Test Suite* (TS).

Such an intuition refers to the ***active testing*** methodology, *i.e.* when the system under test is stimulated. Most of the classical testing techniques and tools found in the Industry tend to perform active testing, *e.g.*, the *xUnit* frameworks [1]. On the

[1] `http://www.martinfowler.com/bliki/Xunit.html`

contrary, in ***passive testing***, the tester does not interact with the system under test, it only observes. For instance, a *monitor* can be used to collect the execution traces of a (running) system under test. We define the term *trace* (or *execution trace*) as a finite sequence of observations made on a software system.

In the following section, we introduce the software testing realm. In Section 2.1.2, we focus on (active) Model-based Testing along with some definitions used in the rest of this thesis. We present what passive testing is in Section 2.1.3.

2.1.1 Types of testing

Nowadays, software testing, if not always applied, is well-known in the Industry. It is considered a good practice and many techniques and tools have been developed over the last 10 years. Most of them are different in nature and have different purposes. In both Academia and the Industry, there are a lot of terms that all end with "Testing" such as: Unit Testing, Integration Testing, Functional Testing, System Testing, Stress Testing, Performance Testing, Usability Testing [DR99; TR03], Acceptance Testing, Regression Testing [LW89; Won+97], Beta Testing, and so on.

All these terms refer to different testing practices that can be sorted in three different manners as depicted in Figure 2.1: by *aspect*, by *phase*, and/or by *accessibility*.

First, ISO 9126 [ISO01], replaced by ISO/IEC 25010:2011 [ISO10a], provides six characteristics of quality (also called aspects) that can be used to sort these testing techniques into six testing types:

- **Efficiency testing:** the capability of the software product to provide appropriate performance, related to the amount of resources used, under stated conditions [ISO01];
- **Functionality testing:** the capability of the software product to provide functions which meet stated and implied needs when the software is used under specified conditions [ISO01];
- **Maintainability testing:** the capability of the software product to be modified. Modifications may include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications [ISO01];
- **Portability testing:** the capability of the software product to be transferred from one environment to another [ISO01];

- **Reliability testing:** the capability of the software product to maintain a specified level of performance when used under specified conditions [ISO01];

- **Usability testing:** capability of the software product to be understood, learned, used and attractive to the user, when used under specified conditions [ISO01].

ISO/IEC 25010:2011 [ISO10a], which replaced ISO 9126 [ISO01], counts eight product quality characteristics, including two new categories:

- **Security:** the degree to which a product or system protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization [ISO10a];

- **Compatibility:** the degree to which a product, system or component can exchange information with other products, systems or components, and/or perform its required functions, while sharing the same hardware or software environment [ISO10a].

Yet, these classifications can not be generally accepted as single or complete. In [BD04], authors suggested to classify the different techniques based on the target of the test (sometimes we refer to this classification as level of detail or phase as shown in Figure 2.1):

- **Unit testing:** individual units (*e.g.*, functions, methods, modules) or groups of related units of the system are tested in isolation [IEE90]. Typically this testing type implies access to the source code being tested;

- **Integration testing:** interactions between software components are tested [IEE90]. This testing type is a continuous task, hence the continuous integration (CI) practice;

- **System Testing:** the whole system is taken into consideration to evaluate the system's compliance with its specified requirements [IEE90]. This testing type is also appropriate for validating not-only-functional requirements, such as security, performance (speed), accuracy, and reliability (fault tolerance).

These requirements are sometimes seen as *objectives* of testing, leading to even more different testing types as listed before. There are also different test approaches [ISO10b] (or perspectives) to perform testing, depending on the information available, for instance, to construct the test cases, *i.e.* accessibility:

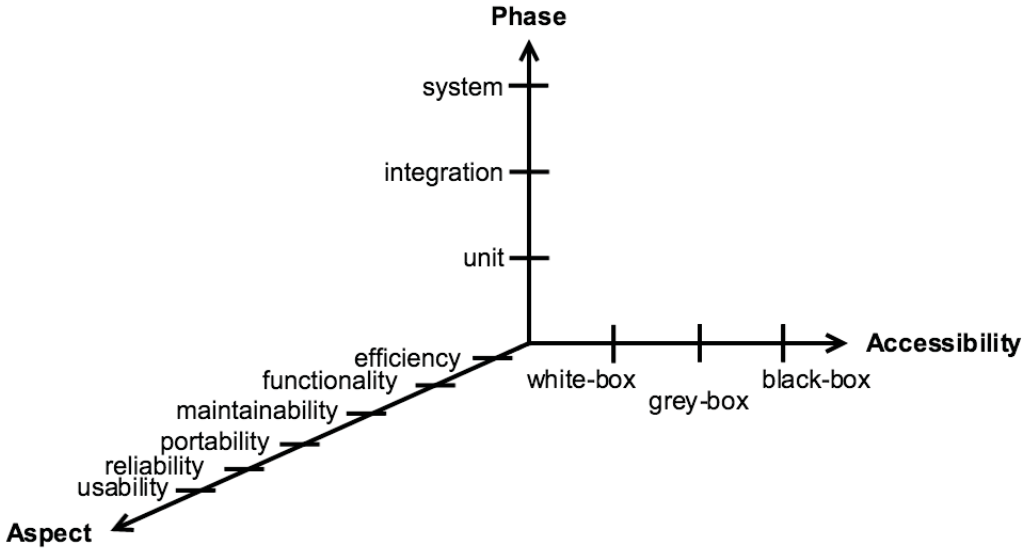

Fig. 2.1: Sorts of testing. We can sort testing techniques by aspect (characteristics of quality), by phase (related to the target of the test), and by accessibility (related to the information available, *e.g.*, to construct the test cases).

- **White-box testing:** (also known as *glass box* [ISO10b]) a method that tests the internal structure of a SUT. It is usually done at the unit level. This technique is also known as *structural testing* [ISO10b];

- **Black-box testing:** a method that tests the functionalities of a SUT without knowing its internal structure. The behavior of the *implementation under test* (IUT) is only visible through a restricted interface called *Points of Control and Observation* (PCOs). Such a technique is often known as functional testing [ISO10b];

- **Grey-box testing:** the combination of white-box testing and black-box testing. One has access to the relevant parts of a SUT such as limited knowledge of the internal part of the SUT, and also knowledge of its fundamental aspects [KK12]. This technique is sometimes called *translucent testing*.

In active testing, independently of the sort of testing one chooses, a common problem is to determine a relevant and efficient set of test cases. Because testing cannot guarantee the absence of faults, a challenge is to select subset of test cases from all possible test cases with a high chance of detecting most faults. It is especially the case for regression testing since it is usually an expensive process [RH97; Gra+01]. We refer to this choice as the *test selection*, which we present below.

Test selection

A lot of research on *test selection* (or strategies) has been done, and there are numerous existing methods. For instance, *Combinatorial Testing* (also known as *Pairwise*) [TL98] is based on the observation that most faults are caused by interactions of at most two factors. Here we test all the possible discrete combinations of the parameters involved. Even though *Pairwise Testing* is often used with black-box approaches, it has been adapted for white-box, *e.g.*, in [Kim+07].

In addition, and because we cannot test all the possible input domain values for practical reasons, *Equivalence Partitioning* [HP13] is a technique that divides the test input data into a range of values, and selects one input value from each range. Similarly, *Boundary Value Analysis* [Ram03] is used to find the errors at boundaries of input domain rather than finding those errors in the center of input. On the contrary, *Fuzz Testing* also known as *Random Testing* [DN81; God+08] is a method that applies random mutations to well-formed inputs of a program, and test the resulting values. Even if it is often used in a white-box context, Random Testing can also be applied using a black-box approach, *e.g.*, [BM83]. On the other hand, *Statistical Testing* [Wal+95] is a technique where test data is generated by sampling from a probability distribution chosen so that each element of the software's structure is exercised with a high probability.

We can also mention *Functional Coverage* (also known as *Inductive Testing*) [Wal+10] where a test set is good enough when it achieves a given level of code coverage, but also all techniques related to the code structure, such as *Statement Testing*, *Path Testing*, *Branch Testing*, *Condition Testing*, *Multiple Condition (MC) Testing*, and *Loop Testing*. Another strategy acts on the source code by mutating it, *i.e.* seeding the implementation with a fault by applying a mutation operator, and then determining whether testing identifies this fault. This is known as *Mutation Testing* [Ham77].

While the Industry created many different testing tools (often originating from Academia), they mostly perform testing by hand. Researchers in software testing have worked for decades on automatic test generation. *Automatic testing* is, for instance, one way to automate white-box approaches [TH08], but there are many other techniques. On the contrary, *Model-based Testing* (MbT) [Jor95] is one research area that tries to automate the testing phase from models. In this thesis, we are interested in making production systems more reliable by means of Model-based Testing.

2.1.2 Model-based Testing

Model-based Testing (MbT) is application of Model-based design for designing and optionally also executing artifacts to perform software testing [Jor95]. *Models* can be used to represent the desired behavior of an SUT, or to represent testing strategies and a test environment. Model-based Testing can be summarized as a three-step process, which is extended in Figure 2.2:

1. Formally modeling the requirements (specification). This is usually done by humans (#1 in Figure 2.2), yet feedback about the requirements can be obtained from the model to ease the process (#2);

2. Generating the test cases from the model (#3 in Figure 2.2);

3. Running these test cases against the SUT, and evaluating the results. The test cases provide the information to control (#4 in Figure 2.2) the implementation. The latter yields outputs that are observed by the tester (#5). The results allow to issue a verdict (#6), which provides feedback about the implementation (#7). Verdicts may also indicate that a mistake was made when creating the model or that the requirements were wrong in the first place (#8).

What is a model?

Generally speaking, a model is a representation of a "thing" that allows for investigation of its properties, and most of the time, a model hides the complexity of the item it represents. In the software engineering field, models help describe software systems in order to: (i) ease the process of studying them, (ii) leverage them to build tools or generate documentation, or (iii) reveal defects (validation or verification).

Such models usually describe the behaviors of the software being modeled, and can also be known as *specifications*, helping understand and predict its behavior. In the Industry, we often encounter a "specifications phase", as in the V-model [Roo86; MM10], that is done before the "coding phase". Yet, the term "specification" refers to “a detailed formulation [...], which provides a definitive description of a system for the purpose of developing or validating the system” [ISO10b]. There are numerous models for expressing software systems, and each describes different aspects of software. For example, control flow, data flow, and program dependency graphs express how the implementation behaves by representing its source code structure. It is worth mentioning that a *partial* model can be effective, *i.e.* a model does not have to describe all behaviors of a software system to be usable, which also means that an implementation can have more features than those expressed in its specification.

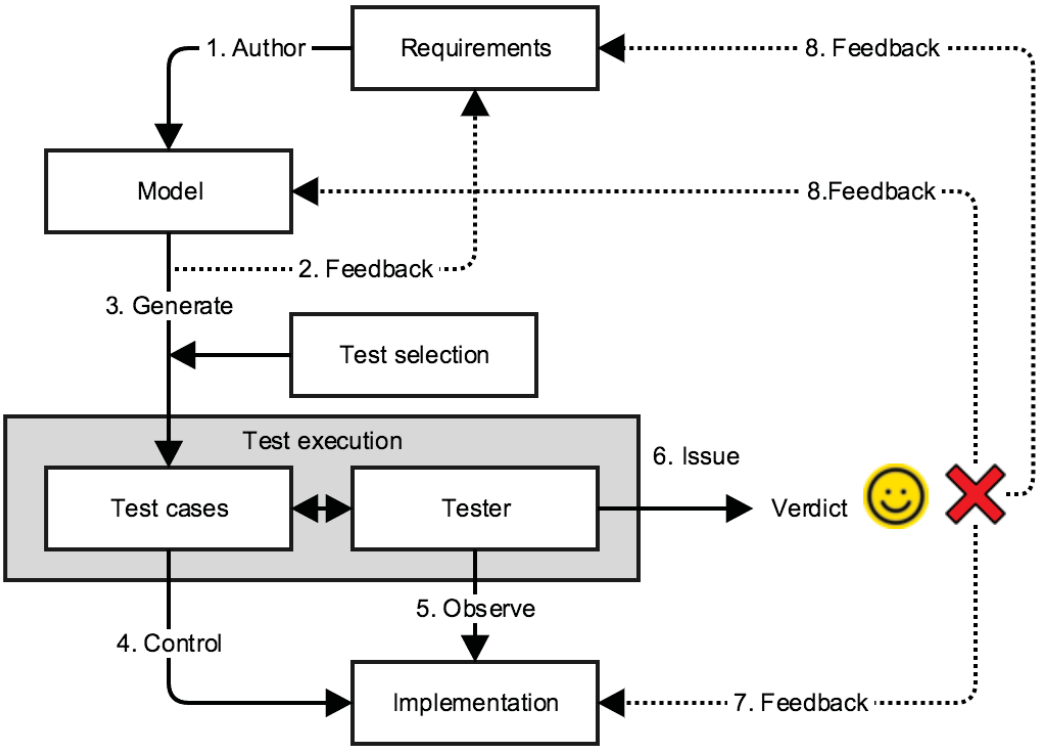

Fig. 2.2: A simplified diagram showing how MBT works. This is a three-step process: (i) we model the requirements (1 and 2), (ii) we generate the test cases (3), and (iii) we run the test cases (4 and 5) that produce verdicts (6), which we can evaluate (7 and 8). Dotted arrows represent feedback, not "actions".

We classify formal models that can be employed for testing into two categories (these lists are not exhaustive):

- **Behavior/Control oriented:** *Finite Automata* (like *Finite State Machines*, *Symbolic Transition Systems*, and *Labeled Transition Systems*) are well-known in software testing. They are very generic and flexible, and are a good fit when it comes to model software systems. For real-time and/or safety-critical systems, we often rely on synchronous languages such as *Lustre* [Hal+91] and *SCADE* [LS11]. We can also mention *Petri Nets*, and *Timed Automata* [AD94];

- **Data oriented (pre/post):** often using annotation languages originating from the *Design-By-Contract* paradigm [Mey92]. These languages make it possible to express formal properties (invariants, pre/post-conditions) that directly annotate program entities (such as classes, methods, attributes) in the source code. Many annotation languages exist, such as the *Java Modeling Language* (JML) [Lea+99], *Spec#* [Bar+11], the *Object Constraint Language* (OCL) [WK99], the *B Language and Method* [Lan96], and *Praspel* [End+11].

In [SS97], Sommerville and Sawyer give some guidelines for choosing a model for software requirements. The choice of a model depends on many factors, such as aspects of the system under test or the testing goals.

Below, we introduce a few definitions of formal models that we use in this thesis. We mainly work with behavior-oriented models based on finite automata since they are particularly suitable for modeling both web applications and production systems behaviors.

Model definitions

Labeled Transition Systems. A *Labeled Transition System* (LTS) [Mil80] is a model compound of *states* and *transitions* labeled with actions. The states model the system states and the labeled transitions model the actions that a system performs to change its state. We give the definition of the LTS model below, but we refer to [Tre96b; Tre08] for a more detailed description.

Definition 1 (Labeled Transition System) *A Labeled Transition System (LTS) is a 4-tuple* $< Q, L, T, q_0 >$ *where:*

- Q *is a countable, non-empty set of states;*
- L *is a countable set of labels;*
- $T \subseteq Q \times (L \cup \{\tau\}) \times Q$*, with* $\tau \notin L$*, is the transition relation;*
- $q_0 \in Q$ *is the initial state.*

We write $q \xrightarrow{t} q'$ *if there is a transition labeled* t *from state* q *to state* q'*, i.e.* $(q, t, q') \in T$*.*

The labels in L *represent the observable actions of a system, i.e. the interactions of the system with its environment. Internal actions are denoted by the special label* $\tau \notin L$*. Both* τ *and states are assumed to be unobservable for the environment.*

The class of all labeled transition systems over L *is denoted by* $\mathcal{LTS}(L)$ *[Tre96b].*

Example 2.1.1 Figure 2.3 presents the Labeled Transition System $lts_{machine}$ representing a coffee machine. There is a first label representing a button interaction ($button$), and two other labels for coffee ($coffee$) and tea (tea). It is represented as a graph where nodes represent states, and labeled edges represent transitions.

We have $lts_{machine} =< \{S1, S2, S3, S4\}, \{button, coffee, tea\}, \{< S1, button, S2 >, < S2, coffee, S3 >, < S2, tea, S4 >\}, S1 >$, and we can write $S1 \xrightarrow{button} S2$, and also $S1 \xrightarrow{button \cdot coffee} S3$, but $S1 \not\xrightarrow{button \cdot tea} S3$.

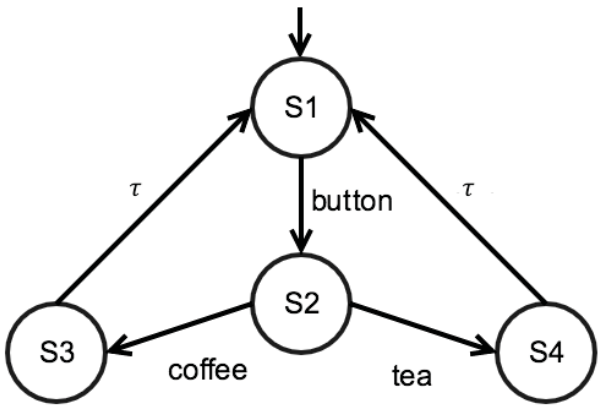

Fig. 2.3: An example of a Labeled Transition System representing a coffee machine and its internal actions (τ).

Based on this definition, we give the formal definition of a *trace*:

Definition 2 (Trace) *A* trace *is a finite sequence of observable actions. The set of all traces over L is denoted by L^*, with ϵ denoting the empty sequence. We have $q \stackrel{\sigma}{\Rightarrow} =_{def} \exists\, q' : q \stackrel{\sigma}{\Rightarrow} q'$ with $q, q' \in Q$ and $\sigma \in L^*$. We also have $Traces(p) =_{def} \{\sigma \in L^* \mid p \stackrel{\sigma}{\Rightarrow}\}$ with p being a state.*

Symbolic Transition Systems. *Symbolic Transition Systems* (STSs) [HL95] extend on LTSs by incorporating the notion of data and data-dependent control flow. The use of symbolic variables helps describe infinite state machines in a finite manner. This potentially infinite behavior is represented by the semantics of a *Symbolic Transition System* (STS), given in terms of LTS. We give some definitions related to the STS model below, but we refer to [Fra+05] for a more detailed description.

Definition 3 (Variable assignment) *We assume that there exists a domain of values denoted by D, and a variable set X taking values in D. The* variable assignment *(also called* valuation *of variables in $Y \subseteq X$ to elements of D is denoted by the function $\alpha : Y \to D$. $\alpha(x)$ denotes the assignment of the variable x to a value in D. The* empty variable assignment *is denoted by $v_\emptyset$.*

We denote by D_Y the set of all variable assignments over Y: $D_Y = \{\alpha : Y \to D \mid v$ is a variable assignment of $Y\}$. We also denote by id_Y the identity assignment over Y: $\forall x \in Y, id_Y(x) = x$.

Finally, the satisfaction *of a first order formula [HR04] (*i.e. *a guard) ϕ with respect to a given variable assignment α is denoted by $\alpha \models \phi$.*

Definition 4 (Symbolic Transition System) *A Symbolic Transition System (STS) consists of* locations *and* transitions *between locations. Locations can be seen as symbolic states. A STS is defined as a tuple* $< L, l0, V, V0, I, \Lambda, \rightarrow >$ *where:*

- L *is a countable set of locations;*
- $l0 \in L$ *is the initial location;*
- V *is a finite set of location (or internal) variables.* D_v *denotes the domain of the variable* v*;*
- $V0$ *is* an *initialization of the location variables* V*;*
- I *is a finite set of parameters (also known as* interaction variables*), disjoint from* V*;*
- Λ *is a finite set of symbolic actions* $a(p)$ *with* a *a symbol (or action), and* $p = (p_1, \ldots, p_k)$ *a finite set of parameters in* $I^k (k \in \mathbb{N})$*;*
- $\rightarrow$ *is a finite set of symbolic transitions. A symbolic transition* $t = (l_i, l_j, a(p), G, A) \in \rightarrow$*, from the location* $l_i \in L$ *to* $l_j \in L$*, also denoted by* $l_i \xrightarrow{a(p),G,A} l_j$*, is labeled by:*
  - *A symbolic action* $a(p) \in \Lambda$*;*
  - *A guard* $G \subseteq D_V \times D_p$ *that restricts the firing of the transition;*
  - *An assignment* $A : D_V \times D_p \rightarrow D_V$ *that defines the evolution of the variables,* A_x *being the function in* A *defining the evolution of the variable* $x \in V$*.*

For readability purpose, if A is the identity function id_V, we denote a transition by $l_i \xrightarrow{a(p),G} l_j$.

We also use the generalized transition relation $\Rightarrow$ to represent STS *paths*: $l \xRightarrow{(a_1,G_1,A_1)\ldots(a_n,G_n,A_n)} l' =_{def} \exists\, l0 \ldots l_n, l = l0 \xrightarrow{(a_1,G_1,A_1)} l_1 \ldots l_{n-1} \xrightarrow{(a_n,G_n,A_n)} l_n = l'$.

Example 2.1.2 Figure 2.4 presents the Symbolic Transition System $sts_{machine}$ representing a simple slot-machine, as in [Fra+05]. The first arrow on $L0$ indicates the initial location, and the fact that the machine starts with no money ($v = 0$). A player can insert a coin ($coin$), and win the jackpot in a non-deterministic manner (v coins are passed over parameter i of output action $tray$), or lose his money

$((i == 0))$. After that, the machine behaves as initially, but with a different amount of coins.

We have $sts_{machine} =< \{L0, L1, L2, L3\}, L0, \{v\}, \{v \mapsto 0\}, \{i\}, \{coin, tray\}, \rightarrow >$ where $\rightarrow$ is given by the directed edges between the locations in Figure 2.4. We can write $L2 \xrightarrow{tray(i),[(i==0)]} L0$.

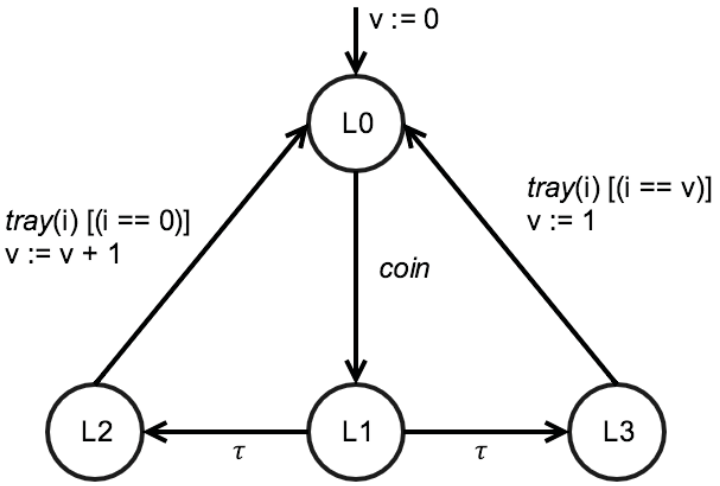

Fig. 2.4: An example of Symbolic Transition System representing a simple slot-machine.

Labeled Transition System semantics. A STS is associated with a Labeled Transition System to formulate its *semantics* [Fra+05]. A LTS semantics corresponds to a valued automaton without symbolic variables, which is often infinite: the LTS states are labeled by internal variable assignments while transitions are labeled by actions combined with parameter assignments.

Definition 5 (Labeled Transition System semantics) *As defined in [Fra+05], the semantics of a STS* $\mathcal{S} =< L, l0, V, V0, I, \Lambda, \rightarrow >$ *is the LTS* $||\mathcal{S}|| =< Q, q_0, \sum, \rightarrow >$ *where:*

- $Q = L \times D_V$ *is a finite set of states;*
- $q_0 = (l0, V0)$ *is the initial state;*
- $\sum = \{(a(p), \alpha) \mid a(p) \in \Lambda, \alpha \in D_p\}$ *is the set of* valued actions*;*
- $\rightarrow$ *is the transition relation* $Q \times \Sigma \times Q$ *defined by the following rule:*

$$\frac{l_1 \xrightarrow{a(p),G,A} l_2,\ \alpha \in D_p,\ v \in D_V, v' \in D_V, v \cup \alpha \models G, v' = A(v \cup \alpha)}{(l_1, v) \xrightarrow{a(p),\alpha} (l_2, v')}$$

The rule above can be read as follows: for a STS transition $l_1 \xrightarrow{a(p),G,A} l_2$, we obtain a LTS transition $(l_1, v) \xrightarrow{a(p),\alpha} (l_2, v')$ with v a variable assignment over the internal variable set if there exists an assignment α such that the guard G evaluates to true with $v \cup \alpha$. Once the transition is executed, the internal variables are assigned with v' derived from the assignment $A(v \cup \alpha)$.

Runs and *traces*, which represent *executions* and *event sequences*, can also be derived from LTS semantics [Jér06]:

Definition 6 (Run and trace) *Given a STS* $\mathcal{S} = < L, l0, V, V0, I, \Lambda, \rightarrow >$*, interpreted by its LTS semantics* $||\mathcal{S}|| = < Q, q_0, \sum, \rightarrow >$*, a* run $q_0 \cdot (a_1(p), \alpha_1) \cdot \cdots \cdot (a_{n-1}(p), \alpha_{n-1}) \cdot q_n$ *is an alternate finite sequence of states and valued actions, concatenated with the* $\cdot$ *operator.* $Runs(\mathcal{S})$ *is the set of runs of* $\mathcal{S}$*. A* trace *of a run* $r \in Runs(\mathcal{S})$ *is the projection* $proj_{\sum}(r)$ *of* r *on actions.* $Traces(\mathcal{S}) =_{def} proj_{\sum}(Runs(\mathcal{S}))$ *denotes the set of traces of* $\mathcal{S}$*.*

Input/Output Symbolic Transition Systems. An Input/Output Symbolic Transition System (IOSTS) [Rus+00] is a STS where the action set is divided into two subsets: one containing the inputs, beginning with ?, to express actions expected by the system, and another containing outputs, beginning with !, to express actions produced by the system.

Definition 7 (Input/Output Symbolic Transition System) *As defined in [Rus+00], an Input/Output Symbolic Transition System (IOSTS)* $\mathcal{S}$ *is a tuple* $< L, l0, V, V0, I, \Lambda, \rightarrow >$ *where:*

- L *is a finite set of locations;*
- $l0 \in L$ *the initial location;*
- V *is a finite set of location (or internal) variables;*
- $V0$ *is* an *initialization of the location variables* V*;*
- I *is a finite set of parameters, disjoint from* V*;*
- Λ *is a finite set of symbolic actions* $a(p)$*, with* a *a symbol, and* $p = (p_1, \ldots, p_k)$ *a finite set of parameters in* $I^k (k \in \mathbb{N})$*.* p *is assumed unique.* Λ *is partitioned into a set of input actions* Λ^I *and a set of output actions* Λ^O*, and we write* $\Lambda = \Lambda^I \cup \Lambda^O$*:* Λ^I*;*
- $\rightarrow$ *is a finite set of symbolic transitions. A symbolic transition* $(l_i, l_j, a(p), G, A)$*, from the location* $l_i \in L$ *to* $l_j \in L$*, also denoted by* $l_i \xrightarrow{a(p),G,A} l_j$*, is labeled by:*

– *An action* $a(p) \in \Lambda$*;*

– *A guard* G *over* $(p \cup V \cup T(p \cup V))$*, which restricts the firing of the transition.* $T(p \cup V)$ *is a set of functions that return boolean values only (also known as predicates) over* $p \cup V$*;*

– *An assignment* A *that defines the evolution of the variables.* A *is of the form* $(x := A_x)_{x \in V}$*, where* A_x *is an expression over* $V \cup p \cup T(p \cup V)$*.*

Example 2.1.3 Figure 2.5 is the IOSTS of the slot-machine introduced in Example 2.1.2 on page 20. We have $iosts_{machine} =< \{L0, L1, L2, L3\}, L0, \{v\}, \{v \mapsto 0\}, \{i\}, \{coin?, tray!\}, \rightarrow >$ with $\Lambda^I = \{coin?\}$ and $\Lambda^O = \{tray!\}$.

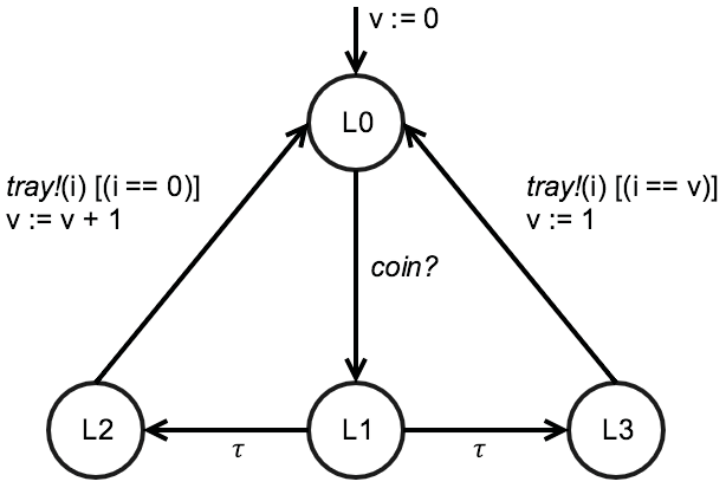

Fig. 2.5: An example of Input/Output Symbolic Transition System representing a simple slot-machine.

In this thesis, we chose to use these models to represent the behaviors of web applications and production systems in order to perform Model-based Testing, *i.e.* relying on models to perform testing, mainly to prevent regressions. *Conformance Testing* is a black-box testing method leveraging formal methods [Tre92], which is both efficient and well-established. We give a few definitions related to conformance testing below.

Conformance testing

In this section, we introduce a few common terms defining what *Conformance Testing* [Bri89; Tre92; Tre94] is, and how it works in general.

Test hypothesis. Executing a test case on a system yields a set of observations. Every observation represents a part of the *implementation model* of the system. The set of all observations made with all possible test cases represents the complete implementation model of the system. The *test hypothesis* [Ber91] is that, for every system, there is a corresponding observational equivalent implementation model: $\forall\, iut \in IMPS, \exists\, i_{iut} \in MODS$, where iut is a concrete *implementation under test*, $IMPS$ is the universe of implementations, i_{iut} is an implementation model of iut, and $MODS$ is the universe of the models of all *implementations under test*.

Conformance. To check whether an implementation under test iut conforms to a specification $spec$, we need to know precisely what it means for iut to conforms to $spec$, *i.e.* a formal definition of conformance is required [Tre08]. As iut is a real, physical "thing", which consists of software (and sometimes physical devices), we cannot use it as a formal object. We rely on the test hypothesis mentioned previously to reason about implementations under test as if they were formal implementations. By doing this, we can define conformance with a formal relation between models of implementations and specifications, *i.e.* an *implementation relation*.

Implementation relation. To formally define conformance between an implementation under test iut and a specification $spec$, we use the notion of an *implementation relation*: $imp \subseteq MODS \times SPECS$, with $SPECS$ the set of specifications. An implementation iut **conforms to** a specification $spec$ if the existing model $i_{iut} \in MODS$ of iut is *imp-related* to $spec$: $i_{iut}\ imp\ spec$.

There are many implementation (or conformance) relations in the literature, *e.g.*, *Isomorphism*, *Bisimulation Equivalence* [Mil89; Fer89], *Trace Equivalence* [Tan+95], *Testing Equivalence* [Abr87], *Refusal Equivalence* [Phi86], *Observation Preorder* [Mil80; HM80], *Trace Preorder* [DNH84; Vaa91], *Testing Preorder* [DNH84; Beo+15], *Refusal Preorder* [Phi87], *Input-Output Testing* [Tre96b], *Input-Output Refusal* [HT97], *ioconf* [Tre96a], and *ioco* [Tre96b].

A simple and easy to understand relation is the *trace preorder* relation $\leq_{tr}$. The intuition behind this relation is that an implementation i may show only behavior, in terms of traces of observable actions, which is specified in the specification s, *i.e.* let $i, s \in \mathcal{LTS}(L)$, then $i \leq_{tr} s =_{def} Traces(i) \subseteq Traces(s)$ [Tre96b].

Example 2.1.4 Figure 2.6 illustrates the trace preorder relation. The traces of the LTS $\mathcal{S}_1$ are included in those of the $\mathcal{S}_2$, *i.e.* $\mathcal{S}_1 \leq_{tr} \mathcal{S}_2$. On the contrary, $Traces(\mathcal{S}_2) \not\subseteq Traces(\mathcal{S}_1)$ because the trace $button \cdot tea$ is not observable in $\mathcal{S}_1$, hence $\mathcal{S}_2 \not\leq_{tr} \mathcal{S}_1$.

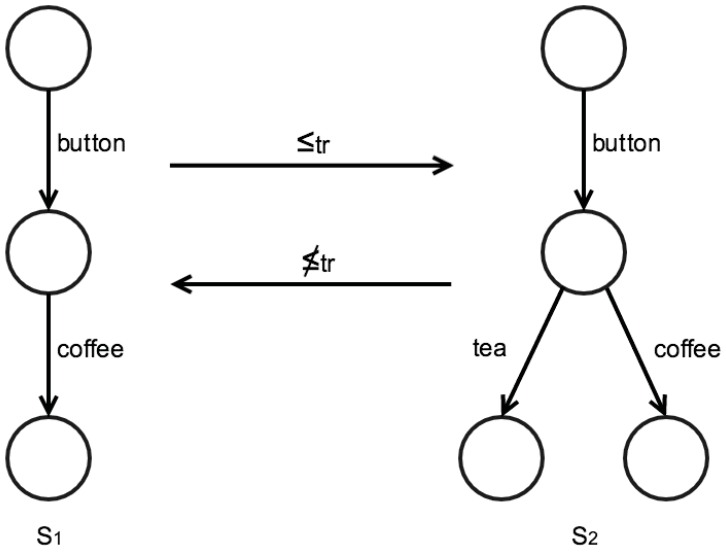

Fig. 2.6: Illustration of the trace preorder relation between two LTSs.

Testing. Conformance testing assesses conformance to an unknown implementation under test iut to its specification $spec$ by means of test experiments. In active testing, experiments consist of stimulating iut in certain ways and observing its reactions with a *tester*. This process is called *test execution*. Test execution may be successful, *i.e.* the observed behaviors correspond to the expected ones, or it may be unsuccessful. The successful execution of a test case TC can be written as follows: i_{iut} **passes** TC. We extend it to a test suite TS: i_{iut} **passes** $TS \Leftrightarrow \forall\, TC \in TS : i_{iut}$ **passes** TC. On the contrary, i_{iut} **fails** $TC \Leftrightarrow i_{iut}$ $\neg$**passes** TC.

This leads to three properties on the test suite TS [Tre96a]:

- **Soundness:** $\forall\, i_{iut} \in MODS, i_{iut}\ imp\ spec \implies i_{iut}$ **passes** TS;
- **Exhaustiveness:** $\forall\, i_{iut} \in MODS, i_{iut}$ **passes** $TS \implies i_{iut}\ imp\ spec$;
- **Completeness:** $\forall\, i_{iut} \in MODS, i_{iut}\ imp\ spec \Leftrightarrow i_{iut}$ **passes** TS.

Until now, we mostly introduced active testing notions. For the record, active testing works by stimulating the system under test, *i.e.* observing outputs of an implementation for predefined inputs. In the next section, we introduce a different approach that does not actively interact with a system, known as *passive testing*.

2.1.3 Passive testing

Passive testing examines the input/output behavior of an implementation without preordaining the input. One advantage of passive testing is that it does not disturb

the system. While most of the works on passive testing are related to networks, protocols, and web services, such a technique is particularly suitable for production systems such as Michelin's systems.

Several works, dealing with passive testing of protocols or components, have been proposed over the last decade. For all of these, the tester is made up of a module, called *monitor*, located in the implementation environment, which collects trace sets. These works can be grouped in three different categories:

- **Invariant satisfiability:** invariants represent properties that are always true. They are often constructed by hand from a specification, and later checked on the collected traces. Similarly to runtime verification [LS09], this approach allows to test complex properties on an implementation. It gave birth to several works in the literature. For instance, the passive testing method presented in [Cav+09b] aims to test the satisfiability of invariants on Mobile *ad hoc* network (MANET) routing protocols. Different steps are required: definition of invariants from the specification, extraction of execution traces with sniffers, verification of the invariants on the trace set. Other works focus on *Component-based System Testing*: in this case, passive methods are usually used to check conformance or security. For instance, the *TIPS* tool [Mor+10] performs an automated analysis of the captured trace sets to determine if a given set of timed extended invariants are satisfied. As in [Cav+09b], invariants are constructed from the specification and traces are collected with network sniffers. Cavalli *et al.* propose an approach for testing the security of web service compositions in [Cav+09a]. Security rules are here modeled with the Nomad language [Cup+05], which expresses authorizations or prohibitions by means of time constraints. Preliminary, a rule set is manually constructed from a specification. Traces of the implementation are extracted with modules that are placed at each workflow engine layer that executes web services. Then, the method checks, with the collected traces, that the implementation does not contradict security rules. Andrés *et al.* presented a methodology to perform passive testing of timed systems in [And+12]. The paper gives two algorithms to decide the correctness of proposed invariants with respect to a given specification and algorithms to check the correctness of a log, recorded from the implementation under test, with respect to an invariant;

- **Forward checking:** implementation reactions are given on-the-fly to an algorithm that detects incorrect behaviors by covering the specification transitions with these reactions. Lee *et al.* proposed a passive testing method dedicated to wired protocols, *e.g.*, [Lee+06]. Protocols are modeled with *Event-driven Extended Finite State Machines* (EEFSM), compound of variables. Several algorithms on the EEFSM model and their applications to the Open Shortest

Path First (OSPF) protocol and Transmission Control Protocol (TCP) state machines are presented. Algorithms check whether partial traces, composed of actions and parameters, meet a given symbolic specification on-the-fly. The analysis of the symbolic specification is performed by means of configuration. A configuration represents a tuple gathering the current state label, and a set of assignments and guards modeling the variable state;

- **Backward checking:** Alcalde *et al.* proposed an approach that processes a partial trace backward to narrow down the possible specifications in [Alc+04]. The algorithm performs two steps. It first follows a given trace backward, from the current configuration to a set of starting ones, according to the specification. With this step, the algorithm finds the possible starting configurations of the trace, which lead to the current configuration. Then, it analyses the past of this set of starting configurations, also in a backward manner, seeking for configurations in which the variables are determined. When such configurations are reached, a decision is taken on the validity of the studied paths (traces are completed). Such an approach is usually applied as a complement to forward checking to detect more errors.

It is worth mentioning that passive testing has also been successfully applied for fault management [MA01], fault detection [Ura+07], performance requirements [CM13], and security purpose [Mal+08]. To summarize, while passive testing is less powerful than active testing, because the latter allows a closer control of the implementation under test, passive testing still presents interesting advantages:

- Passive testing does not disturb the system or its environment, *i.e.* "passive testing only observes and does not intervene" [Cav+03];

- Passive testing can be applied to large systems where active testing is not even feasible, such as systems with many different components, *e.g.*, service oriented architectures, distributed systems (as well as cloud computing [Cav+15]), but also production systems such as the ones we target in this thesis.

2.2 Model inference

In the Industry, software models as defined in Chapter 2.1 • Section 2.1.2 (page 15) are often neglected: specifications are not up to date (or even missing), models are neither accurate nor sound, and also rarely formals.

Such a situation can be comprehensible because writing complete documentation and especially formal models is often a tedious and error prone task. That is why

lightweight and incomplete models are usually found in the Industry. This leads to several issues, *e.g.*, the toughness of testing applications with a good test coverage, the difficulty to diagnose failures, or to maintain models up to date since they are poorly documented.

Solutions to these problems can be initiated by means of model inference. This research domain originates from works on language learning started in the 1970's with Gold [Gol67], based on previous work to formalize natural languages. Model inference here describes a set of methods that infer a specification by gathering and analyzing system executions and concisely summarizing the frequent interaction patterns as state machines that capture the system's behavior. These models, even if partial, can be examined by a developer, to refine the specification, to identify errors, and can be very helpful for in-depth analysis, etc. Models can be generated from different kinds of data samples such as affirmative and negative answers [Ang87], execution traces [Krk+10], documentation [Zho+11], source code [Sal+05; PG09], network traces [Ant+11] or from more abstract documentation such as Web Service Description Language (WSDL) description [Ber+09]. Model inference is employed for different purposes. One can infer models from log files in order to retrieve important information in order to identify failure causes [MP08]. It has also successfully been applied to intrusion detection [MG00], searching for features in execution traces that allow to distinguish browsers from other software systems, and security testing [Gro+12]. Among all, two uses of model inference are Model-based Testing [Lor+08; MS11; Ama+14] and verification [Amm+02; Gro+08].

In the literature, we find different techniques to infer models that can be organized in two main categories. In the first category, we find the techniques that interact with systems or humans to extract knowledge that is then studied to build models. We refer to these methods as active methods. Other works infer models by assuming that system samples, *e.g.*, a set of traces, is provided for learning models. We refer to this principle as passive methods since no interaction is required.

In the following section, we give an overview of prominent active inference approaches: we introduce $\mathcal{L}^*$-based techniques on the next page, incremental learning techniques on page 31, and the model generation by means of automatic testing of event-driven applications providing Graphical User Interfaces (GUIs) on page 33. Passive inference techniques are extensively described in Section 2.2.2 on page 36. We cover passive inference techniques using event sequence-based abstraction on page 38, using state-based abstraction on page 39. Then, we present some white-box approaches on page 41, and we also mention a few techniques that leverage documentation to infer models on page 43.

2.2.1 Active model inference

Active inference refers to algorithms that actively interact with black-box systems or people to extract knowledge about a (software) system, which is then expressed in a model. As depicted in Figure 2.7, interactions are performed with kinds of queries, which are sometimes replaced by testing techniques. A model generator or learner then uses this feedback to incrementally build several models, or to refine the model under generation. Many existing active inference techniques have been initially conceived upon two concepts: the $\mathcal{L}^*$ algorithm, presented on page 28, and incremental learning, presented on page 31. Several more recent papers also proposed crawling techniques of Graphical User Interface (GUI) applications, *i.e.* exploring them through their GUIs with automatic testing. We introduce some of them on page 33.

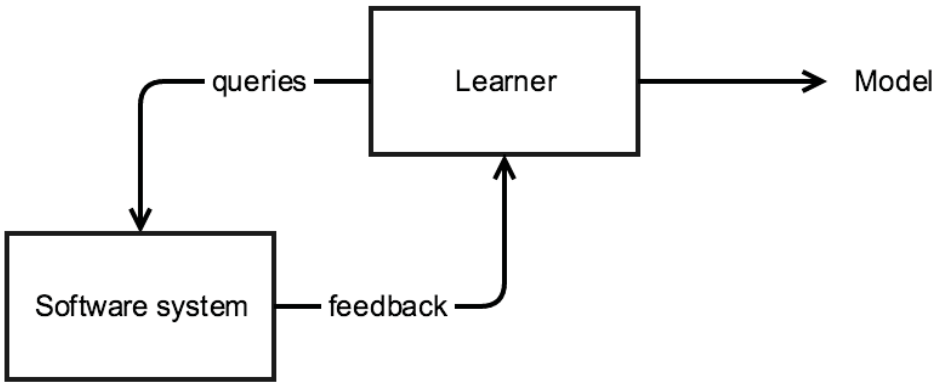

Fig. 2.7: Active model inference principle. Interactions are performed with queries that produce feedback a learner can use to build a model.

$\mathcal{L}^*$-based techniques and related

The $\mathcal{L}^*$ algorithm by Angluin [Ang87] is one of the most widely used active learning algorithm for learning *Deterministic Finite Automata* (DFA). The algorithm has indeed been applied to various problem domains, *e.g.*, protocol inference or the testing of circuits. It is designed to learn a regular language $\mathcal{L}$ by inferring a minimal DFA $\mathcal{A}$ such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}$, with $\mathcal{L}(\mathcal{A})$ the set of strings of $\mathcal{A}$ leading to one of its states from its initial one. The algorithm, also known as the *learner*, knows nothing about $\mathcal{A}$ except its input alphabet. It relies on two roles, a *teacher*, who may answer whether a given string is in the language, and an *oracle*, answering whether the current DFA proposed by the learner is correct or not. The learner interacts with the teacher and the oracle by asking them two kinds of queries, as depicted in Figure 2.8:

- "A *membership query*, consisting in asking the teacher whether a string is contained in the regular language" [Ber06]. If the string is accepted, it is considered as a *positive* example, otherwise it represents a *negative* example;

- "An *equivalence query*, consisting in asking the oracle whether a hypothesized DFA $\mathcal{M}$ is correct, *i.e.* $\mathcal{L}(\mathcal{M}) = \mathcal{L}(\mathcal{A})$" [Ber06]. When the oracle answers no, it provides a counterexample.

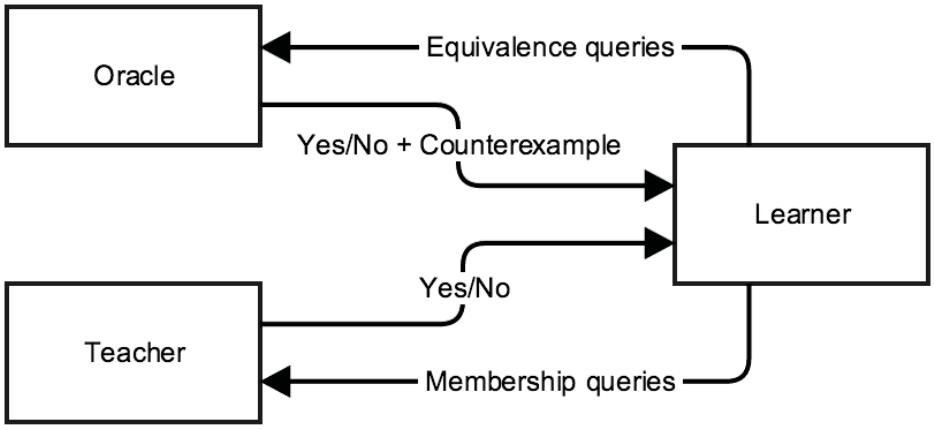

Fig. 2.8: The principle of $\mathcal{L}^*$-based learning algorithms with two kinds of queries: membership queries and equivalence queries.

By taking counterexamples into account, the $\mathcal{L}^*$ algorithm iterates by asking new queries and constructing a new hypothesized DFA $\mathcal{M}$, until it gets an automaton that is equivalent to the black-box. The $\mathcal{L}^*$ algorithm uses an observation table to classify the strings given in membership queries as members or non-members of the unknown regular language. In an observation table, the rows are filled with prefix-closed strings, the columns are labeled by suffix-closed strings. The algorithm gradually fills the entry (u, v) for row u and column v by a boolean value after receiving a reply for a membership query for uv. Once the oracle answers yes, the minimal DFA for the language is derived from this observation table. The $\mathcal{L}^*$ algorithm has also been applied to Mealy machines [Mea55] as shown in [Nie03; Ste+11]. With Mealy machines, the alphabet is segmented into input and output events. The rows of the table are filled with prefix-closed input sequences. The columns of the table are labeled with input suffix sequences, which represent the distinguishing sequences of the states of the future Mealy machine. The table cells are completed by the last output event given by the teacher after receiving the concatenations of prefixes and suffixes.

In domains such as model inference where strings and answers usually do not come from human experts but from experiments, active learning with query synthesis is considered a promising direction [Set09]. Nevertheless, this method has the disadvantages of requiring a lot of iterations and an heavy use of an expert oracle.

Several papers focused on these issues by revisiting [Ber+06; Ber+08], optimizing and/or upgrading $\mathcal{L}^*$ [Raf+05; Irf+12] as summarized below.

Raffelt *et al.* introduced *LearnLib* [Raf+05], a library for learning both DFA and Mealy machines. It implements the $\mathcal{L}^*$ [Ang87] learning algorithm, which is optimized with *approximate equivalence queries*. Furthermore, the notion of teacher is replaced with conformance testing techniques. The query optimization is mainly based on *filters*, which are specific properties of reactive systems that help in the removal of useless queries. A query whose response can be deduce from the previous ones is also ignored. Additionally, statistical data acquisition can be employed to evaluate the learning procedure. These features make *LearnLib* a powerful tool if a teacher is available. Later, Merten *et al.* revisited *LearnLib* in a tool called *Next Generation LearnLib* (NGLL) [Mer+11], a machine learning framework providing infrastructure for practical application, including the tool *LearnLib Studio*, a graphical interface for designing and executing learning and experimentation setups, plus a set of Java libraries. Howar *et al.* pursued the inference of models and proposed a technique and an implementation on-top of *LearnLib* to actively learn register automata in [How+12a]. Register automata, also known as *Finite Memory Automata* [KF94], are models that are capable of expressing the influence of data on control flows. The algorithm directly infers the effect of data values on control flows as part of the learning process. As a consequence, the models are more expressive than DFA, and the implementation also outperforms the classic $\mathcal{L}^*$ algorithm. Howar *et al.* optimized this approach and proposed a method to infer *semantic interfaces* of data structures on the basis of active learning and systematic testing in [How+12b]. Semantic interfaces transparently reflect the behavioral influence of parameters at the interface level. They defined *Register Mealy Machines* (RMMs) to express the data structures behavior concisely, but also because RMMs can be learned much more efficiently than both *Register Automata* and plain Mealy machines.

Berg *et al.* also revisited the $\mathcal{L}^*$ algorithm in [Ber+06] to infer parameterized systems, which are kinds of automata composed of parameters and guards over parameters (boolean expressions) labeled on transitions. The approach completes the $\mathcal{L}^*$ algorithm with guard inference. This algorithm is intended to infer parameterized systems where guards of transitions use only a small subset of all parameters of a particular action type. Such a work has been used later to infer state machines for systems with parameterized inputs [Ber+08]. Here, behavioral models are inferred (finite-state Mealy machine) for finite data domains, and then, models are abstracted to symbolic Mealy machines, encoding extrapolated invariants on data parameters as guarded transitions.

Other works proposed to optimize the $\mathcal{L}^*$ algorithm itself. For instance, Irfan *et al.* proposed the $\mathcal{L}_1$ algorithm [Irf+12] to infer Mealy machines, which uses a modified

observation table and avoids adding unnecessary elements to its columns and rows. In short, the algorithm only keeps the distinguishing input sequences and their suffixes in the columns of the observation table, and the access sequences, *i.e.* the input sequences allowing to reach a state, in the rows. These improvements reduce the table size, and lower the worst case time complexity.

A common problem to the algorithms presented above is the time spent querying the oracle or the teacher, and several competitions, *e.g.*, the ZULU challenge [Com+10] have been proposed to optimize the learning task by trying to make easier queries, or queries for which the oracle's answer is simpler. Concretely, the purpose of such competitions is to optimize the learning algorithm with heuristics to reduce the number of equivalence and membership queries. But having an oracle knowing all about the target model is a strong assumption. Hence, other works propose a completely different solution called incremental learning.

Incremental learning

Instead of querying teachers and oracles to check whether strings and models are correct, incremental learning techniques assumes receiving positive or negative samples, also called observations, one after another. Several models are incrementally built in such a way that if a new observation α is not consistent with the current model, the latter is modified such that α becomes consistent with the resulting new model. In general, a learning algorithm is said incremental if: (i) it constructs a sequence of hypothesis automata $H_0, \ldots, H_n$ from a sequence of observations $o_0, \ldots, o_n$ about an unknown automaton A, and (ii) the construction of hypothesis H_i can reuse aspects of the construction of the previous hypothesis H_{i-1}.

Dupont proposed an incremental extension of the Regular Positive and Negative Inference (RPNI) algorithm, called *Regular Positive and Negative Incremental Inference* (RPNII) [Dup96]. In short, RPNI requires positive and negative samples as a whole and builds DFA. It merges the blocks of states having the same prefixes (strings accepted from a state leading to all the other states) and such that the prefixes augmented by one symbol are not in the negative samples. The inference process of the RPNI algorithm can be seen as a passive approach that is not incremental since it has to be restarted from scratch when new learning data are available. The RPNII algorithm overcomes this limitation by dealing with sequential presentation, *i.e.* the learning data are presented one at a time in a random order.

Parekh *et al.* [Par+98] proposed an incremental extension of Angluin's *ID* algorithm [Ang81], the latter being not incremental since only a single model is ever produced. This extension called *Incremental ID* (IID) constructs DFA from observation sets.

Membership queries are sent to an oracle to check whether the current DFA is correct. IID is guaranteed to converge to the target DFA, and has polynomial time and space complexities. Sindhu *et al.* also enhanced the ID algorithm by proposing another incremental version called *Incremental Distinguishing Sequences* (IDS) [SM12]. The state merging is here performed by generating the distinguishing sequence set DS of every state and by refining blocks of states such that two blocks of states are distinct if and only if they do not have the same DS sets. IDS also has polynomial time and space complexities. In contrast with IID, the IDS algorithm solves some technical errors and its proof of correctness is much simpler.

Meinke introduced the *Congruence Generator Extension* (CGE) algorithm in [Mei10] to infer Mealy automata by applying the term rewriting theory and a congruence generator. Here congruence is an equivalence relation on states and on outputs. Meinke describes this algorithm as being both sequential and incremental. First, it produces a sequence of hypothesis automata $\mathcal{A}_0, \ldots, \mathcal{A}_n$, which are approximations to an unknown automaton $\mathcal{A}$, based on sequence of information (queries and results) about $\mathcal{A}$. CGE is incremental because the computation of a new hypothesis automaton $\mathcal{A}_i$ is based upon the previous $\mathcal{A}_{i-1}$. Meinke shows here that using finite generated congruences increases the efficiency of hypothesis automaton representation, which are deterministic and can be directly analyzed (e.g, for verification purpose). CGE has some features in common with the RPNII algorithm mentioned previously: both RPNII and CGE perform a recursive depth-first search of a lexicographically ordered state set with backtracking. Nevertheless, RPNII is designed for Moore machines while CGE is designed for Mealy machines and, according to Meinke, "RPNII represents the hypothesis automaton state set by computing an equivalence relation on input strings whereas CGE relies on finite generated congruence sets represented as string rewriting systems that are used to compute normal forms of states" [Mei10].

Some incremental learning algorithms have been associated with testing techniques to detect bugs without having an initial specification. In [Mei04; MS11], this concept is called *Learning-Based Testing*. It aims at automatically generating the test cases by combining model-checking with model inference. For example, Meinke and Sindhu [MS11] introduced an incremental learning algorithm named *Incremental Kripke Learning* (IKL) for Kripke structures modeling reactive systems. Here, the test cases are generated and executed to extract observations. Afterwards, models are derived from these observations with an incremental learning algorithm, and model-checking is applied to the inferred models to check if initial requirements are met. If not, counterexamples are collected, and the test cases are generated from them.

Combining incremental model inference with testing has also been studied with applications providing Graphical User Interfaces (GUIs), which are event-driven. We

present some works related to the model inference of GUI applications in the next section.

Model inference of GUI applications

The works presented in this section originate from the testing of GUI applications, *i.e.* event-driven applications offering a Graphical User Interface (GUI) to interact with, and which respond to a sequence of events sent by a user. Partial models can be inferred by exploring interfaces with automatic testing techniques. As depicted in Figure 2.9, models are generated by a learner that updates the model under generation using events and states given by an automatic test engine. This test engine stimulates the application through its GUI (by triggering events, *e.g.*, clicking on a button or a link), and collects all the observed screens. Screen after screen, the application is explored (the technical term is: *crawled*) until there is no more new screen to explore or until some conditions (*e.g.*, based on the processing time or the code coverage rate) are satisfied. The collected events and screens are often represented with transitions and states in a graph or state machine, which expresses the functional behavior of the application observed from its GUI. As the number of screens may be infinite, most of the approaches require state abstractions to limit the model size [Ama+14; Ngu+13; Ama+11; Yan+13; SL15], and a few others merge equivalence states [Mes+12; Ama+08] (the equivalence relation being defined with regard to the context of the application).

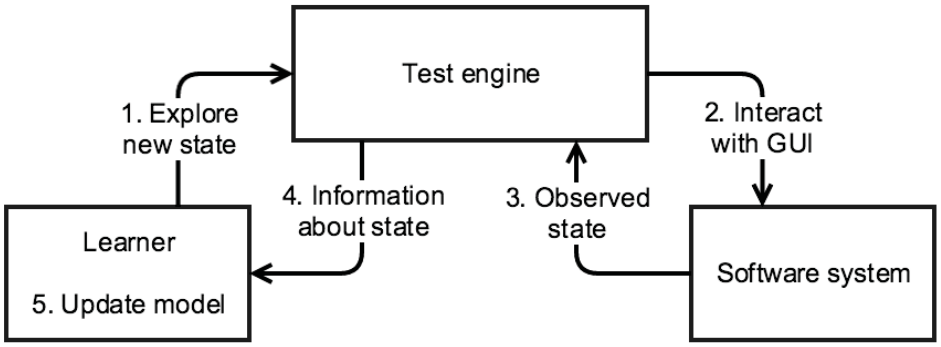

Fig. 2.9: The principle of inferring models with crawling techniques. The learner asks a test engine to stimulate the software under test through its GUI, and constructs a model using the information gathered by the test engine.

All these approaches focus on different applications seen in different viewpoints. We chose to summarize them in Table 2.1, considering the following features:

- **Type of application and accessibility (col. 2 and 3):** most of the approaches focus on desktop, web, or mobile applications. These applications share some common features, *e.g.*, the events that can be applied to screens. Some papers

focus on other kinds of systems though, *e.g.*, distributed and legacy systems [Hun+02].

Applications are often seen as black-boxes even though some authors prefer to consider white- or grey-boxes. Column 3 in Table 2.1 indicates the accessibility chosen for each technique. Some works consider a grey-box perspective to reduce the exploration complexity. Azim *et al.* chose to apply static analyses on Android application source code to guide the application exploration [AN13]. Yang *et al.* [Yan+13] perform static analyses of Android application source code as well in order to list the available events that may be applied to screens;

- **Application environment (col. 4):** a few works [AN13; SL15] take the external environment (*e.g.*, the operating system) of the GUI application into account. The approach proposed by Azim *et al.* [AN13] exercises Android applications with User Interface (UI) events but also with system events to improve code coverage. Examining application environments while testing is more complicated in practice, and it requires more assumptions on the applications. On the other hand, considering the application environment helps build more complete and accurate models [SL15];

- **Model generation (col. 5):** all the approaches cited in Table 2.1 learn either formal (FM) or informal models (IM). Memon *et al.* [Ngu+13] introduced the tool *GUITAR* for scanning desktop applications. This tool produces event flow graphs and trees showing the GUI execution behaviors. The tool *Crawljax* [Mes+12], which is specialized in Asynchronous JavaScript and XML[2] (AJAX) applications, produces state machine models to capture the changes of Document Object Model[3] (DOM) structures of web documents by means of events (click, mouseover, etc.). Amalfitano *et al.* proposed in [Ama+12] a crawler that generates straightforward models, called GUI trees, only depicting the observed screens. In [Yan+13], Yang *et al.* presented an Android application testing method that constructs graphs expressing the called methods. Salva and Laurençot proposed in [SL15] a crawler of Android applications that infers Parameterized Labeled Transition Systems (PLTSs) capturing the UI events, the parameters used to fill the screens, and the screen contents (widget properties).

  These models, and specifically the formal ones, offer the advantage to be reusable for Model-based methods (*e.g.*, Model-based Testing and Verification methods). Nevertheless, the model inference of GUI applications is often paired with the state space explosion problem. To limit the state space, these approaches [Ama+14; Ngu+13; Ama+11; Yan+13; SL15] require state-

[2] A term and a technique that has been coined in 2005: `http://adaptivepath.org/ideas/ajax-new-approach-web-applications/`.
[3] `http://www.w3.org/DOM/`

abstractions specified by users, given in a high level of abstraction. This choice is particularly suitable for comprehension aid, but it often implies a lack of information when it comes to generate test cases. Alternatively, some approaches try to reduce the model on-the-fly. The algorithms introduced in [Mes+12; Ama+08] reduce the model size by concatenating identical states of the model under construction. But this cannot be applied to all applications in a generic manner because a state abstraction definition has to be (manually) given;

- **Exploration strategy (col. 6):** many papers propose at least one strategy to explore GUI applications. The Depth-First Search (DFS) is often considered because this strategy offers the advantage of resetting the application under test fewer time than any other strategy. Nonetheless, some works proposed different strategies [Ama+12; Ama+11; Mes+12; Yan+13] and demonstrated that it can either reduce the exploration time or help increase code coverage. In [SL15], Salva and Laurençot combined the Ant colony optimization heuristic with a model inference engine to support different kinds of exploration strategies. For instance, their algorithm supports semantics-based strategy, *i.e.* strategies guided by the content found in the application' screens;

- **Crash report (col. 7):** we define a crash as any unexpected error encountered by an application. Crash reporting is another feature supported by some of the approaches in Table 2.1. When crashes are observed, reports are proposed to give the error causes and the corresponding application states. Furthermore, the methods proposed in [Ama+14; Ngu+13; SL15] perform stress testing for trying to reveal more bugs, for instance by using random sequences of events. The resulting models are completed thanks to these fault observations. In addition, the tool *AndroidRipper* [Ama+12] generates the test cases for each crash observed.

Generally speaking, these techniques focus more on the GUI application exploration to detect bugs than on the model generation. For instance, a small number of algorithms consider state merging or the definition of state equivalence classes to reduce the model size. At the time of writing, only Choi *et al.* introduced an algorithm combining testing and the use of active learning [Cho+13]. This algorithm is close to the $\mathcal{L}^*$-based approaches presented on page 28, but it also limits the number of times an application has to be reset. A test engine, which replaces the teacher, interacts with the GUI application to discover new application states. The events and states are given to a learning engine that builds a model accordingly. If an input sequence contradicts the current model, the learning engine rebuilds a new model that meets all the previous scenarios. This learning-based testing algorithm avoids restarts and aggressively merges states in order to quickly prune the state

space. Consequently, the authors show that their solution outperforms the classical $\mathcal{L}^*$ technique but models are over-approximated. Salva and Laurençot also shown in [SL15] that this approach requires much more time to build a model than the others.

Active inference approaches repeatedly query systems or humans to collect positive or negative observations. Nonetheless, this can lead to some issues like disturbing the system for instance. That is why we did not choose to perform active model inference, but rather passive model inference, *i.e.* without stimulating the software system. In the next section, we introduce some of these passive model inference techniques.

2.2.2 Passive model inference

The passive model inference category includes all techniques that infer models from a fixed set of samples, such as a set of execution traces, source code, or even documentation, as shown in Figure 2.10. Since there is no interaction with the system to model, theses techniques are said passive or offline. In contrast to active model inference, the samples are usually considered positive elements only. Also, models are often constructed by initially representing sample sets with automata whose equivalent states are merged. This section presents an overview of these techniques.

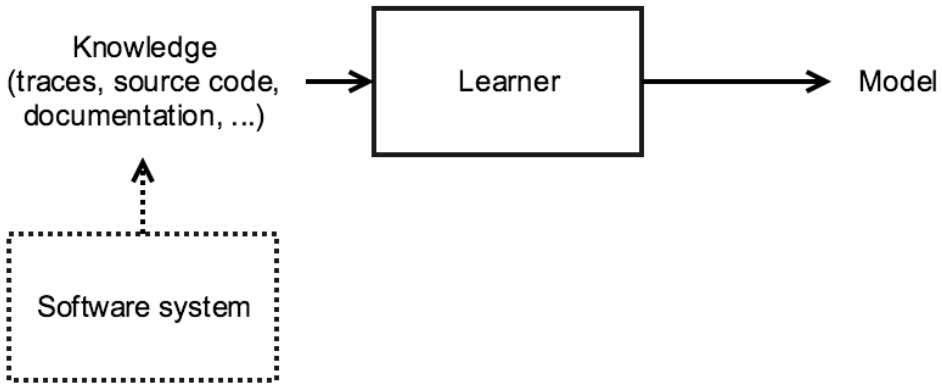

Fig. 2.10: Passive model inference principle. The learner does not interact with the software system but rather relies on a fixed set of information (knowledge).

A substantial part of the papers covering this topic proposes approaches either based upon event sequence abstraction or state-based abstraction to infer models. We introduce them on page 38 and page 39. We present some white-box techniques, which retrieve models from source code, on page 41, as well as a few alternative works leveraging documentation on page 43.

Paper	Type	Accessibility	External environment	Model generation	Strategy	Crash report
[Hun+02]	Distributed systems	Black-box	No	Formal	$\mathcal{L}^*$	No
[JM12]	Mobile	Black-box	No	Informal	DFS	**Yes**
[Dal+12]	Web	Black-box	No	Informal	-	**Yes**
[Ama+11; Ama+12]	Mobile	Black-box	No	Informal	BFS, DFS	**Yes**
[Ngu+13; Ama+14]	desktop, Mobile	Black-box	No	Formal	DFS	**Yes**
[Mes+12]	Web	Black-box	No	Formal	Multiple	No
[Ama+08]	Web	Black-box	No	Informal	DFS	No
[Cho+13]	Mobile	Black-box	No	Formal	DFS	No
[Yan+13]	Mobile	Grey-box	No	Informal	Multiple	No
[AN13]	Mobile	Grey-box	**Yes**	Informal	DFS	**Yes**
[SL15]	Mobile	Black-box	**Yes**	Formal	Multiple	**Yes**

Tab. 2.1: An overview of some works on GUI application crawling. Column 2 represents the type of application under analysis. Column 3 indicated the accessibility (black-, grey-, or white-box). Column 4 gives whether the technique requires an external environment. Column 5 indicates whether the technique builds formal models or informal models. Column 6 gives the strategy or algorithm used to crawl the application. Last column (7) shows whether the technique handles potential crashes, *i.e.* errors encountered during crawling.

Passive inference using event sequence abstraction

Most of the following approaches, which build models from execution traces by means of event sequence abstraction, are build on-top of these two algorithms: *kTail* [BF72] and *kBehavior* [MP07].

kTail generates *Finite State Automata* (FSA) from trace sets in two steps. First, it builds a *Prefix Tree Acceptor* (PTA), which is a tree whose edges are labeled with the event names found in traces. Then, kTail transforms the PTA into a FSA by merging each pair of states as far as they exhibit the same future of length k, *i.e.* if they have the same set of event sequences having the maximum length k, which are all accepted by the two states. This state merging step often yields over-generalized models containing undesirable behaviors though [LK06].

Reiss and Renieris modified the kTail algorithm to reduce the size of the final FSA. Indeed, their algorithm merges two states if they share at least one *k-future* [RR01]. By using a merging criterion that is weaker, their variant actually merges more states than kTail. Yet, the resulting FSA express much more behaviors than those possible in the real system. They are usually more approximate than the models obtained with kTail. Lo *et al.* [Lo+09] also enhanced the kTail algorithm to lower over-approximation. Traces are mined to extract temporal properties that statistically hold in most of the traces. Such temporal properties aim at capturing relations between non consecutive events. The kTail algorithm is then upgraded to prevent the merging of states with the same k-future, which would produce FSA that violate the inferred properties.

Lorenzoli *et al.* extended kTail to produce *Extended Finite State Machines* (EFSMs), which are FSMs extended with parameters and algebraic constraints on transitions [Lor+08]. Their technique, called *gkTail*, generates an EFSM from a set of traces, which incorporates information about both the event sequences and the values of the parameters associated with the event sequences. gkTail starts by combining the traces that share the same event sequence, each event being associated with the set of values obtained as the union of all value assigned to this same event in each merged trace. gkTail infers a constraint from the values associated with each event, and combines this constraint with the associated event. Finally, the kTail algorithm is applied to these traces.

kBehavior [MP07] is another algorithm, which works quite differently than kTail. It generates a FSA from a set of traces by taking every trace one after one, and by completing the FSA such that it accepts the trace. More precisely, whenever a new trace is submitted to kBehavior, it first identifies the sub-traces that are accepted by sub-automata in the current FSA (the sub-traces must have a minimal length k,

otherwise they are considered too short to be relevant). Then, kBehavior extends the model with the addition of new branches that connect the identified sub-automata, producing a new version of the model that accepts the entire trace. They successfully applied this algorithm to automatically analyze log files and retrieved important information to identify failure causes [MP08]. They also automatically analyzed logs obtained from workloads to highlight useful information that can relate the failure to its cause [Cot+07]. Both works [MP08; Cot+07] use an extended version of kBehavior, called *kLFA*, that supports events combined with data values. kLFA performs a preliminary step by analyzing the traces to infer parameter constraints. It encodes these constraints in the event names with specific symbols to yield new traces. kBehavior is then called to infer a FSA whose transitions are still labeled with an event and a symbol representing a data constraint.

Lo *et al.* presented an empirical comparative study of kTail, kBehavior, gkTail, and kLFA with a set of 10 case studies extracted from real software systems in [Lo+12]. This study quantifies both the effect of adding data flow information within automata and the effectiveness of the techniques when varying sparseness of traces. One of the main conclusions is that adding algebraic constraints to FSA does not compromise quality but negatively affects performance. This is the case for gkTail for instance, which becomes extremely slow. The study also revealed that increasing the trace set improves the rate of correct behaviors in models, especially for kTail and kBehavior. But increasing the trace set does not particularly affect illegal behavior rates, *i.e.* the number of behaviors found in the models but not in the traces. This can be explained by the fact that these algorithms are not really able to control over-generalization when only positive samples are available. A comparison has also been made between the kTail- and kBehavior-based methods. In short, kTail provides low illegal behavior rate, but also low correct behavior rate. On the other hand, kBehavior has higher illegal behavior rate, but good correct behavior rate.

The next section gathers the approaches building models from traces that also consider invariants to merge states.

Passive inference using state-based abstraction

Most of the approaches presented in this section rely on the generation of state invariants to define equivalence classes of states that are combined together to form final models. The *Daikon* tool [Ern+99] was originally proposed to infer invariants composed of data values and variables found in execution traces. An invariant is a property that holds at a certain point or points in a software. These are often used in assert statements, documentation, and formal specifications. An invariant generator mines the data found in the traces that a software system produces, and

then reports properties that are $true$ over the observed executions. This is a machine learning technique that can be applied to arbitrary data. Daikon is used in many different kinds of works, *e.g.*, for generating test cases, predicting incompatibilities in component integration, automating theorem proving, repairing inconsistent data structures, and checking the validity of data streams, among other tasks [Ern+07].

Krka *et al.* inferred object-level behavioral models (FSA) from object executions, by using both invariants, representing object states, and dynamic invocation sequences, expressing method invocations [Krk+10]. Invariants are still generated with Daikon. The authors show that their FSA inference technique offers more precise models than those obtained with kTail, which means that the rate of over-approximation is lower. These results are not surprising since they use a state merging technique combining event sequence abstraction and state abstraction. Hence, state merging is here done with more precision.

In [Ghe+09], Ghezzi *et al.* described an approach called *SPY* to recover a specification of a software component from its traces. They infer a formal specification of stateful black-box components (Java classes that behave as data containers with their own states). Model inference is performed in two main steps. It starts by building a DFA that models the partial behavior of the instances of the classes. Then, the DFA is generalized *via* graph transformation rules that are based upon the following assumption: the behavior observed during the partial model inference process benefits from the so called "continuity property" (*i.e.* a class instance has a sort of "uniform" behavior). Transformation rules generate data constraints that hold for each encountered data value found in the instance pools. Such constraints add over-approximation to the model though.

Walkinshaw *et al.* presented the *Query-driven State Merging* (QSM) algorithm in [Wal+07], an *interactive* grammar inference technique to infer the underlying state machine representation of software. The QSM algorithm does not infer invariants but uses the Price's "blue-fringe" state merging algorithm [Lan+98], which is mainly based upon fixed invariants defining order relations over states. QSM generalizes a supplied trace set obtained from an application by applying two steps: (i) a trace abstraction is performed with functions given by a user, and (ii) states are merged with the Price's blue-fringe state merging algorithm. To avoid over-generalization, the algorithm queries the user whenever the resulting machine accepts or rejects sequences that have not been ratified. This approach can be compared to the work done by Hungar in [Hun+04], who used the $\mathcal{L}^*$ algorithm instead. But the QSM algorithm presumes that the input sequences offer some basic coverage of the essential functionality of the system, in which case the machine can be inferred relatively cheaply by a process of state merging, compared to the $\mathcal{L}^*$ technique that systematically and comprehensively explores the state space of the target machine.

Some tools such as *Synapse* [LS+14] implement the QSM algorithm to perform automatic behavior inference and implementation comparison for the programming language Erlang.

Taking another direction by leveraging genetic algorithms, Tonella *et al.* [Ton+13] applied a data-clustering algorithm to infer FSMs from traces. Traces are transformed into a first FSM where every state is considered as one distinct equivalence class, called cluster. Then, invariants are generated with Daikon for each cluster in order to group states. The clustering is iteratively improved by using a genetic algorithm that randomly updates the clusters. But the clustering is yet guided with the computation of quality attributes on the current FSM model. Each distinct set of invariants produced for each cluster at the end of the optimization represents an abstract state, and is used as the abstraction function that maps states to more abstract ones. Even though this approach offers originality, it is time consuming, especially with a large set of traces.

The works presented in this section adopted either a grey- or black-box approach. The next section introduces some white-box techniques.

White-box techniques

The works presented in this section adopt a white-box perspective. Models are built by following two different procedures: (i) the source code is instrumented and the system is executed or tested to collect traces from which a specification can be generated, and (ii) the source code is statically analyzed to directly infer models.

Ammons *et al.* reuse the kTail algorithm to produce non-deterministic automata from source code [Amm+02]. The latter is instrumented to collect the function calls and parameters. Application traces are collected and scenarios (small sets of interdependent interactions) are then derived. kTail is finally employed to build automata. Several automata are generated from the different scenarios.

Whaley *et al.* suggested to use multiple FSM submodels to model object-oriented component interfaces [Wha+02]. A FSM is built for each attribute of a class, with one state per method that writes that attribute. The FSMs are reduced with restrictions on the methods: a distinction is made between side-effect-free methods, *i.e.* those that do not change the object state, and the others. The side-effect-free methods are ignored to build FSMs. The others are completed with dotted transitions to represent the states from which it is possible to call these side-effect-free methods. Two techniques are given to automatically construct such models: (i) a dynamic instrumentation technique that records real method call sequences, and (ii) a static

analysis that infers pairs of methods that cannot be called consecutively. The work described in [Alu+05] generates behavioral interface specifications from Java classes by means of predicate abstraction and active learning. This is a white-box approach inspired by [Wha+02] that, first, uses *predicate abstraction* to generate an abstract version of the considered class. Afterwards, a minimal version (interface) of this abstraction is constructed by leveraging the $\mathcal{L}^*$ algorithm. The tool *Java Interface Synthesis Tool* (JIST) is the resulting implementation of such a technique. It takes Java classes as input, and generates useful and safe interfaces automatically.

Still in the purpose of modeling object invocation, Salah *et al.* proposed *Scenariographer* [Sal+05], an algorithm and a tool to estimate the usage scenarios of a class from its execution profiles. The results are quite different than the above approaches tough, since Scenariographer infers a set of regular expressions expressing the generalized usage scenarios of the class over its methods. The source code is still instrumented to collect method calls, but this approach employs the notion of canonical sets to categorize method sequences into groups of similar sequences, where each group represents a usage scenario for a given class.

Yang *et al.* [Yan+06] also focused on the model inference of Java programs but they use a state abstraction technique generating temporal invariants. The source code is instrumented to collect method invocations. Then, temporal invariants are extracted to capture relations among several consecutive or non-consecutive events. The resulting tool *Perracotta* infers temporal invariants and FSMs from traces. It is able to scale to large software systems since it can infer models from millions of events. Additionally, it works effectively with incomplete trace sets typically available in industrial scenarios. Scalability is here obtained by the use of heuristics to prune the set of inferred invariants.

In [PG09], Pradel and Gross presented a scalable dynamic analysis that infers extremely detailed specifications of correct method call sequences on multiple related objects. Again, Java applications are executed to collect with debuggers methods calls. This produces a large set of traces. Given that methods generally implement small and coherent pieces of functionality, the source code is statically analyzed to identify small sets of related objects (object collaborations), and method calls that can be analyzed separately. Then, they derive FSMs that model legal sequences of method calls on a set of related objects. This work has been extended in [Dal+10] by means of active testing. The inference engine generates the test cases that cover previously unobserved behaviors, systematically extending the execution space and enriching the models while detecting bugs. Such an approach is similar to the crawling techniques presented in Section 2.2.1 except that crawlers take GUI applications as input.

Even though the remaining papers are oriented towards the same purpose, that is model inference from source code, they do not collect traces from executions but perform static analysis only. Shoham *et al.* introduced an approach to infer finite automata describing method calls of an API from source code of the client side, *i.e.* the code that calls the API [Sho+07]. Finite automata are constructed with two main steps: (i) abstract-traces are collected with a static analysis of the source code and split into sets called abstract histories that give birth to several finite automata, and (ii) a summarization phase filters out noise. This step is done thanks to two available criteria: either all the histories are considered (no filter) or the histories having the same k-future (as with kTail) are assembled together. Despite encouraging results, the inferred specifications are often too detailed, and their solution does not scale well.

In [Was+07], Wasylkowski *et al.* proposed *JADET*, a Java code analyzer to infer method models. One finite state automaton is built for every class to express the method calls. JADET then infers temporal properties grouped into patterns that can be used to automatically find locations in programs that deviate from normal object usage. Temporal properties are obtained by the use of the data mining technique frequent item set mining, which is applied to the source code. These properties are fed into a classifier that detects and removes the abnormal properties. It then identifies the methods that violate the remaining ones.

The next section introduces a few other works leveraging documentation to infer models.

Model inference from documentation

We present in this section some other passive inference techniques that rely on other sets of knowledge to infer models. Their main disadvantage lies in the use of external documentation that can be outdated, leading to models describing incorrect behaviors (over-approximation). Furthermore, the resulting models are often under-approximated since documentation is usually not complete. That is probably why there are few works based on documentation mining in the literature.

Bertolino *et al.* presented *StrawBerry* in [Ber+09], a method that infers a Behavior Protocol automaton from a Web Service Description Language (WSDL) document. WSDL is a format for documenting web service interactions, containing information about the inputs, outputs, and available methods. StrawBerry automatically derives a partial ordering relation among the invocations of the different WSDL operations, that is represented as an automaton called Behavior Protocol automaton. This automaton models the interaction protocol that a client has to follow in order to

correctly interact with a web service. The states of the behavior protocol automaton are web service execution states, and the transitions, labeled with operation names and input/output data, model possible operation invocations from the client to the web service.

Later, Zong *et al.* [Zho+11] proposed to infer specifications from API documentation to check whether implementations match it. A linguistic analysis is applied as API documentation is written in a natural language. It results in a set of methods that are linked together with action-resource pairs that denote what action the method takes on which resource. Then, a graph is built from these methods and a predefined specification template. Such specifications do not reflect the implementation behaviors though. Furthermore, this method can be applied only if the API documentation is available in a readable format, and if a specific template is provided.

2.3 Conclusion

As stated previously, the use of active inference techniques is not possible in our industrial context. Production systems are both distributed and heterogeneous event-driven systems, compound of software and several physical devices (*e.g.*, points, switches, stores, and production machines). Applying active inference on them without disturbing them is therefore not feasible in practice. We are not supposed to cause damages on these systems while studying them, that is why we investigated the existing passive inference techniques.

Given the constraints formulated in Chapter 1 • Section 1.2 (page 3), we are not able to leverage white-box or documentation-based works. For the record, we cannot safely reuse existing documentation, and due to the heterogeneous set of software we target, we cannot rely on white-box approaches either. At first glance, and because production systems are event-driven, techniques using event sequence abstraction seemed the most relevant to us.

Yet, our main concern regarding methods such as kTail and kBehavior was the over-approximation tied to the inferred models. Indeed, while it may not always be an issue, depending on the use cases, our context requires exact models for testing. That is why we present two new approaches combining passive model inference, machine learning, and expert systems to infer models from traces for web applications (Chapter 3) and industrial systems (Chapter 4) in the sequel. The second technique is an adaptation of the first one.

We believe that knowledge of human domain experts is valuable, hence the use of expert systems to integrate their knowledge with our inference techniques. Expert

systems are computer systems that emulate the decision-making abilities of humans. We "transliterate" knowledge of domain experts into inference rules, which our inference techniques leverage thanks to expert systems. Our work is still similar to those using event sequence abstraction (page 38), except that state merging is replaced with a context-specific state reduction based on an event sequence abstraction. This state reduction can be seen as the kTail algorithm introduced previously where k would be dynamic and as high as possible. Furthermore, our two simple yet scalable techniques infer exact models for testing purpose. We focus on both speed and scalability to be able to construct models of Michelin's production systems in an efficient manner, which is often a must-have for adoption in the Industry.

Among all potential use cases, we need such models to perform passive testing on these production systems (Chapter 5). We chose to perform passive testing for the same reasons mentioned before, *i.e.* because passive testing does not disturb the system, and because it can be applied to large and heterogeneous systems.

3 Model inference for web applications

Our target has always been to construct models of (legacy) production systems in an automated fashion. But, before tackling the model inference of production systems, we chose to formulate our ideas and concepts by applying them on web applications. Given a black-box approach, web applications are often smaller and less complex than production systems: no physical devices involved, a well-known client-server architecture, and a well-defined protocol to exchange information.

In this chapter, we introduce the first version of *Autofunk* (v1), our modular framework for inferring models, *i.e.* Input/Output Symbolic Transition Systems here, by using a rule-based expert system. A bisimulation minimization technique [Par81] is applied to reduce the model size. This framework generates several models of a same application at different levels of abstraction. *Autofunk* relies on an *automatic testing* technique to improve the completeness of the inferred models. We evaluate our tool on the GitHub[1] website, showing encouraging results.

Contents

[1] `https://github.com/`

3.1 Introduction

Web applications are often smaller and easier to understand than production systems. The design of web applications is well-established: we have a web server that runs the application, no matter its underlying programming language, and a client (a web browser most of the time) that connects to the web server so that it can communicate with the application. *Hypertext Transfer Protocol* (HTTP) is the protocol that allows such a discussion by means of HTTP requests and HTTP responses [FR14]. Usually, the client initiates a request, and the server responds to it. That is why we consider web applications as event-driven applications: they react to events. Requests embed HTTP verbs (or methods), *Unique Resource Identifiers* (URIs also known as the "web links"), and sometimes parameters along with their values. When one submits a registration form on a web page, it is likely that the HTTP verb is $POST$, and that there are at least as many parameters and values as fields in the form. By clicking on the "submit" button (or "save", "register", etc.), the web browser crafts an HTTP request containing all the information, and sends it to the web server that delivers the request to the application. When the application sends back the appropriate response, it also goes through the web server. By hooking between the web browser and the web server, we are able to read both HTTP requests and responses as far as there is no *Secure Socket Layer* (SSL) or *Transport Layer Security* (TLS) involved, *i.e.* the communication channel should not be encrypted.

The information collected by reading HTTP requests and HTTP responses are known as (HTTP) *traces*. We use such traces to infer raw models of the behaviors of a web application. We then construct more abstract models that can be used for different purposes, *e.g.*, documentation or test case generation. We chose to work on this to explore different options for production systems in the future. We present an active technique that uses automatic testing to stimulate the web application in order to enhance the completeness of the inferred models.

We start by giving an overview of this framework called *Autofunk* in the next section, which we detail in Section 3.3 and Section 3.4, which covers the notion of strategies to infer more complete models. We present our results in Section 3.5, and we conclude on this chapter in Section 3.6.

Publications. This work has been published in *Actes de la 13eme édition d'AFADL, atelier francophone sur les Approches Formelles dans l'Assistance au Développement de Logiciels* (AFADL'14) [DS14a], and in the Proceedings of the Fifth Symposium on Information and Communication Technology (SoICT'14) [DS14b].

3.2 Overview

We propose a new approach to infer models of web applications, which is divided into several modules as depicted in Figure 3.1. The *Models generator* is the centerpiece of this framework. It takes HTTP traces as inputs, which can be sent by a *Monitor* collecting them on-the-fly. Such a monitor would be responsible for reading HTTP requests and responses in our case. Nonetheless, it is worth mentioning that the traces can also be sent by any tool or even any user, as far as they comply to a chosen standard format like *HTTP Archive* (HAR), which is implemented in recent browsers' developer tools. The Models generator is based upon an expert system, which is an artificial intelligence engine emulating acts of a human expert by inferring a set of rules representing his knowledge. Such knowledge is organized into a hierarchy of several layers. Each layer gathers a set of inference rules written with a first-order logic. Typically, each layer creates a model, and the higher the layer is, the more abstract the model becomes. Models are then stored and can be later analyzed by experts, verification tools, etc. The number of layers is not strictly bounded even though it is manifest that it has to be finite. We choose to infer *Input/Output Symbolic Transition Systems* (IOSTSs) to model behaviors of web applications. Indeed, such models are suitable to describe communicating systems, *e.g.*, systems based on client-server architectures, with data by means of automata along with inputs and outputs [Rus+00].

The Models generator relies upon traces to construct IOSTSs (cf. Chapter 2 • Section 2.1.2 (page 21)), but the given trace set may not be substantial enough to generate relevant IOSTSs. More traces could be yet collected as far as the application being analyzed is an event-driven application. Such traces can be produced by stimulating and exploring the application with automatic testing. In our approach, this exploration is achieved by the *Robot explorer*, which is our second main contribution in this framework. In contrast with most of the existing crawling techniques, which are detailed in Chapter 2 • Section 2.2.1 (page 33), our robot does not cover the application in blind mode or with a static traversal strategy. Instead, it is cleverly guided by the Models generator, which applies an exploration strategy carried out by inference rules. This involves the capture of new traces by the Monitor or by the Robot explorer that returns them to the Models generator. The advantages of this approach are manifold:

- It takes a predefined set of traces collected from any kind of applications producing traces. In the context of web applications, traces can be produced using automatic testing;

- The application exploration is guided with a strategy that can be modified according to the type of application being analyzed. This strategy offers

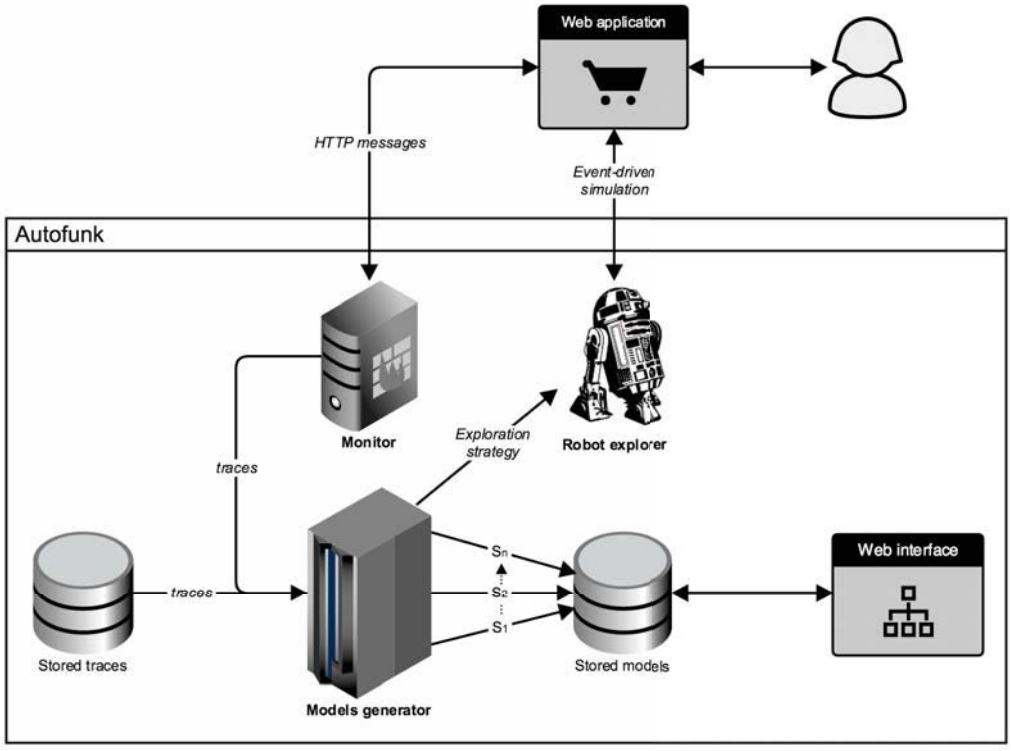

Fig. 3.1: Very first overall architecture of *Autofunk* (v1), our model generation framework targeting web applications.

the advantage of directly targeting some states of the application when its state number is too large for being traversed in a reasonable processing time. Strategies are defined by means of inference rules;

- The knowledge encapsulated in the expert system can be used to cover trace sets of several applications thanks to generic rules. For instance, the same rules can be applied to the applications developed with the same framework or programming language;

- But, the rules can also be specialized and refined for one application to yield more precise models. This is interesting for application comprehension;

- Our approach is both flexible and scalable. It does not produce one model but several ones, depending on the number of layers of the Models generator, which is not limited and may evolve in accordance to the application's type. Each model, expressing the application's behaviors at a different level of abstraction, can be used to ease the writing of complete formal models, to apply verification techniques, to check the satisfiability of properties, to automatically generate functional test cases, etc.

In the following section, we describe our model inference method.

3.3 Inferring models with rule-based expert systems

The Models generator is mainly composed of a rule-based expert system, adopting a forward chaining. Such a system separates the knowledge base from the reasoning: the former is expressed with data also known as facts and the latter is realized with inference rules that are applied to the facts. Our Models generator initially takes traces as an initial knowledge base and owns inference rules organized into layers for trying to fit the human expert behavior. These layers are depicted in Figure 3.2.

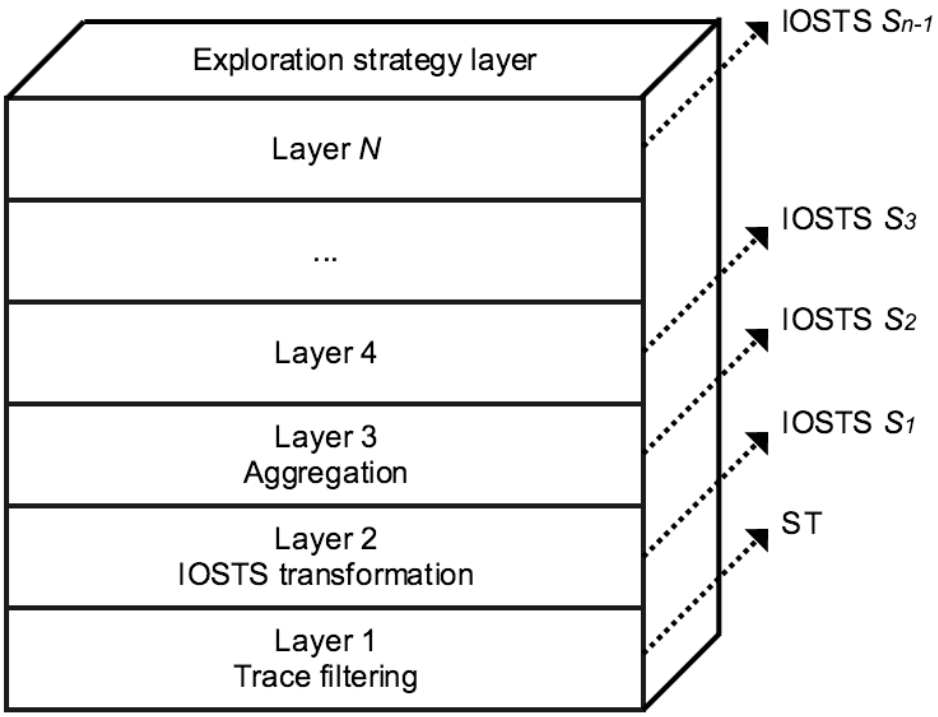

Fig. 3.2: The Models generator stack. Each layer yields a model that is reused by the next layer to yield another model, and so on. An orthogonal layer describes any kind of exploration strategy by means of rules.

Usually, when a human expert has to read traces of an application, she often filters them out to only keep those that make sense against the current application. She has a general understanding of the application, and she is able to "infer" more abstract behaviors of the traces read with a focus on relevant information only. This step is done by the first layer whose role is to format the received HTTP traces into sequences of valued actions and to delete those considered as unnecessary. The rules of this layer depend on the nature of the input traces. The resulting structured trace set, denoted by ST, is then given to the next layer. This process is incrementally done, *i.e.* every time new traces are given to the Models generator, these are formatted

and filtered before being given to Layer 2. The remaining layers yield an IOSTS each $\mathcal{S}_i (i \geq 1)$, which has a tree structure derived from the traces. The role of Layer 2 is to carry out a first IOSTS transformation from the structured traces of ST, and then to yield a minimized IOSTS $\mathcal{S}_1$. The next layers 3 to N (with N a finite integer) are composed of rules that emulate the ability of a human expert to simplify transitions, to analyze the transition syntax for deducing its meaning in connection with the application, and to construct more abstract actions that aggregate a set of initial ones. Theses deductions are often not done in one step. This is why the Models generator supports a finite but not defined number of layers. Each of these layers i takes the IOSTS $\mathcal{S}_{i-1}$ given by the direct lower layer. This IOSTS, which represents the current base of facts, is analyzed by the rules of a expert system to infer another IOSTS whose expressiveness is more abstract than the previous one. We state that the lowest layers (at least Layer 3) should be composed of generic rules that can be reused on several applications of the same type. In contrast, the highest layers should own the most precise rules that may be dedicated to one specific application.

For readability purpose, we chose to represent inference rules with the *Drools*[2] rule inference language. Drools is a rule-based expert system adopting a forward chaining, *i.e.* it starts with the available information and uses inference rules to extract more information. A *Drools rule* has the following rough structure:

```
rule "name"
  when
    LHS
  then
    RHS
end
```

LHS stands for Left-Hand Side. It is the conditional part of the rule, containing the *premises* of the rule. RHS is the Right-Hand Side of the rule, *i.e.* the actions that are executed (also known as conclusions) whenever LHS is evaluated to true. The rule above could have been expressed as follows: $\frac{LHS}{RHS}$, but for more complex rules, the Drools formalism is more readable. Furthermore, one of the biggest advantages of Drools is its ability to handle structured data as facts, such as Java objects.

Independently on the application's type, Layers 2 to N handle the following fact types: *Location*, which represents an IOSTS location, and *Transition*, which represents an IOSTS transition composed of two locations $Linit$, $Lfinal$, and two data collections *Guard* and *Assign*. It is manifest that the inference of models has to be done in a finite time and in a deterministic way, otherwise our solution may not be usable

[2] `http://www.jboss.org/drools/`

in practice. To reach that purpose, we formulate the following hypotheses on the inference rules:

1. **Finite complexity:** a rule can only be applied a limited number of times to the same knowledge base;

2. **Soundness:** the inference rules are *Modus Ponens*, *i.e.* simple implications that lead to sound facts if the original facts are true (P implies Q; P is asserted to be true, so therefore Q must be true.);

3. **No implicit knowledge elimination:** after the application of a rule r expressed by the relation $r : T_i \rightarrow T_{i+1} (i \geq 2)$, with T_i a transition base, for a transition $t = (l_n, l_m, a(p), G, A)$ extracted from T_{i+1}, l_n is still reachable from $l0$, *i.e.* rules should not break the existing IOSTS paths.

In the following, we illustrate each layer with examples.

3.3.1 Layer 1: Trace filtering

Traces of web applications are based upon the HTTP protocol, conceived in such a way that each HTTP request is followed by only one HTTP response. Consequently, the traces, given to Layer 1, are sequences of couples (HTTP request, HTTP response). This layer begins formatting these couples so that these can be analyzed in a more convenient way.

An HTTP request is a textual message containing an HTTP verb (also called method), followed by a Unique Resource Identifier (URI). It may also contain header sections such as Host, Connection, or Accept. The corresponding HTTP response is also a textual message containing at least a status code. It may encompass headers (*e.g.*, Content-Type, Content-Length) and a content. All these notions can be easily identified. For instance, Figure 3.3 depicts an HTTP request followed by its response. This is a GET HTTP request, meaning a client wants to read the content of the $/hello$ resource, which is, in this case, a web page in HTML.

For a couple *(HTTP request, HTTP response)*, we extract the following information: the HTTP verb, the target URI, the request content that is a collection of data (headers, content), and the response content that is the collection (HTTP status, headers, response content). A header may also be a collection of data or may be null. Contents are textual, *e.g.*, HTML. Since we wish to translate such traces into IOSTSs, we turn these textual items into a structured *valued action* $(a(p), \alpha)$ with a the HTTP verb and α a valuation over the variable set $p = \{URI, request, response\}$. This is captured by the next definition.

```
GET /hello HTTP/1.1
Host: example.org
Connection: keep-alive
Accept: text/html
```

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 13
Hello, World!
```

Fig. 3.3: An example of HTTP request and response. HTTP messages are sent in plain text, according to RFC 7230 [FR14].

Definition 8 (Structured HTTP trace) *Let* $t = req_1, resp_1, \ldots, req_n, resp_n$ *be a raw HTTP trace composed of an alternate finite sequence of HTTP request* req_i *and HTTP response* $resp_i$*. The structured HTTP trace* σ *of* t *is the sequence of valued actions* $\sigma(t) = (a_1(p), \alpha_1) \ldots (a_n(p), \alpha_n)$ *where:*

- a_i *is a HTTP verb used to make the request in* req_i*;*
- p *is a parameter set* $\{URI, request, response\}$*;*
- α_i *is a valuation* $p \rightarrow D_p$ *that assigns a value to each variables of* p*.* α *is deduced from the values extracted from* req_i *and* $resp_i$*.*

The resulting structured HTTP trace set derived from the raw HTTP traces is denoted by ST*.*

Example 3.3.1 If we take back the HTTP messages given in Figure 3.3, we obtain the structured HTTP trace $\sigma = (Get(p), \alpha)$ with $\alpha = \{URI := "/hello", request := \{headers = [Host := "example.org", Connection := "keep-alive", Accept := "text/html"]\}, response := \{status\_code := 200, headers = [Content-Type := "text/html", Content-Length := 13], content := "Hello, World!"\}\}$.

For a main request performed by a user, many other sub-requests are also launched by a browser in order to fetch images, CSS[3] and JavaScript files. Generally speaking, these do not enlighten a peculiar functional behavior of the application. This is why we propose to add rules in Layer 1 to filter these sub-requests out from the traces. Such sub-requests can be identified by different ways, *e.g.*, by focusing on the file extension found at the end of the URI or on the content-type value of the request

[3] Cascading Style Sheets

headers. Consequently, this layer includes a set of rules, constituted of conditions on the HTTP content found in an action, that remove valued actions when the condition is met.

A straightforward rule example that removes the actions related to the retrieval of PNG[4] images is formally given below (with σ being a trace). A human expert knows that the retrieval of a PNG image implies a GET HTTP request of a file whose extension is png or whose content type is $image/png$ (defined in a header of the response):

$$\frac{valued\_action = (Get(p), \alpha) \in \sigma,\ \alpha \models [(request.file\_extension == 'png') \vee (response.headers.content\_type == 'image/png']}{\sigma = \sigma \setminus \{valued\_action\}}$$

Yet, the Drools rule language is much simple and more convenient, especially for the domain experts who have to write such rules. That is why we chose to write inference rules with this language. In the sequel, inference rules for *Autofunk* are given as Drools rules.

For example, the inference rule above can be written with the Drools formalism as depicted in Figure 3.4. $\$valued\_action$ is a variable representing a Get valued action. Actions own $request$ and $response$ attributes, respectively representing the HTTP request and HTTP response for that given valued action. Such attributes can be accessed into the action's parentheses: $Action(request.attr1, response.attr2)$ where $attr1$ is a request's attribute (such as $file\_extension$) and $attr2$ a response's attribute (such as $headers$, which is also a set), along with logical symbols such as or, and, and not. Actions can be manipulated in the $then$ part of the rule, as shown in Figure 3.4. The $retract$ keyword is used to remove a fact from the knowledge base.

After the instantiation of the Layer 1 rules, we obtain a formatted and filtered trace set ST composed of valued actions. At this point, we are ready to extract the first IOSTS.

Completeness, soundness, and complexity of Layer 1. HTTP traces are sequences of valued actions modeled with positive facts, which form Horn clauses [Hor51]. Furthermore, inference rules are Modus Ponens (soundness hypothesis introduced previously). Consequently, Layer 1 is sound and complete [MH07]. Keeping in mind the finite complexity hypothesis, its complexity is proportional to $\mathcal{O}(m(k+1))$ with m the number of valued actions, and k the number of rules (worst case scenario is when every valued action is covered $k+1$ times in the expert system engine).

[4]Portable Network Graphics

```
rule "Filter PNG images"
when
  $valued_action: Get(
    request.file_extension = 'png' or
    response.headers.content_type = 'image/png'
  )
then
  retract($valued_action)
end
```

Fig. 3.4: Filtering rule example that retracts the valued actions related to PNG images based on the request's file extension.

3.3.2 Layer 2: IOSTS transformation

An IOSTS is also associated with an Input/Output Labeled Transition System (IOLTS) to formulate its semantics [Rus+05; Fra+05; Fra+06]. Intuitively, IOLTS semantics correspond to valued automata without symbolic variables, which are often infinite. IOLTS states are labeled by internal variable valuations, and transitions are labeled by actions and parameter valuations. The semantics of an IOSTS $\mathcal{S} =< L, l0, V, V0, I, \Lambda, \rightarrow >$ is the IOLTS $[\![\mathcal{S}]\!] =< S, s_0, \Sigma, \rightarrow >$, where:

- $S = L \times D_V$ is a set of valued states;
- $s_0 = (l0, V0) \in S$ is the initial state;
- Σ is a set of valued actions;
- $\rightarrow$ is the transition relation.

Given an IOSTS transition $l_1 \xrightarrow{a(p),G,A} l_2$, we obtain an IOLTS transition $(l_1, v) \xrightarrow{a(p),\alpha} (l_2, v')$ with v a set of valuations over the internal variable set if there exists a parameter valuation set α such that the guard G evaluates to true with $v \cup \alpha$. Once the transition is executed, the internal variables are assigned with v' derived from the assignment $A(v \cup \alpha)$. Our IOSTS transformation relies upon this IOLTS semantics transformation that we have reversed. In order to generate the first IOSTS denoted by $\mathcal{S}_1$, the associated runs are first computed from the structured traces by injecting states between valued actions. These steps are detailed below.

Traces to runs

Given a structured HTTP trace $\sigma = (a_1(p), \alpha_1) \dots (a_n(p), \alpha_n)$, a run r is first derived by constructing and injecting states on the right and left sides of each valued action $(a_i(p), \alpha_i)$ of σ. Keeping in mind the IOLTS semantics definition, a state shall be modeled by the couple $((URI, k), v_\emptyset)$ with $v_\emptyset$ the empty valuation. (URI, k) is a couple composed of a URI and an integer $(k \geq 0)$, which shall be a location of the future IOSTS. All the runs r of the run set SR start with the same state $(l0, v_\emptyset)$. Then, a run is constructed by incrementally covering one trace: for an action $(a_i(p), \alpha_i)$ found in a trace, we extract the valuation $URI := val$ from α_i giving the URI value of the next resource reached after the action a_i, and we complete the current run r with $(a_i(p), \alpha_i)$ followed by the state $((val, k), v_\emptyset)$. Since we wish to preserve the sequential order of the actions found in the traces, when a URI previously encountered is once more detected, the resulting state is composed of the URI accompanied with an integer k, which is incremented to yield a new and unique state (Proposition 9).

The translation of the structured traces into a run set is performed by Algorithm 1, which takes a trace set ST as input, and returns a run set SR. It owns a set $States$ storing the constructed states. All the runs r of SR start with the same *initial state* $(l0, v_\emptyset)$. Algorithm 1 iterates over the valued actions $(a_i(p), \alpha_i)$ of a trace σ in order to construct the next states s. It extracts the valuation $URI := val$ (line 10) from α_i giving the URI value of the next resource reached after the action a_i. The state $s = ((val, k+1), v_\emptyset)$ is constructed with k such that there exists $((URI, k), v_\emptyset) \in States$ composed of the greatest integer $k \geq 0$. The current run r is completed with the valued action $(a_i(p), \alpha_i)$ followed by the state s (line 16). The operator $\cdot$ denotes this completion. Finally, SR gathers all the constructed runs (line 17).

Proposition 9 *For each trace $\sigma \in ST$, Algorithm 1 constructs a run $r = s_0 \cdot (a_0(p), \alpha_0) \cdot s_1 \cdot \dots \cdot s_n \cdot (a_n(p), \alpha_n) \cdot s_{n+1} \in SR$ such that $\forall (a_i(p), \alpha_i)_{(0<i\leq n)} \in \sigma, \exists\,!\, s_i \cdot (a_i(p), \alpha_i) \cdot s_{i+1}$ in r.*

IOSTS generation

The first IOSTS $\mathcal{S}_1$ is now derived from the run set SR in which runs are disjoint except for the initial state $(l0, v_\emptyset)$. Runs are translated into IOSTS paths that are assembled together by means of a disjoint union. The IOSTS forms a tree compound of paths, each expressing one trace, and starting from the same initial location.

Algorithm 1: Traces to runs algorithm

Input : Trace set ST
Output : Run set SR

```
1  BEGIN;
2  States = ∅ is the set of the constructed states;
3  if ST is empty then
4      SR = {(l0, v_∅)};
5  else
6      SR = ∅;
7      foreach trace σ = (a_0(p), α_0)...(a_n(p), α_n) ∈ ST do
8          r = (l0, v_∅);
9          for 0 ≤ i ≤ n do
10             extract the valuation URI := val from α_i;
11             if ((val, 0), v_∅) ∉ States then
12                 s = ((val, 0), v_∅);
13             else
14                 s = ((val, k + 1), v_∅) with k ≥ 0 the greatest integer such that
                   ((val, k), v_∅) ∈ States;
15             States = States ∪ {s};
16             r = r · (a_i(p), α_i) · s;
17         SR = SR ∪ {r};
18 END;
```

Definition 10 (IOSTS tree) *Given a run set SR, the IOSTS $\mathcal{S}_1$ is called the* IOSTS tree *of SR, and corresponds to the tuple $< L_{\mathcal{S}_1}, l0_{\mathcal{S}_1}, V_{\mathcal{S}_1}, V0_{\mathcal{S}_1}, I_{\mathcal{S}_1}, \Lambda_{\mathcal{S}_1}, \rightarrow_{\mathcal{S}_1} >$ such that:*

- $L_{\mathcal{S}_1} = \{l_i \mid \exists\, r \in SR, (l_i, v_\emptyset)$ *is a state found in* $r\}$;
- $l0_{\mathcal{S}_1}$ *is the initial location such that* $\forall\, r \in SR$, r *starts with* $(l0_{\mathcal{S}_1}, v_\emptyset)$;
- $V_{\mathcal{S}_1} = \emptyset$;
- $V0_{\mathcal{S}_1} = v_\emptyset$;
- $I_{\mathcal{S}_1}$ *is a finite set of parameters, disjoint from* $V_{\mathcal{S}_1}$;
- $\Lambda_{\mathcal{S}_1} = \{a_i(p) \mid \exists\, r \in SR, (a_i(p), \alpha_i)$ *is a valued action in* $r\}$;
- $\rightarrow_{\mathcal{S}_1}$ *is defined by the inference rule given below, applied to every run* $r = s_0 \cdot (a_0(p), \alpha_0) \cdot s_1 \cdot \cdots \cdot s_i \cdot (a_i(p), \alpha_i) \cdot s_{i+1} \in SR$:

$$\frac{s_0{=}(l0_{\mathcal{S}_1},v_\emptyset),\ s_i{=}(l_i,v_\emptyset),\ s_{i+1}{=}(l_{i+1},v_\emptyset),\ G_i{=}\bigwedge_{(x_i:=v_i)\in\alpha_i}(x_i == v_i)}{l0 \xrightarrow{a_i(p),G_i,(x:=x)_{x\in V}}_{\mathcal{S}_1} l_{i+1}}$$

The inference rule given in the definition above states that each run $r \in SR$ is transformed into a unique *IOSTS path*, only sharing a common initial location $l0$. And for each path, the guard is defined as the conjunction of the assignments of the variables found in the corresponding trace.

IOSTS minimization

Methods such as gkTail and KLFA could be applied to reduce the previous inferred IOSTS tree, but due to the way these methods work, it would likely lead to an over-generalized model (cf. Chapter 2 • Section 2.2.2 (page 38)). We prefer rejecting such solutions to preserve the trace equivalence [PY06] of the IOSTS $\mathcal{S}_1$ against the structured trace set ST before applying inference rules. Instead, we propose the use of a minimization technique that does not create over-generalization.

This IOSTS tree can be reduced in term of location size by applying a bisimulation minimization technique that still preserves the functional behaviors expressed in the original model. This minimization constructs the state sets (blocks) that are bisimilar equivalent [Par81]. Two states are said bisimilar equivalent, denoted by $q \sim q'$ if and only if they simulate each other and go to states from where they can simulate each other again. Two bisimulation minimization algorithms are given in [Fer89; Abd+06].

When receiving new traces from the *Monitor* (cf. Figure 3.1), the model yield by this layer is not fully regenerated, but rather completed on-the-fly. New traces are translated into IOSTS paths that are disjoint from $\mathcal{S}_1$ except from the initial location. We perform a union between $\mathcal{S}_1$ and IOSTS paths. Then, the resulting IOSTS is minimized again.

Completeness, soundness, complexity. Layer 2 takes any structured trace set obtained from HTTP traces. If the trace set is empty then the resulting IOSTS $\mathcal{S}_1$ has a single location $l0$. A structured trace set is translated into an IOSTS in finite time: every valued action of a trace is covered once to construct states, then every run is lifted to the level of one IOSTS path starting from the initial location. Afterwards, the IOSTS is minimized with the algorithm presented in [Fer89]. Its complexity is proportional to $\mathcal{O}(mlog(m+1))$ with m the number of valued actions. The soundness of Layer 2 is based upon the notion of traces: an IOSTS $\mathcal{S}_1$ is composed of transition sequences derived from runs in SR, itself obtained from the structured

trace set ST. $\mathcal{S}_1$ is a reduction of ST (trace inclusion) as defined in [PY06], *i.e.* $Traces(\mathcal{S}_1) \subseteq Traces(ST)$.

Example 3.3.2 We take as example a trace obtained from the GitHub website [5] after having executed the following actions: login with an existing account, choose an existing project, and logout. These few actions already produced a large set of requests and responses. Indeed, a web browser sends thirty HTTP requests on average in order to display a GitHub page. The trace filtering from this example returns the following structured traces where the request and response parts are concealed for readability purpose:

```
Get(https://github.com/)
Get(https://github.com/login)
Post(https://github.com/session)
Get(https://github.com/)
Get(https://github.com/willdurand)
Get(https://github.com/willdurand/Geocoder)
Post(https://github.com/logout)
Get(https://github.com/)
```

After the application of Layer 2, we obtain the IOSTS given in Figure 3.5. Locations are labeled by the URI found in the request and by an integer to keep the tree structure of the initial traces. Actions are composed of the HTTP verb enriched with the variables URI, request, and response. This IOSTS exactly reflects the trace behavior but it is still difficult to interpret. More abstract actions shall be deduced by the next layers.

[5] `https://github.com/`

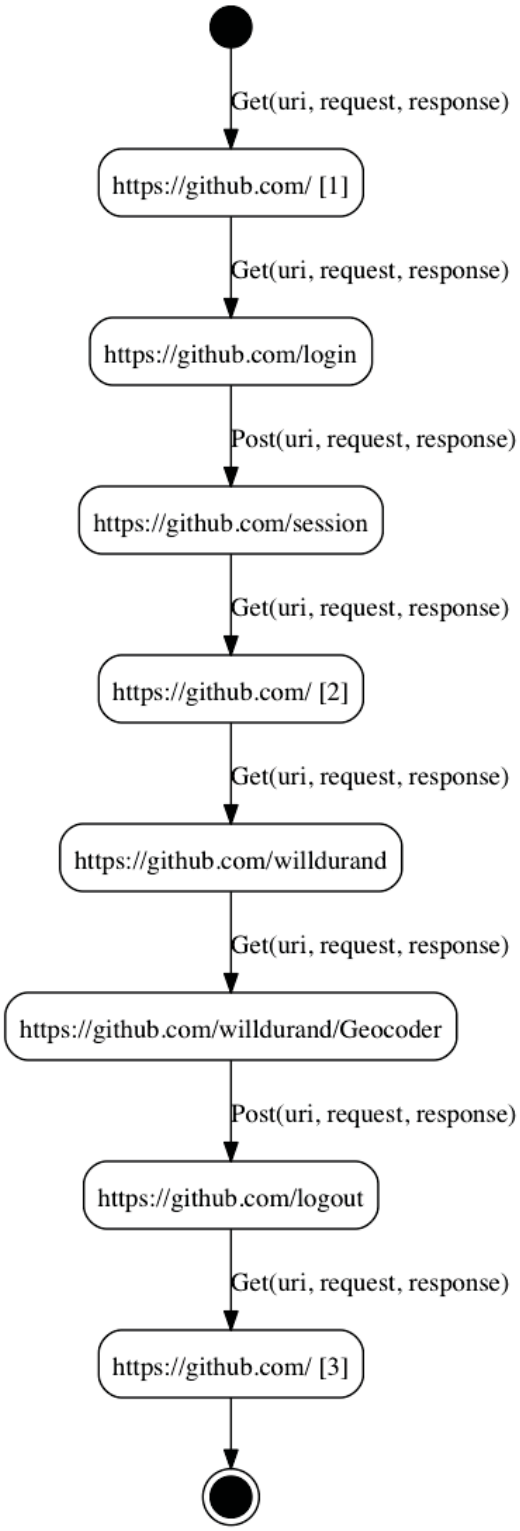

Fig. 3.5: IOSTS $\mathcal{S}_1$ obtained after the application of Layer 2.

3.3.3 Layers 3-N: IOSTS abstraction

As stated earlier, the rules of the upper layers analyze the transitions of the current IOSTS for trying to enrich its semantics while reducing its size. Given an IOSTS $\mathcal{S}_1$, every next layer carries out the following steps:

1. Apply the rules of the layer and infer a new knowledge base;
2. Apply a bisimulation minimization;
3. Store the resulting IOSTS.

Without loss of generality, we now restrict the rule structure to keep a link between the generated IOSTSs. Thereby, every rule of Layer i ($i \geq 3$) either enriches the sense of the actions (transition per transition) or aggregates transition sequences into one unique new transition to make the resulting IOSTS more abstract. It results in an IOSTS $\mathcal{S}_{i-1}$ exclusively composed of some locations of the first IOSTS $\mathcal{S}_1$. Consequently, for a transition or path of $\mathcal{S}_{i-1}$, we can still retrieve the concrete path of $\mathcal{S}_1$. This is captured by the following proposition:

Proposition 11 *Let $\mathcal{S}_1$ be the first IOSTS generated from the structured trace set ST. The IOSTS $\mathcal{S}_{i-1}(i > 1)$ produced by Layer i has a location set $L_{\mathcal{S}_{i-1}}$ such that $L_{\mathcal{S}_{i-1}} \subseteq L_{\mathcal{S}_1}$.*

Completeness, soundness, complexity. The knowledge base is exclusively composed of (positive) transition facts that have a Horn form. The rules of these layers are Modus Ponens (soundness hypothesis). Therefore, these inference rules are sound and complete. With regard to the (no implicit knowledge elimination) hypothesis and to Proposition 11, the transitions of $\mathcal{S}_{i-1}$ are either unchanged, enriched, or combined together into a new transition. The application of these layers ends in a finite time (finite complexity hypothesis).

Layer 3

This layer includes a (potentially empty) set of generic rules that can be applied to a large set of applications sharing similarities, *e.g.*, applications written with the same framework or same programming language. This layer has two roles:

- The **enrichment of the meaning captured in transitions**. In this step, we chose to mark the transitions with new internal variables. These shall help deduce more abstract actions in the upper layers. These rules are of the form:

```
rule "Layer 3 rule"
when
  $t: Transition(conditions on action, Guard, Assign)
then
  modify ($t) {
    Assign.add(new assignment over internal variables)
  }
end
```

$\$t$ is a transition object, like the valued actions in the previous rules. The $modify$ keyword allows to change information on a given object.

For example, the rules depicted in Figure 3.6 aims at recognizing the receipt of a login or logout page. The first rule means that if the response content, which is received after a request sent with the GET method, contains a login form, then this transition is marked as a "login page" with the assignment on the variable $isLoginPage$;

```
rule "Identify Login page"
when
  $t: Transition(
    Action == "GET",
    Guard.response.content contains('login-form')
  )
then
  modify ($t) {
    Assign.add("isLoginPage := true")
  }
end
```

```
rule "Identify Logout action"
when
  $t: Transition(
    Action == "GET",
    Guard.uri matches("/logout")
  )
then
  modify ($t1) {
    Assign.add("isLogout := true")
  }
end
```

Fig. 3.6: Login page and logout action recognition rules. The first rule adds a new assignment to any transition having a response's content containing a login form. The second transition adds a new assignment to all transitions where the uri (guard) matches $/logout$, identifying logout actions.

- The **generic aggregation of some successive transitions**. Here, some transitions (two or more) are analyzed in the conditional part of the rule. When the rule condition is met, then the successive transitions are replaced by one transition carrying a new action. These rules have the following form:

```
rule "Simple aggregation"
when
  $t1: Transition(
    conditions on action, Guard, ..., $lfinal := Lfinal
  )
  $t2: Transition(Linit == $lfinal, conditions)
  not exists Transition(
    Linit == $lfinal and Lfinal != $t2.Lfinal
  )
then
  insert(new Transition(
    new Action(),
    Guard($t1.Guard, t2.Guard),
    Assign($t1.Assign, $t2.Assign),
    Linit := $t1.Linit,
    Lfinal := $t2.Lfinal
  ))
  retract($t1)
  retract($t2)
end
```

Both $\$t1$ and $\$t2$ are transition objects. Transitions own guards ($Guard$ attribute), assignments ($Assign$ attribute), but also two locations: $Linit$ and $Lfinal$. In order to aggregate two consecutive transitions, we have to add a condition on the initial location of the second transition ($Linit == \$lfinal$), *i.e.* the final location of the first transition must be the initial location of the second transition, otherwise these two transitions are not consecutive. The third line in the LHS part ensures the absence of any other transition following $t1$ that is not $t2$. The $insert$ keyword in the $then$ part is used to add a new fact to the knowledge base. Here, we add a new transition with a more meaningful action (that reflects the aggregation), the unions of the guards and assignments of both transitions $\$t1$ and $\$t2$, and we set the initial location of this new transition to the initial location of the first transition and we set the final location of this new transition to the final location of the section transition.

```
rule "Identify Redirection after a POST"
when
  $t1: Transition(
    Action == "POST",
    (Guard.response.status = 301 or Guard.response.status = 302),
    $t1final := Lfinal
  )
  $t2: Transition(
    Action == "GET", Linit == $t1final
  )
  not exists Transition(
    Linit == $t1final and Lfinal != $t2.Lfinal
  )
then
  insert(new Transition(
    "PostRedirection",
    Guard($t1.Guard, $t2.Guard),
    Assign($t1.Assign, $t2.Assign),
    $t1.Linit,
    $t2.Lfinal
  )
  retract($t1)
  retract($t2)
end
```

Fig. 3.7: An inference rule that represents a simple aggregation identifying a redirection after a $POST$ request, leading to the creation of a new $PostRedirection$ transition.

The rule given in Figure 3.7 corresponds to a simple transition aggregation. It aims at recognizing the successive sending of information with a POST request followed by a redirection to another web page. If a request sent with the $POST$ method has a response identified as a redirection, (identified by the status code 301 or 302), and a GET request comes after, both transitions are reduced into a single one carrying the new action $PostRedirection$. Just like valued actions, guards can be accessed: $Guard.something$ where $something$ is a guard. The Drools rule language also provides keywords such as $contains$ or $match$ to deal with the values.

Example 3.3.3 When we apply these rules on the IOSTS example given in Figure 3.5, we obtain a new IOSTS illustrated in Figure 3.8. Its size is reduced since it has 6 transitions instead of 8 previously. Nonetheless, this new IOSTS does not precisely reflect the initial scenario yet (it is not as straightforward and understandable as what a project manager or a customer would express for instance). Rules deducing more abstract actions are required. These are found in the next layer.

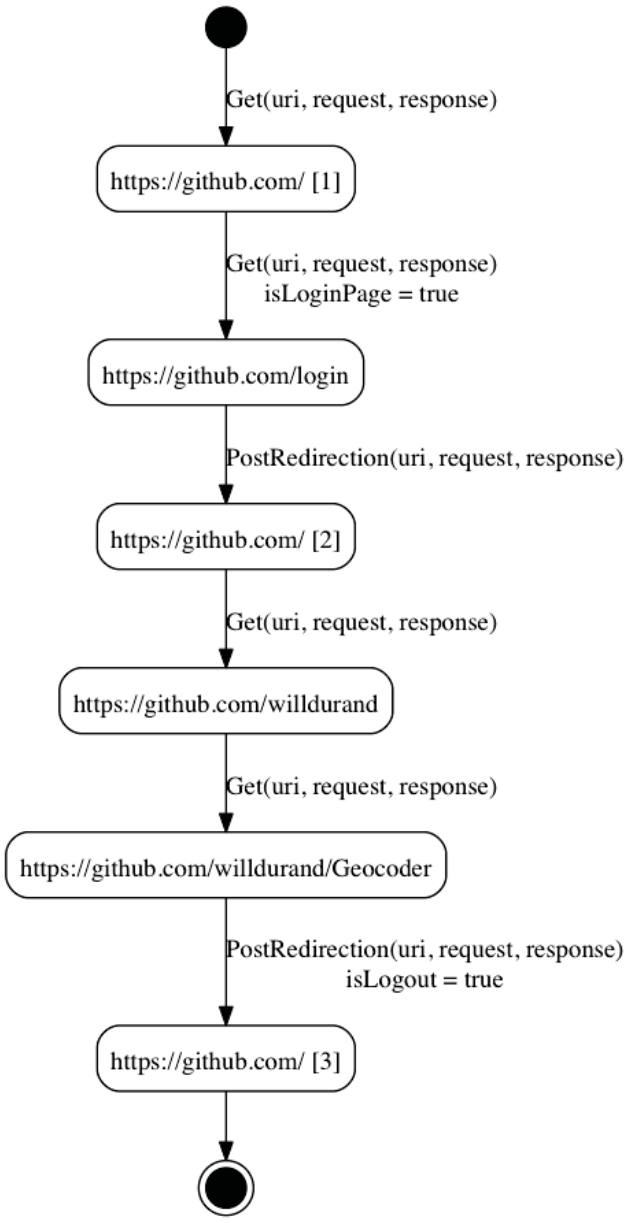

Fig. 3.8: IOSTS $\mathcal{S}_2$ obtained after the application of Layer 3.

Layer 4

This layer aims at inferring a more abstract model composed of more expressive actions and whose size should be reduced. Its rules may have different forms:

- They can be applied to a single transition only. In this case, the rule replaces the transition action to add more sense to the action. The rule given in Figure 3.9 is an example, which recognizes a user "deauthentication", *i.e.* when a user logs out, and adds a new action $Deauthentication$. This rule means that if a $PostRedirection$ action is triggered against a "Logout" endpoint (given by the variable *isLogout* added by Layer 3), then this is a deauthentication;

- The rules can also aggregate several successive transitions up to complete paths into one transition labeled by a more abstract action. For instance, the rule illustrated in Figure 3.10 recognizes a user authentication thanks to the variable $isLoginPage$ added by Layer 3. This rule means that if a "login" page is displayed, followed by a redirection triggered by a $POST$ request, then this is an authentication step, and the two transitions are reduced into a single one composed of the action $Authentication$.

Other rules can also be application-specific, so that these bring specific new knowledge to the model. For instance, the GitHub web application has a dedicated URL grammar (a.k.a. routing system). GitHub users own a profile page that is available at: *https://github.com/*{*username*} where {*username*} is the nickname of the user. However, some items are reserved, *e.g.*, *edu* and *explore*. This is typically the piece of information a human expert would give us, and that has been transliterated in the rule given in Figure 3.11, whose aim is to produce a new action $ShowProfile$ offering more sense. Similarly, a GitHub page describing a project has a URL that always matches the pattern: *https://github.com/*{*username*}*/*{*project_name*}. The rule given in Figure 3.12 captures this pattern and derives a new action named $ShowProject$.

Example 3.3.4 The application of the four previous rules leads to the final IOSTS depicted in Figure 3.13. Now, it can be used for application comprehension since most of its actions have a precise meaning, and describe the application's behaviors.

```
rule "Identify Deauthentication"
when
  $t: Transition(
    action == "PostRedirection",
    Assign contains "isLogout := true"
  )
then
  modify ($t) {
    setAction("Deauthentication")
  }
end
```

Fig. 3.9: This rule recognizes an action that we call "deauthentication", *i.e.* when a user logs out.

```
rule "Identify Authentication"
when
  $t1: Transition(
    Action == "GET",
    Assign contains "isLoginPage := true",
    $t1final := Lfinal
  )
  $t2: Transition(
    Action == "PostRedirection",
    Linit == $t1final
  )
  not exists Transition(
    Linit == $t1final and Lfinal != $t2.Lfinal
  )
then
  insert(new Transition(
    "Authentication",
    Guard($t1.Guard,$t2.Guard),
    Assign($t1.Assign, $t2.Assign),
    $t1.Linit,
    $t2.Lfinal
  )
  retract($t1)
  retract($t2)
end
```

Fig. 3.10: Authentication recognition by leveraging information carried by the rule given in Figure 3.7. When a user browses a web page containing a login form, following by a $PostRedirection$, this is an $Authentication$ action.

```
rule "GitHub profile pages"
when
  $t: Transition(
    action == "GET",
    (Guard.uri matches "/[a-zA-Z0-9]+$",
     Guard.uri not in [ "/edu", "/explore" ])
  )
then
  modify ($t) {
    setAction("ShowProfile")
  }
end
```

Fig. 3.11: User profile recognition. This rule is specific to the application under analysis, and works because profile pages are anything but */edu* and */explore*.

```
rule "GitHub project pages"
when
  $t: Transition(
    action == "GET",
    Guard.uri matches "/[a-zA-Z0-9]+/.+$",
    $uri := Guard.uri
  )
then
  String s = ParseProjectName($uri)
  modify ($t) {
    setAction("ShowProject")
    Assign.add("ProjectName := " + s)
  }
end
```

Fig. 3.12: Project choice recognition. Here again, this is a specific rule for the application under analysis that works because the routing of this application defines projects at URIs matching */{username}/{project name}*.

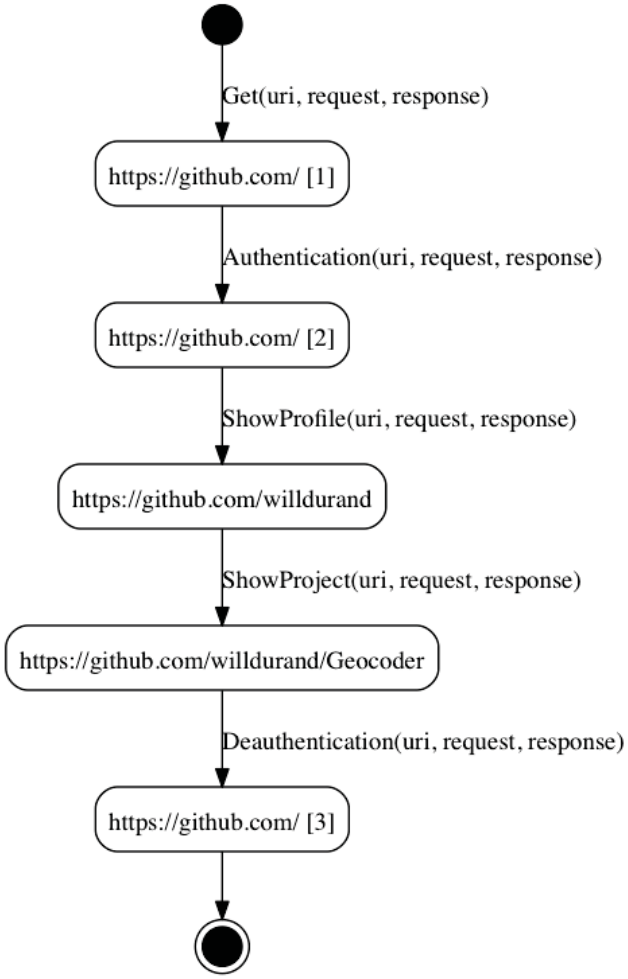

Fig. 3.13: The final IOSTS $\mathcal{S}_3$ obtained after the application of Layer 4.

3.4 Getting more samples by automatic exploration guided with strategies

Because the inference of our models depends on both the amount and the quality of the collected traces, these models are said partial. In order to improve the completeness of our inferred models, we rely on an automatic testing technique (that is, crawling) to interact with the application. This is the role of the *Robot explorer*, driven by the Models generator.

Rather than using a static traversal strategy as in [Mem+03; Ana+12; Mes+12; Ama+12; Yan+13], we propose the addition of an orthogonal layer in the Models generator to describe any kind of exploration strategy. This layer is compound of an algorithm that chooses the next states to explore with respect to a given exploration strategy, modeled with inference rules.

The simplified algorithm of the Strategy layer is given in Algorithm 2. The latter applies the rules on any stored IOSTS $\mathcal{S}_i$ chosen by the user. It emerges a location list Loc that are marked with "explored" by the rules to avoid re-using them twice (line 4). Then, the algorithm goes back to the first generated IOSTS $\mathcal{S}_1$ in order to extract one complete and executable path p ended by a location l of Loc (line 5). This step is sound since all the locations of $\mathcal{S}_i$ belong to the location set of $\mathcal{S}_1$ (Proposition 11). Such an IOSTS *preamble* is required by the Robot explorer for trying to reach the location l by executing every action of p. The algorithm finally returns a list of paths $List$, which is sent to the Robot explorer. The exploration ends once all the locations of $\mathcal{S}_i$ or of $\mathcal{S}_1$ are visited (line 3). The algorithm returns unexplored locations even if, while executing the algorithm, the IOSTS $\mathcal{S}_i$ is regenerated several times because the marked locations are also stored in the set L. Hence, if a location of $\mathcal{S}_i$ is chosen a second time by the rules, the algorithm checks whether it has been previously visited (line 7).

The rules of the Strategy layer can encode different strategies. We propose two examples below:

- **Classical traversal strategies** (*e.g.*, Depth-First Search and Breadth-First Search) can still be performed. For example, Figure 3.14 depicts two rules expressing the choice the next location to explore in a breadth-wise order first. The initial location $l0$ is chosen and marked as explored (rule BFS). Then, the transitions having an initial location marked as explored and a final location not yet explored are collected by the rule BFS2, except for the transitions carrying an HTTP error (response status upper or equal to 400). The $accumulate$ keyword is a Drools function that allows to operate on sets of data, which we

Algorithm 2: Exploration strategy

Input : IOSTS $\mathcal{S}_1$, $\mathcal{S}_i$
Output : List of preambles $List$

```
1  L := ∅ is the list of explored locations of S_1;
2  BEGIN;
3  while L ≠ L_{S_1} and L ≠ L_{S_i} do
4      Apply the rules on S_i and extract a location list Loc;
5      Go back to S_1;
6      foreach l ∈ Loc do
7          if l ∉ L then
8              Compute a preamble p from l0_{S_1} that reaches l;
9              L := L ∪ {l};
10             List := List ∪ {p};
11 END;
```

used to transcribe the previous sentence. The selected locations are marked as explored in the IOSTS $\mathcal{S}_i$ with the method $SetExplored$ in the "then" part of the rule;

- **Semantic-driven strategies** could also be applied, when the meaning of some actions is recognizable. For instance, for e-commerce applications, the login step and the term "buy" are usually important. Thereby, a strategy targeting firstly the locations of transitions carrying theses actions can be defined by the rule "semantic-driven strategy" given in Figure 3.15. It is manifest that the semantic-driven strategy domain can be tremendously vast since it depends on the number of recognized actions and on their relevance.

Many other strategies could be defined in relation to the desired result in terms of model generation and application coverage. Other criteria, *e.g.*, the number of UI elements per Graphical User Interface (GUI) or the number of observed crashes could also be taken into consideration.

3.5 Implementation and experimentation

We implemented this technique in a prototype tool called *Autofunk v1*. A user interacts with *Autofunk* through a web interface and either gives a URL or a file containing traces. These traces have to be packaged in the HTTP Archive (HAR) format as it is the *de facto* standard to describe HTTP traces, used by various HTTP related tools. Such traces can be obtained from many HTTP monitoring tools (Mozilla Firefox or Google Chrome included). Then, *Autofunk* produces IOSTS models, which are stored in a database. The last model is depicted in a web interface.

```
rule "BFS"
when
  $l: Location(name == l0, explored == false)
then
  $l.SetExplored()
end
```

```
rule "BFS2"
when
  $Loc: ArrayList<Location>() from accumulate(
    $t: Transition(Guard.response.status > 199
      && Guard.response.status < 400
      && Linit.explored == true
      && Lfinal.explore==false
    ),
    init(ArrayList<Transition> Loc = new ArrayList<Transition>()),
    action(Loc.add($t.Lfinal)),
    result(Loc)
  )
then
  Loc.SetExplored()
end
```

Fig. 3.14: Two rules used to implement a Breadth-first search (BFS) exploration strategy.

The JBoss Drools Expert tool has been chosen to implement the rule-based system. Such an engine leverages Object-Oriented Programming in the rule statements and takes knowledge bases given as Java objects (*e.g.*, $Location$, $Transition$, Get, $Post$ objects in this work). Inference rules are written in *.drl* files, which makes it simple for human domain experts to write rules. These files are then read by the Drools engine during its initialization. Our prototype does not allow its user to add rules through the web interface though.

The GitHub website [6] is an example of application giving significant results. We recorded a trace set composed of 840 HTTP requests / responses. Then, we applied *Autofunk* to them with a Models generator composed of 5 layers gathering 18 rules whose 3 are specialized to GitHub. After having performed trace filtering (Layer 1), we obtained a first IOSTS tree composed of 28 transitions. The next 4 layers automatically inferred a last IOSTS tree $\mathcal{S}_4$, given in Figure 3.16, that is composed of 12 transitions from which 9 have a clear and intelligible meaning. Layers 4 and 5 include inference rules that are specific to GitHub. For other websites, these rules cannot be reused and have to be rewritten.

[6] `https://github.com/`

```
rule "semantic-driven strategy"
when
  $t: Transition (
    Assign contains "isLogin := true"
    or Guard.response matches "*buy*"
  )
then
  ArrayList Loc = new ArrayList()
  Loc.add($t.Linit, $t.Lfinal)
  Loc.SetExplored()
end
```

Fig. 3.15: A semantic-driven exploration strategy that focuses on the term "buy" in the HTTP responses, *i.e.* displayed the web pages.

Based on the IOSTS tree $\mathcal{S}_4$, we also built an over-generalized model $\mathcal{S}_5$ by merging similar states (*i.e.* those having the same URI). Figure 3.17 depicts this model, which is now close to a UML state diagram. Such a model $\mathcal{S}_5$ cannot be used for testing, yet it becomes interesting for documentation purpose because its representation reflects what a user uses to work with when it comes to write documentation (*i.e.* UML diagrams).

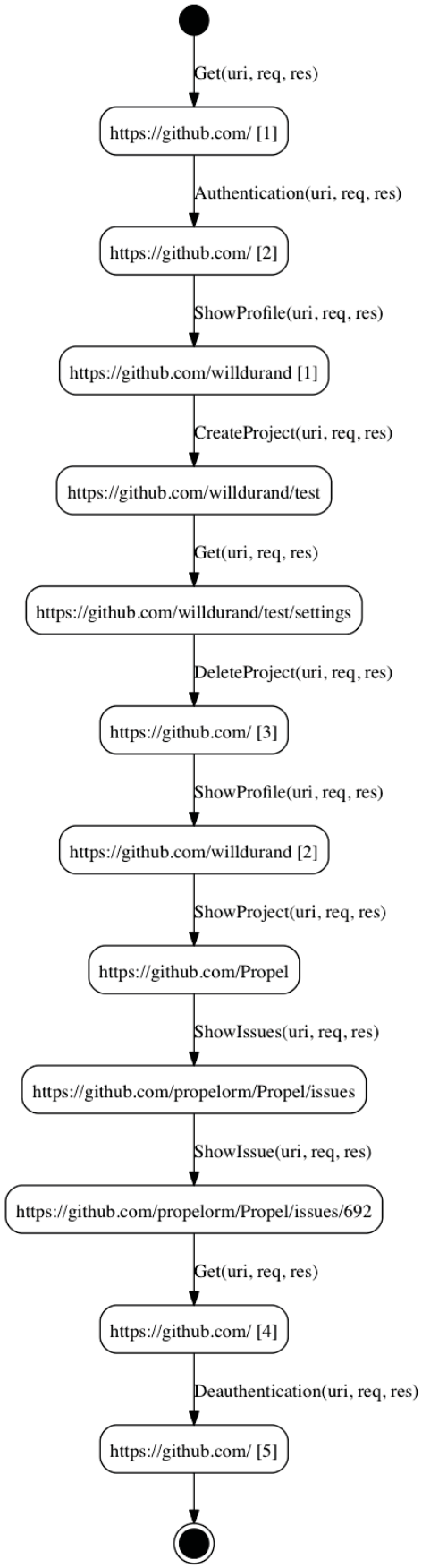

Fig. 3.16: This model is the IOSTS $\mathcal{S}_4$ obtained from a trace set composed of 840 HTTP requests and responses, and after the application of 5 layers gathering 18 rules.

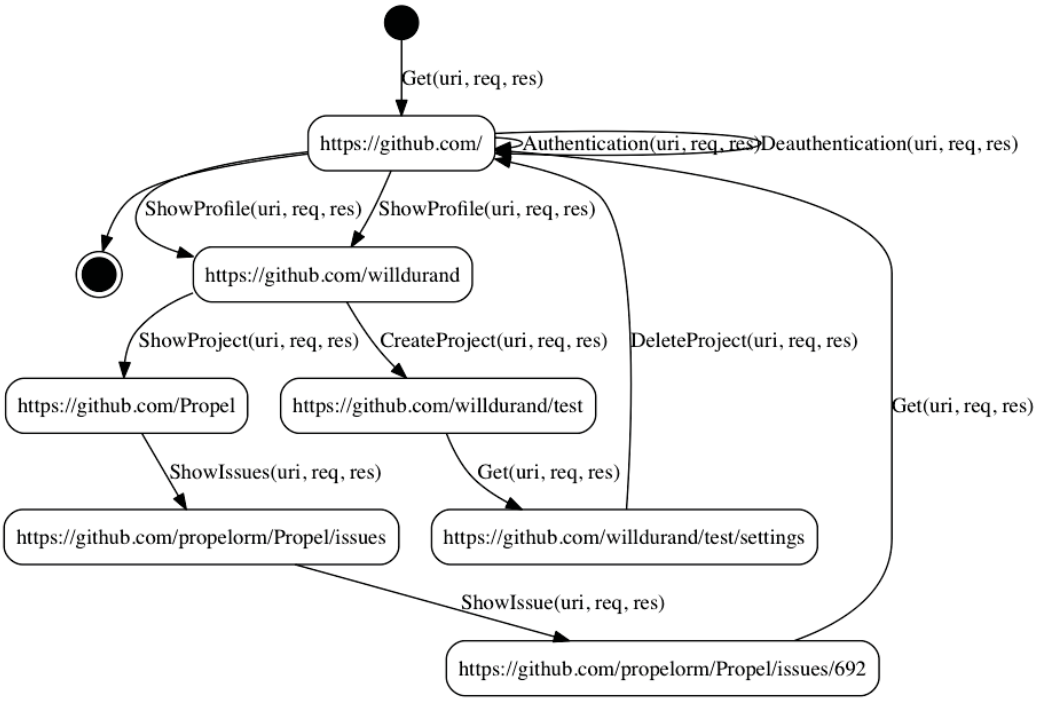

Fig. 3.17: This model $\mathcal{S}_5$ is the over-generalization of the model $\mathcal{S}_4$.

3.6 Conclusion

In this chapter, we presented an original approach combining model inference, expert systems, and automatic testing to derive IOSTSs (Input/Output Symbolic Transition Systems) models by means of a model transformation using inference rules. We chose a minimization technique over existing algorithms such as gkTail and KLFA to keep exactness of the inferred models. Our proposal yields several models, reflecting different levels of abstraction of the same application with the use of inference rules that capture the knowledge of a human expert. The first contribution lies in the flexibility and scalability brought by the inference rules since they can be applied to several applications or on a single application only when the rules are specific. The whole framework does not have to be re-implemented for each application, but inference rules have to. Our approach can be applied to event-driven applications since our framework supports their exploration. Furthermore, it can also be applied to other kinds of application as far as they produce traces.

Combining expert systems (gathering knowledge) and formal models is really interesting but this is not a silver bullet. Indeed, writing rules is still a heavy task, and thus not suitable as is. It is almost as difficult as writing a formal model, hence the need for an automated way to write the rules of Layers 3 to N. Ideally, the framework itself should generate a set of rules, *e.g.*, using a machine learning technique, or at

least ease the process of writing and managing these rules, but we did not pursue this path.

We chose to model behaviors of web applications using IOSTSs. Yet, we did not leverage is main characteristic, *i.e.* the distinction between input and output actions. At first, we thought that distinguishing HTTP requests from HTTP responses would have made sense but it was a mistake. Indeed, to represent a full action of a web application, one cannot separate the HTTP request from its HTTP response. Our inferred IOSTSs are actually STSs, which is perfectly fine because IOSTS is a specialization of STS. In the sequel, we use STS models.

The first results on model inference were very encouraging, so we decided to rework our architecture to build a better framework on top of this one in order to construct models of production systems. That is the purpose of the next chapter.

4 Model inference for production systems

In this chapter, we introduce *Autofunk v2* and *Autofunk v3*. This is our passive model inference framework, improving *Autofunk v1*'s capabilities, and focused on inferring models of production systems. It combines model inference, machine learning, and expert systems to infer Symbolic Transition Systems (STSs). We replaced the minimization technique presented in the previous chapter by a context-specific reduction method, still preserving exactness of the inferred models, but more efficient in terms of processing time. This is illustrated by some experiments with *Autofunk v2*. These results conducted to the release of *Autofunk v3*. The main difference between *Autofunk v2* and *Autofunk v3* is the use of the *k-means clustering* method. This is a machine learning technique that we leverage to segment and filter input trace sets, so that inferred models contain complete behaviors only.

Contents

4.1 Introduction

In the Industry, building models for production systems, *i.e.* event-driven systems that run in production environments and are distributed over several devices and sensors, is frequent since these are valuable in many situations like testing and fault diagnosis for instance. Models may have been written as storyboards or with languages such as the *Unified Modeling Language* (UML) or even more formal languages. Usually, these models are designed when brand-new systems are built. It has been pointed out by our industrial partner that production systems have a life span of many years, up to 20 years, and are often incrementally updated, but their corresponding models are not. This leads to a major issue which is to keep these models up to date and synchronized with the respective systems. This is a common problem with documentation in general, and it often implies rather under-specified or not documented systems that no one wants to maintain because of lack of understanding.

In this chapter, we focus on this problem for production systems that exchange thousands of events a day. Several approaches have already been proposed for different types of systems. Yet, we noticed that these approaches were not tailored to support production systems. From the literature, Chapter 2 • Section 2.2 (page 26), we deduced the following key observations:

- Most of the existing model inference approaches give approximate models capturing the behaviors of a system and more. In our context, we want exact models that could be used for regression testing, and fault diagnosis;

- Applying active inference on production systems is complicated since these must not be disrupted (otherwise it may lead to severe damages on the production machines), but also because we cannot rely on any (human) oracle or teacher;

- Production systems exchange thousands and thousands events a day. Again, most of the model inference approaches cannot take such a huge amount of information to build models. We need a solution that is scalable.

Based on these observations, we propose a pragmatic *passive model inference* approach that aims at building formal models describing functional behaviors of a system. Our goal is to quickly build exact models from large amounts of production events. Furthermore, execution speed takes an important place for building up to date models. Such models could also be used for diagnosis every time an issue would be experienced in production. The strong originality of our approach lies in the combination of two domains for model inference: model-driven engineering and

expert systems. We consider formal models and their definitions to infer models by means of different transformations. But we also take into consideration the knowledge of human experts captured by expert systems. A part of our approach is based upon this notion of knowledge implemented with inference rules. We reuse what worked well for web applications and enhance many parts of our framework to come up with a better version of our *Autofunk* framework, which is presented throughout this thesis.

In the following section, we describe the context in which this work has been conducted. In Section 4.3, we present *Autofunk* for Michelin's production systems along with a case study. Section 4.4 highlights our work on improving usability of the generated models. We give our results in Section 4.5. We highlight an improvement made on our framework in Section 4.6, which is part of *Autofunk v3*. We conclude on this chapter in Section 4.7.

Publications. This work has been published in the Proceedings of Formal Methods 2015 (FM'15) [DS15a], and in the Proceedings of the 9th International Conference on Distributed Event-Based Systems (DEBS'15) [SD15].

4.2 Context at Michelin

Michelin is a worldwide tire manufacturer and designs most of its factories, production systems, and software by itself. Like many other industrial companies, Michelin follows the *Computer Integrated Manufacturing* (CIM) approach [RK04], using computers and software to control the entire manufacturing process. In this thesis, we focus on the Level 2 of the CIM approach, *i.e.* all the applications that monitor and control several production devices and *points*, *i.e.* locations where a production line branches into multiple lines, in a workshop. In a factory, there are different *workshops* for each step of the tire building process. At a workshop level, we observe a continuous stream of products from specific entry points to a finite set of exit points, *i.e.* where products go to reach the next step of the manufacturing process, and disappear of the workshop frame in the meantime, as shown in Figure 4.1. Depending on the workshops, products can stay in this frame for days up to several weeks. Indeed, a workshop may contain storage areas where products can stay for a while, depending on the production campaigns or needs for instance. Thousands and thousands of *production events* are exchanged among the industrial devices of the same workshop every day, allowing some factories to build over 30,000 tires a day.

Although there is a finite number of applications, each has different versions deployed in factories all over the world, potentially highlighting even more different behaviors

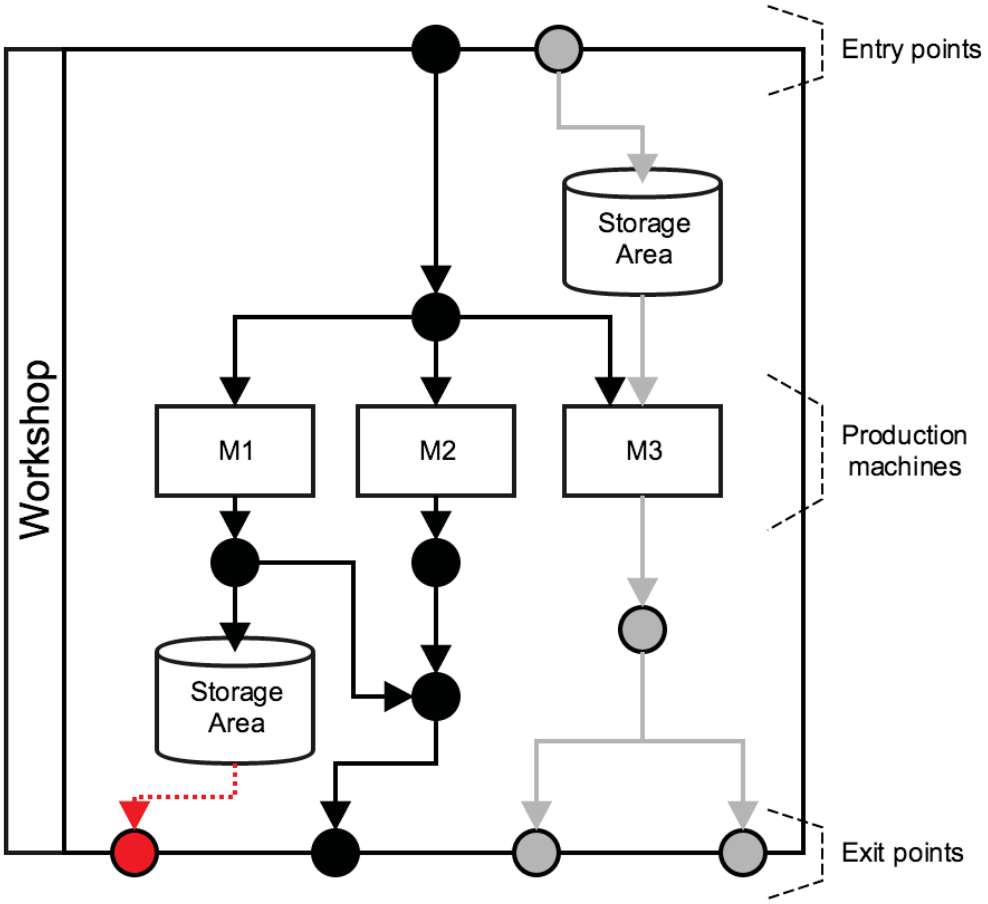

Fig. 4.1: A workshop owns a set of known entry and exit points. A continuous stream of products starts from known entry points, and ends at known exit points. This figure shows two production lines: the grey line having two exit points (represented by the circles), and the black one having only one real exit point, not two. The red dotted line represents a false positive here, which is a problem we have to take into account while detecting (entry and) exit points.

and features. Even if a lot of efforts are put into standardizing applications and development processes, different programming languages and different frameworks are used by development teams, making difficult to focus on a single technology. Last but not least, the average lifetime of these applications is 20 years. This set is large and too disparate to apply conventional testing techniques, yet most of the applications exchange events using dedicated custom internal protocols.

Our industrial partner needs a safe way to infer up to date models, independent of the underlying technical details, and without having to rely on any existing documentation. Additionally, Michelin is interested in building regression test suites to decrease the time required to deploy or upgrade systems. We came up to the conclusion that, in order to target the largest part of all Michelin's Level 2 applications, taking advantage of the production events exchanged among all devices

would be the best solution, as it would not be tied to any programming language or framework. In addition, these events contain all information needed to understand how a whole industrial system behaves in production. Event synchronization is guaranteed by Michelin's custom exchange protocols. All these events are collected synchronously through a (centralized) logging system. Such a system logs all events with respect to their order, and does not miss any event. From these, we chose not to use extrapolation techniques to infer models (*i.e.* model inference techniques that build over-approximated models), meaning our proposal generates exact models for testing purpose, exclusively describing what really happens in production.

This context leads to some assumptions that have been considered to design our framework:

- **Black-box systems:** production systems are seen as black-boxes from which a large set of production events can be passively collected. Such systems are compound of production lines fragmented into several devices and sensors. Hence, a production system can have several entry and exit points. We denote such a system by Sua (*system under analysis*);

- **Production events:** an event of the form $(a(p), \alpha)$ must include a distinctive label a along with an assignment α over the parameter set p. Two events $(a(p), \alpha_1)$ and $(a(p), \alpha_2)$ having the same label a must own assignments over the same parameter set p. The events are ordered and processed with respect to this order;

- **Traces identification:** traces are sequences of events $(a_1(p), \alpha_1) \ldots (a_n(p), \alpha_n)$. A trace is identified by a specific parameter that is included in all variable assignments of the same trace. This identifier is denoted by pid and identifies products, *e.g.*, tires at Michelin. Besides this, variable assignments include a time stamp to sort the events into the traces.

4.3 *Autofunk*'s models generator revisited

In this section, we introduce our framework *Autofunk v2*, whose main architecture is depicted in Figure 4.2. This new version also contains different modules (in grey in the figure): four modules are dedicated to build models, and an optional one can be used to derive more abstract and readable models.

Here, we consider *Symbolic Transition Systems* (STSs) as models for representing production system behaviors (cf. Chapter 2 • Section 2.1.2 (page 18)). As a reminder, STSs are state machines incorporating actions (*i.e.* events in this context), labeled on

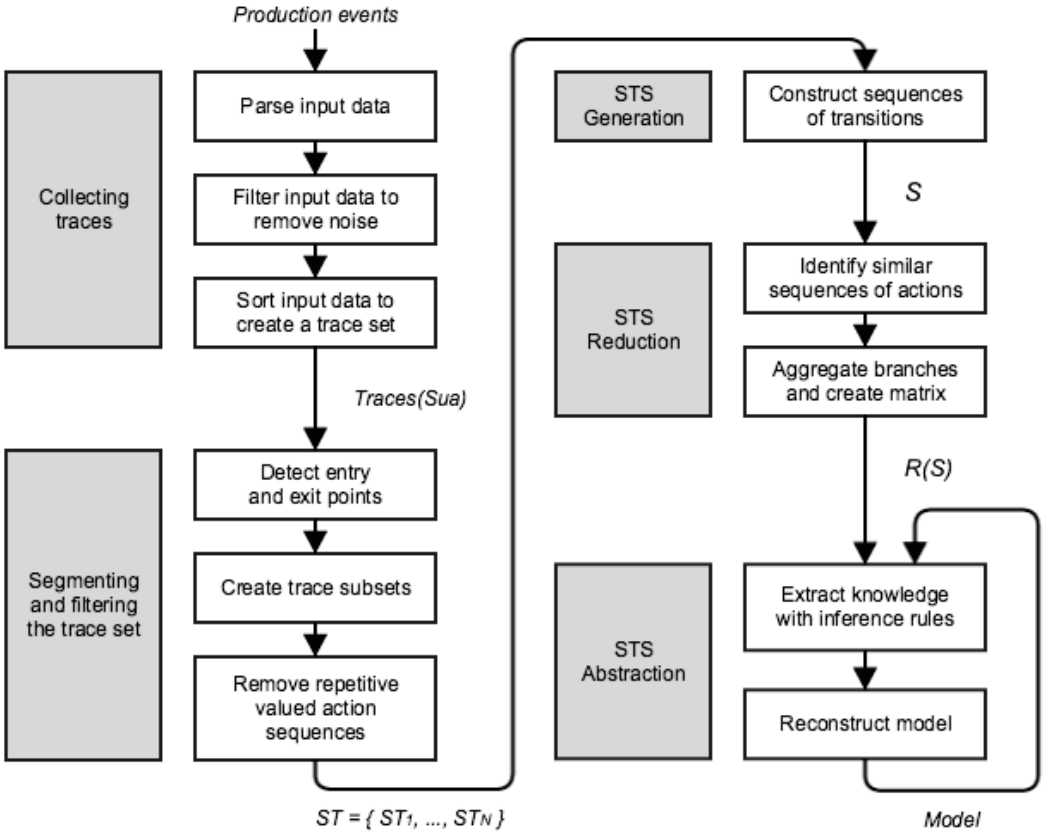

Fig. 4.2: Overview of *Autofunk v2*. It is a set of different modules (in grey) along with their corresponding steps. The last module (STS abstraction) is optional.

transitions, that show what can be given to and observed on the system. In addition, actions are tied to an explicit notion of data. In this chapter, we consider STSs of the form:

Definition 12 (Symbolic Transition System) *A STS is defined as a tuple* $< L, l0, V, V0, I, \Lambda, \rightarrow >$ *where:*

- L *is a countable set of locations;*
- $l0 \in L$ *is the initial location;*
- V *is a finite set of location (or internal) variables;*
- $V0$ *is* a condition *on the initialization of the location variables* V*;*
- I *is a finite set of parameters (also known as interaction variables), disjoint from* V*;*
- Λ *is a finite set of symbolic actions* $a(p)$ *with* a *a symbol, and* $p = (p_1, \ldots, p_k)$ *a finite set of parameters in* $I^k (k \in \mathbb{N})$*;*

- $\rightarrow$ *is a finite set of symbolic transitions. A symbolic transition* $t = (l_i, l_j, a(p), G, A) \in \rightarrow$*, from the location* $l_i \in L$ *to* $l_j \in L$*, also denoted by* $l_i \xrightarrow{a(p),G,A} l_j$*, is labeled by:*

  - *A symbolic action* $a(p) \in \Lambda$*;*
  - *A guard* G *over* $(p \cup V)$*, which restricts the firing of the transition. Throughout this thesis, we often consider guards written as* conjunctions *of equalities:* $\bigwedge_{x \in I \cup V} (x == val)$*;*
  - *An assignment* A *that defines the evolution of the variables,* A_x *being the function in* A *defining the evolution of the variable* $x \in V$*.*

We also denote by $proj_x(G)$ the projection of the guard G over the variable $x \in I \cup V$, which extracts the equality $(x == val)$ from G. For example, given the guard $G_1 = [(nsys == 1) \wedge (nsec == 8) \wedge (point == 1) \wedge (pid == 1)]$, $proj_{nsys}(G_1) = (nsys == 1)$.

Autofunk v2 is a better version of the work introduced in the previous chapter, targeting production systems. Given a system Sua and a set of production events, *Autofunk v2* builds *trace-included* models [PY06], *i.e.* the traces of a model $\mathcal{S}$ are included in the traces of Sua.

```
17-Jun-2014 23:29:59.00|INFO|New File

17-Jun-2014 23:29:59.50|17011|MSG_IN  [nsys: 1] \
  [nsec: 8] [point: 1] [pid: 1]

17-Jun-2014 23:29:59.61|17021|MSG_OUT [nsys: 1] \
  [nsec: 8] [point: 3] [tpoint: 8] [pid: 1]

17-Jun-2014 23:29:59.70|17011|MSG_IN  [nsys: 1] \
  [nsec: 8] [point: 2] [pid: 2]

17-Jun-2014 23:29:59.92|17021|MSG_OUT [nsys: 1] \
  [nsec: 8] [point: 4] [tpoint: 9] [pid: 2]
```

Fig. 4.3: An example of some production events. Each event is time-stamped, has a label (*e.g.*, 17011), and may own assignments of variables (*e.g.*, $nsys$).

Example 4.3.1 To explain how *Autofunk v2* works with production systems, we consider a case study based upon the example of Figure 4.3. It depicts simplified production events similar to those extracted from Michelin's logging system. $INFO$,

17011, and 17021 are labels along with assignments of variables *e.g.*, $nsys$, which indicates an industrial device number, and $point$, which gives the product position. Real events own around 20 parameters in average. Such a format is specific to Michelin but other kinds of events could be considered by updating the first module of *Autofunk*. In this example, each line represents an event that happened in the system. This format has been borrowed from Michelin's logging system, hence its special formatting. It is worth mentioning that the first line of this set is an event that is tied to the logging system, not the production system we are interested in.

4.3.1 Production events and traces

Autofunk v2 takes production events as input from a system under analysis Sua. To avoid disrupting the (running) system Sua, we do not instrument the industrial equipment composing the whole system. Everything is done offline with a logging system or with monitoring. We start by formatting these events to obtain a set of events of the form $(a(p), \alpha)$ with a a label, and α an assignment over the parameter set p. We call these formatted events, *valued events*. Valued events are similar to valued actions (as in Chapter 3) except that this term is more accurate in our industrial context. Performing such a step allows to collect productions events from various sources, and still be able to perform the next steps in a unique manner.

In this set, some of these valued events are irrelevant. For instance, some events may capture logging information and are not part of the functioning of the system. In Figure 4.3, the event having the type $INFO$ belongs to this category and can be safely removed. Filtering is achieved by an expert system and inference rules. Here again, a human expert knows which events should be filtered out, and inference rules offer a natural way to express his knowledge. We use the same form of inference rules as seen in Chapter 3, that is:

When $(a(p), \alpha)$*, condition on* $(a(p), \alpha)$*, then retract* $(a(p), \alpha)$.

Example 4.3.2 The remaining valued events are time-ordered to produce an initial set of traces denoted by $Traces(Sua)$. Figure 4.4 illustrates this set obtained from the events of Figure 4.3.

Figure 4.5 shows two concrete rules applied to Michelin systems. These two rules are written with the Drools formalism, already introduced in the previous chapter. The first rule removes valued events including the $INFO$ parameter, which do not contain any business value. The variable $\$valued\_event$ is a valued event (instance of $ValuedEvent$), containing parameter assignments ($Assign$ attribute) that can be

$$Traces(Sua) = (17011(\{nsys, nsec, point, pid\}), \{nsys := 1, nsec := 8, point := 1, pid := 1\})\ 17021(\{nsys, nsec, point, tpoint, pid\}, \{nsys := 1, nsec := 8, point := 3, tpoint := 8, pid := 1\})\ 17011(\{nsys, nsec, point, pid\}, \{nsys := 1, nsec := 8, point := 2, pid := 2\})\ 17021(\{nsys, nsec, point, tpoint, pid\}, \{nsys := 1, nsec := 8, point := 4, tpoint := 9, pid := 2\})$$

Fig. 4.4: Initial trace set $Traces(Sua)$ based on the events given in Figure 4.3.

accessed using the following notation: $Assign.foo$ where foo is a name and the result its corresponding value.

The second rule in Figure 4.5 removes valued events extracted from very specific events, *i.e.* those whose key matches a pattern and having a inc value that is not equal to 1. To fully understand the syntax of this rule, one has to know that when there is no logical connective between the conditions, it defaults to a conjunction. Such a rule has been given by a Michelin expert and allows to remove some duplicate events. In the context of Michelin, we use four inference rules to remove all irrelevant events.

```
rule "Remove INFO events"
when:
  $valued_event: ValuedEvent(Assign.type == TYPE_INFO)
then
  retract($valued_event)
end
```

```
rule "Remove events that are repeated"
when
  $valued_event: ValuedEvent(
    Assign.key matches "KEY_NAME_[0-9]+",
    Assign.inc != null,
    Assign.inc != "1"
  )
then
  retract($valued_event)
end
```

Fig. 4.5: An example of inference rules used for filtering purpose. It contains two rules: the first one is used to remove events including the $INFO$ label, the second one is used to omit events that are repeated.

From this filtered valued event base, we reconstruct the corresponding traces from the trace identifier pid, present in each valued event, and time stamps. We call the resulting trace set $Traces(Sua)$:

Definition 13 ($Traces(Sua)$) *Given a system under analysis Sua, $Traces(Sua)$ denotes its* formatted trace set. *$Traces(Sua)$ includes traces of the form $(a_1(p), \alpha_1) \dots (a_n(p), \alpha_n)$ such that $(a_i(p), \alpha_i)_{(1 \leq i \leq n)}$ are ordered valued events having the same identifier assignment.*

We can now state that a STS model $\mathcal{S}$ is said *trace-included* if and only if $Traces(\mathcal{S}) \subseteq Traces(Sua)$ as defined in [PY06].

4.3.2 Trace segmentation and filtering

In Section 4.2, we indicated that products could stay in a workshop for days and even weeks. From our point of view, this means that we can likely collect partial executions, *i.e.* events belonging to products that were present in the workshop we are observing before we start to collect any events. We can determine which product was present in the workshop before by retrieving the first event tied to it. If the physical location of this event is not one of the known entry points of the workshop, then the product was there before. In a similar manner, we are able to retrieve *incomplete executions* related to the end of the collection, *i.e.* the events tied to products that did not reach any of the known exit points of the workshop.

We define a *complete trace* as a trace containing all events expressing the path taken by a product in a production system, from the beginning, *i.e.* one of its entry points, to the end, *i.e.* one of its exit points. In the trace set $Traces(Sua)$, we do not want to keep incomplete traces, *i.e.* traces related to products which did not pass through one of the known entry points or moved to the next step of the manufacturing process using one of the known exit points. In addition, we chose to split $Traces(Sua)$ constructed in the previous step into subsets $ST_i \in ST$, one for each entry point of the system under analysis Sua. Later, every trace set ST_i shall give birth to one model, describing all possible behaviors starting from its corresponding entry point.

An *Autofunk* module performs these two steps, which are summarized in Algorithm 3. The first step starts by splitting $Traces(Sua)$ into several trace sets ST_i, one for each entry point of the system Sua, and then removes incomplete traces. Since we want a framework as flexible as possible, we first chose to perform a naive *statistical analysis* on $Traces(Sua)$ aiming at automatically detecting the entry and exit points. In Michelin systems, the parameter $point$ stores the product physical location and can be used to deduce the entry and exit points of the systems. This analysis is performed on the assignments $(point := val)$ found in the first and last valued events of the traces of $Traces(Sua)$ since $point$ captures the product physical location and especially the entry and exit points of Sua. We obtain two

ratios $Rinit((point := val1))$ and $Rfinal((point := val2))$. Based on these ratios, one can deduce the entry point set $POINT_{init}$ and the exit point set $POINT_{final}$ if $Traces(Sua)$ is large enough. Pragmatically, we observed that the traces collected during one or two days are not sufficient because they do not provide enough differences between the ratios. In this case, we assume that the number of entry and exit points, N and M, are given and we keep the first N and M ratios only. On the other hand, a week offers good results. We chose to set a fixed yet configurable minimum limit to 10%. Assignments $(point := val)$ having a ratio below this limit are not retained. Then, for each assignment $\alpha_i = (point := val1)$ in $POINT_{init}$, we construct a trace set ST_i such that a trace of ST_i has a first valued event including the assignment α_i, and ends with a valued event including an assignment $(point := val2)$ in $POINT_{final}$. We obtain the set $ST = \{ST_1, \ldots, ST_N\}$ with N the number of entry points of the system Sua.

Example 4.3.3 In our straightforward example, we obtain one trace set $ST_1 = Traces(Sua) \in ST$.

The second step consists in scanning the traces in ST to detect repetitive patterns $p \ldots p$. A *pattern* is a sequence of valued events that should contain at least one valued event. For example, $p_x = (a_x(p), \alpha_x)$ is a pattern, and $p_y = (a_y(p), \alpha_y)(a_y(p), \alpha_y)$ is another pattern, which is not equivalent to p_x. This notion of pattern is specific to our industrial context where physical devices may send information multiple times until they get an acknowledgment.

If *Autofunk* finds a trace t having a repetitive pattern p, and another *equivalent trace* t' including this pattern p once, then t is removed since we suppose that t does not express a new and interesting behavior. Traces are removed rather than deleting the repetitive patterns to prevent from modifying traces and to keep the *trace inclusion* [PY06] property between $CTraces(Sua)$ and $Traces(Sua)$. The $\sim_{(pid)}$ relation is used to define equivalence between two traces:

Definition 14 (The $\sim_{(pid)}$ relation) *Let t and t' be two traces. We denote by $\sim_{(pid)}$ the relation that defines the equivalence between t and t', and we write $t \sim_{(pid)} t'$, if and only if t and t' have the same successive valued events after having removed the assignments of the variable pid.*

In Algorithm 3, two traces $t = \sigma_i\ p \ldots p\ \sigma_j$ and $t' = \sigma'_i\ p'\ \sigma'_j$ are said *equivalent* if the patterns p, p', and the sub-traces $\sigma_i = (a_1(p), \alpha_1) \ldots (a_i(p), \alpha_i)$, $\sigma_j = (a_j(p), \alpha_j) \ldots (a_n(p), \alpha_n)$, σ'_i, σ'_j are equivalent, *i.e.* $p \sim_{(pid)} p'$, $\sigma_i \sim_{(pid)} \sigma'_i$, and $\sigma_j \sim_{(pid)} \sigma'_j$, and we can remove t from ST.

At the end of Algorithm 3 we obtain the complete trace set $CTraces(Sua)$, which is the union of all complete traces of each trace set ST_i.

Algorithm 3: Trace segmentation algorithm

Input : $Traces(Sua)$, optionally the number of entry points N and/or the number of exit points M

Output : $ST = \{ST_1, \ldots, ST_n\}, CTraces(Sua) = \bigcup_{1 \leq i \leq n} ST_i$

1 BEGIN;
2 *Step 1.* $Traces(Sua)$ *segmentation*
3 **foreach** $t = (a_1(p), \alpha_1) \ldots (a_n(p), \alpha_n) \in Traces(Sua)$ **do**
4 $\quad Rinit((point := val1) \in \alpha_1) + +$;
5 $\quad Rfinal((point := val2) \in \alpha_n) + +$;
6 $POINT_{init} = \{(point := val) \mid Rinit((point := val)) > 10\%$ or belongs to the N highest ratios$\}$;
7 $POINT_{final} = \{(point := val) \mid Rfinal((point := val)) > 10\%$ or belongs to the M highest ratios$\}$;
8 $ST = \emptyset$;
9 **for** $i = 1, \ldots, n$ **do**
10 $\quad$ **foreach** $\alpha_i = (point := val) \in POINT_{init}$ **do**
11 $\quad\quad ST_i = \{(a_1(p), \alpha_1) \ldots (a_n(p), \alpha_n) \in Traces(Sua) \mid \alpha_i \in \alpha_1, \exists\ (point := val2) \in \alpha_n \wedge (point := val2) \in POINT_{final}\}$;
12 $\quad\quad ST = ST \cup \{ST_i\}$;
13 *Step 2. Trace filtering*
14 **for** $i = 1, \ldots, n$ **do**
15 $\quad$ **foreach** $t = \sigma_i\ p \ldots p\ \sigma_j \in ST_i$ **do**
16 $\quad\quad$ **if** $\exists\ t' = \sigma'_i\ p'\ \sigma'_j \in ST_i \mid p \sim_{(pid)} p' \wedge \sigma_i \sim_{(pid)} \sigma'_i \wedge \sigma_j \sim_{(pid)} \sigma'_j$ **then**
17 $\quad\quad\quad ST_i = ST_i \setminus \{t\}$;
18 $CTraces(Sua) = \bigcup_{1 \leq i \leq n} ST_i$;
19 END;

Definition 15 (Complete trace) *Let* Sua *be a system under analysis and* $Traces(Sua)$ *be its trace set. A trace* $t = (a_1(p), \alpha_1) \ldots (a_n(p), \alpha_n) \in Traces(Sua)$ *is said* complete *if and only if* α_1 *includes an assignment* $(point := val1)$*, which denotes an entry point of* Sua*, and* α_n *includes an assignment* $(point := val2)$*, which denotes an exit point.*

The complete traces *of* Sua *are denoted by* $CTraces(Sua) \subseteq Traces(Sua)$.

4.3.3 STS generation

One model $\mathcal{S}_i$ is then built for each trace set $ST_i \in ST$ by reusing the technique introduced in Chapter 3 • Section 3.3.2 (page 57).

The translation of ST_i into a run set denoted by $Runs_i$ is done by completing traces with states. Each run starts by the same initial state $(l0, v_\emptyset)$ with $v_\emptyset$ the empty assignment. Then, new states are injected after each event. $Runs_i$ is formally given by the following definition:

Definition 16 (Structured run) *Let ST_i be a* complete trace set *obtained from Sua. We denote by $Runs_i$ the* set of runs *derived from ST_i with the following inference rule:*

$$\frac{\sigma_{k(1\leq k\leq n)}=(a_1(p),\alpha_1)...(a_n(p),\alpha_n) \in ST_i}{(l0,v_\emptyset)\cdot(a_1(p),\alpha_1)\cdot(l_{k1},v_\emptyset)\cdot\cdots\cdot(l_{kn-1},v_\emptyset)\cdot(a_n(p),\alpha_n)\cdot(l_{kn},v_\emptyset) \in Runs_i}$$

The above definition preserves trace inclusion [PY06] between $Runs_i$ and $Traces(Sua)$ because we only inject states between valued events, and we can deduce the following proposition:

Proposition 17 *Let ST_i be a trace set obtained from Sua. We have $Traces(Runs_i) \subseteq Traces(Sua)$.*

Runs are transformed into *STS paths* that are assembled together by means of a disjoint union. The resulting STS forms a tree compound of branches starting from the initial location $l0$, mimicking what we have done in the previous chapter but generalizing the definition to any run. Parameters and guards are extracted from the assignments found in valued events:

Definition 18 (Run set to STS) *Given a run set $Runs_i$, The STS $\mathcal{S}_i =< L_{\mathcal{S}_i}, l0_{\mathcal{S}_i}, V_{\mathcal{S}_i}, V0_{\mathcal{S}_i}, I_{\mathcal{S}_i}, \Lambda_{\mathcal{S}_i}, \rightarrow_{\mathcal{S}_i} >$ expresses the behaviors found in $Runs_i$ where:*

- $L_{\mathcal{S}_i} = \{l_i \mid \exists\, r \in Runs_i, (li, v_\emptyset)$ *is a state found in* $r\}$*;*
- $l0_{\mathcal{S}_i} = l0$ *is the initial location such that* $\forall r \in Runs_i$*,* r *starts with* $(l0, v_\emptyset)$*.* $l0$ *is a common location shared by all* $\mathcal{S}_i$*;*
- $V_{\mathcal{S}_i} = \emptyset$*;*
- $V0_{\mathcal{S}_i} = v_\emptyset$*;*
- $I_{\mathcal{S}_i}$ *is a finite set of parameters, disjoint from* $V_{\mathcal{S}_i}$*;*
- $\rightarrow_{\mathcal{S}_i}$ *and* $\Lambda_{\mathcal{S}_i}$ *are defined by the following inference rule applied to every element* $r \in Runs_i$*:*

$$\frac{(l_i,v_\emptyset)\cdot(a_i(p),\alpha_i)\cdot(l_{i+1},v_\emptyset)\in r,\ p=\{x|(x:=v)\in\alpha_i\},G_i=\bigwedge_{(x:=v)\in\alpha_i}(x == v)}{l_i \xrightarrow{a_i(p),G_i,id_{V_{\mathcal{S}_i}}}_{\mathcal{S}_i} l_{i+1}}$$

We obtain a model having a tree structure and whose traces are equivalent [PY06] to those of $CTraces(Sua)$ because each run is transformed into a unique STS path, only sharing a common initial location $l0$, and for each path, the guard corresponds to the conjunction of the assignments of the variables found in the trace. This is captured by the proposition below:

Proposition 19 *Let ST_i be a complete trace set obtained from Sua, and $Runs_i$ the set of runs derived from ST_i. We have $Traces(Runs_i) = CTraces(Sua) \subseteq Traces(Sua)$.*

At this point, production events are called *actions* of the STS.

Example 4.3.4 Figure 4.6 depicts the model obtained from the traces given in Figure 4.4. Every initial trace is now represented as a STS branch. Parameter assignments are modeled with constraints over transitions, called *guards*.

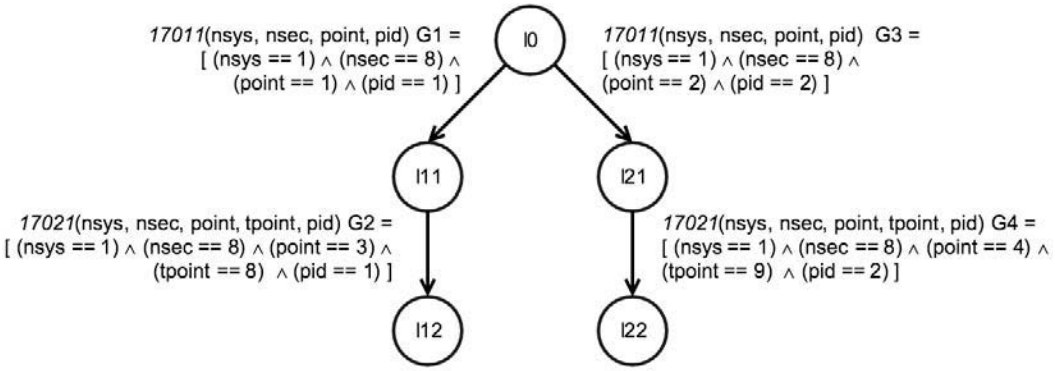

Fig. 4.6: First generated Symbolic Transition System model, based on the traces given in Figure 4.4.

This STS expresses the behaviors found in $Traces(Sua)$ but in a slightly different manner. Trace inclusion [PY06] between an inferred STS and $Traces(Sua)$, and trace equivalence [PY06] between an inferred STS and $CTraces(Sua)$ are captured by the following proposition:

Proposition 20 *Let Sua be a system under analysis and $Traces(Sua)$ be its trace set. $\mathcal{S}_i$ is an inferred STS from $Traces(Sua)$. We have $Traces(\mathcal{S}_i) = CTraces(Sua) \subseteq Traces(Sua)$.*

Definition 21 (STS set $\mathcal{S}$) *We denote by $\mathcal{S} = \{\mathcal{S}_1, \ldots, \mathcal{S}_n\}$ the set of all Symbolic Transition Systems $\mathcal{S}_i$ built from each trace set $ST_i \in ST$, and sharing the same common initial location $l0$.*

4.4 Improving generated models' usability

The size of the generated models is likely too large to be used in an efficient manner. That is why we added a reduction step, described in the next section. Then, we revisit the abstraction mechanism already introduced in the previous chapter.

4.4.1 STS reduction

A model $\mathcal{S}_i \in \mathcal{S}$ constructed with the steps presented before is usually too large, and thus cannot be beneficial as is. Using such a model for testing purpose would lead to too many test cases for instance. That is why we use a reduction step, aiming at diminishing the first model into a second one, denoted by $R(\mathcal{S}_i)$ that is more usable. As discussed in Chapter 3, most of the existing approaches propose two solutions: (i) inferring models with high levels of abstraction, which leads to over-approximated models, (ii) applying a minimization technique, which guarantees trace equivalence. Nonetheless, after having investigated this path in Chapter 3, we concluded that minimization is costly and highly time consuming on large models.

Given that a production system has a finite number of elements and that there should only be deterministic decisions, the STS $\mathcal{S}_i$ should contain branches capturing the same sequences of events (without necessarily the same parameter assignments). As a result, we chose to apply an approach tailored to support large models that consists in combining STS branches that have the same sequences of actions so that we still obtain a model having a tree structure. When branches are combined together, parameter assignments are wrapped into matrices in such a way that trace equivalence between the first model and the new one is preserved. More precisely, a sequence of successive guards found in a branch is stored into a matrix column. By doing this, we reduce the model size, we can still retrieve original behaviors (and only these ones, *i.e.* no approximation), and we still preserve trace inclusion between the reduced STS and $Traces(Sua)$. The use of matrices offers here another advantage: the parameter assignments are now packed into a structure that can be easily analyzed later. As shown later in Section 4.5, this straightforward approach gives good results in terms of STS reduction, and requires low processing time, even with millions of transitions.

Given a STS $\mathcal{S}_i$, every STS branch is adapted to express sequences of guards in a vector form to ease the STS reduction. Later, the concatenation of these vectors shall give birth to matrices. This adaptation is obtained with the definition of the *STS operator* Mat:

Definition 22 (The Mat operator) *Let* $\mathcal{S}_i =< L_{\mathcal{S}_i}, l0_{\mathcal{S}_i}, V_{\mathcal{S}_i}, V0_{\mathcal{S}_i}, I_{\mathcal{S}_i}, \Lambda_{\mathcal{S}_i}, \rightarrow_{\mathcal{S}_i} >$ *be a STS. We denote by* $Mat(\mathcal{S}_i)$ *the* STS operator *that consists in expressing guards of*

STS branches b_i in a vector form, such that $Mat(\mathcal{S}_i) =< L_{Mat(\mathcal{S}_i)}, l0_{Mat(\mathcal{S}_i)}, V_{Mat(\mathcal{S}_i)}, V0_{Mat(\mathcal{S}_i)}, I_{Mat(\mathcal{S}_i)}, \Lambda_{Mat(\mathcal{S}_i)}, \rightarrow_{Mat(\mathcal{S}_i)} >$ where:

- $L_{Mat(\mathcal{S}_i)} = L_{\mathcal{S}_i}$;
- $l0_{Mat(\mathcal{S}_i)} = l0_{\mathcal{S}_i}$;
- $I_{Mat(\mathcal{S}_i)} = I_{\mathcal{S}_i}$;
- $\Lambda_{Mat(\mathcal{S}_i)} = \Lambda_{\mathcal{S}_i}$;
- $V_{Mat(\mathcal{S}_i)}$, $V0_{Mat(\mathcal{S}_i)}$, *and* $\rightarrow_{Mat(\mathcal{S}_i)}$ *are given by the following rule:*

$$\frac{b_i = l0 \xRightarrow{(a_1(p_1),G_1,A_1)...(a_n(p_n),G_n,A_n)} l_n}{V0_{Mat(\mathcal{S}_i)} := V0_{Mat(\mathcal{S}_i)} \wedge M_i = \begin{bmatrix} G_1 \\ \vdots \\ G_n \end{bmatrix}} \quad l0_{Mat(\mathcal{S}_i)} \xRightarrow{(a_1(p_1),M_i[1],id_V)...(a_n(p_n),M_i[n],id_V)}_{Mat(\mathcal{S}_i)} l_n$$

Given a branch $b_i \in (\rightarrow_{Mat(\mathcal{S}_i)})^n$, we also denote by $Mat(b_i) = M_i$ the vector used with b_i.

It is now possible to merge the STS branches that have the same sequences of actions. This last sentence can be interpreted as an equivalence relation over STS branches from which we can derive *equivalence classes*:

Definition 23 (STS branch equivalence class) *Let $\mathcal{S}_i =< L_{\mathcal{S}_i}, l0_{\mathcal{S}_i}, V_{\mathcal{S}_i}, V0_{\mathcal{S}_i}, I_{\mathcal{S}_i}, \Lambda_{\mathcal{S}_i}, \rightarrow_{\mathcal{S}_i} >$ be a STS obtained from $Traces(Sua)$ (and having a tree structure).*

$[b]$ denotes the equivalence class *of $\mathcal{S}_i$ branches such that:* $[b] = \{b_j = l0_{\mathcal{S}_i} \xRightarrow{(a_1(p_1),G_{1j},A_{1j})...(a_n(p_n),G_{nj},A_{nj})} l_{nj} (1 \leq j) \mid b = l0_{\mathcal{S}_i} \xRightarrow{(a_1(p_1),G_1,A_1)...(a_n(p_n),G_n,A_n)} l_n\}$.

One of the main advantages of such a relation is that its computation can be done very quickly with an algorithm based on hash functions (cf. Section 4.5.1).

The reduced STS denoted by $R(\mathcal{S}_i)$ of $\mathcal{S}_i$ is obtained by concatenating all the branches of each equivalence class $[b]$ found in $Mat(\mathcal{S}_i)$ into one branch. The vectors found in the branches of $[b]$ are concatenated as well into the same unique matrix $M_{[b]}$. A column of this matrix represents a complete and ordered sequence of guards found in one initial branch of $\mathcal{S}_i$. $R(\mathcal{S}_i)$ is defined as follows:

Definition 24 (Reduced STS $R(\mathcal{S}_i)$) *Let $\mathcal{S}_i =< L_{\mathcal{S}_i}, l0_{\mathcal{S}_i}, V_{\mathcal{S}_i}, V0_{\mathcal{S}_i}, I_{\mathcal{S}_i}, \Lambda_{\mathcal{S}_i}, \rightarrow_{\mathcal{S}_i} >$ be a STS inferred from a structured trace set $Traces(Sua)$. The reduction of $\mathcal{S}_i$ is modeled by the STS $R(\mathcal{S}_i) =< L_R, l0_R, V_R, V0_R, I_R, \Lambda_R, \rightarrow_R >$ where:*

$$\frac{[b] = \{b_1, \ldots, b_m\}, b = l0_{\mathcal{S}_i} \xrightarrow{(a_1(p_1),G_1,A_1)\ldots(a_n(p_n),G_n,A_n)}_{Mat(\mathcal{S}_i)} l_n}{\begin{array}{c} V0_R := V0_R \wedge M_{[b]} = [Mat(b_1), \ldots, Mat(b_m)] \wedge (1 \leq c_{[b]} \leq m), \\ l0_R \xrightarrow{(a_1(p_1),M_{[b]}[1,c_{[b]}],id_V)\ldots(a_n(p_n),M_{[b]}[n,c_{[b]}],id_V)}_R (l_{n1} \cdots l_{nm}) \end{array}}$$

The resulting model $R(\mathcal{S}_i)$ is a STS composed of variables assigned to matrices whose values are used as guards. A matrix column represents a successive list of guards found in a branch of the initial STS $\mathcal{S}_i$. The choice of the column in a matrix depends on a new variable $c_{[b]}$.

Example 4.4.1 Figure 4.7 depicts the reduced model obtained from the STS depicted in Figure 4.6. Its guards are placed into two vectors $M_1 = \begin{bmatrix} G1 \\ G2 \end{bmatrix}$ and $M_2 = \begin{bmatrix} G3 \\ G4 \end{bmatrix}$, combined into the same matrix $M_{[b]}$. The variable $c_{[b]}$ is used to take either the guards of the first column or the guards of the second one.

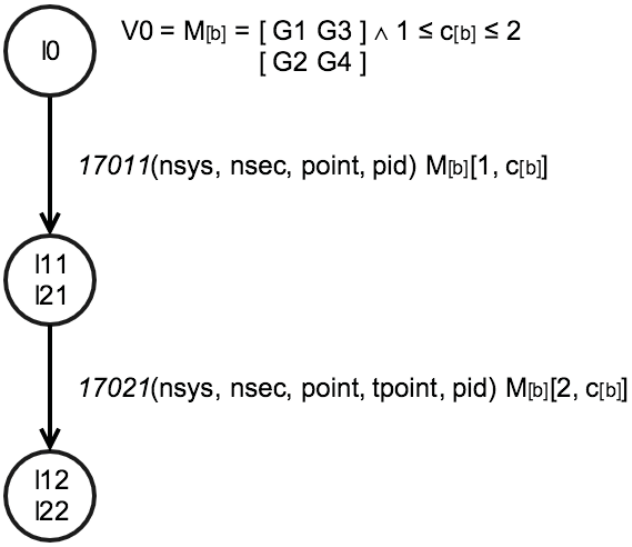

Fig. 4.7: Reduced Symbolic Transition System model obtained from the model depicted in Figure 4.6.

The STS $R(\mathcal{S}_i)$ has less branches but still expresses the initial behaviors described by the STS $\mathcal{S}_i$. $R(\mathcal{S}_i)$ and $\mathcal{S}_i$ are said trace equivalent [PY06]. This is captured by the following proposition:

Proposition 25 *Let Sua be a system under analysis and $Traces(Sua)$ be its traces set. $R(\mathcal{S}_i)$ is a STS derived from $Traces(Sua)$. We have $Traces(R(\mathcal{S}_i)) = Traces(\mathcal{S}_i) \subseteq CTraces(Sua) \subseteq Traces(Sua)$.*

Definition 26 (STS set $R(\mathcal{S})$) *We denote by $R(\mathcal{S}) = \{R(\mathcal{S}_1), \ldots, R(\mathcal{S}_n)\}$ the set of all reduced Symbolic Transition Systems $R(\mathcal{S}_i)$.*

4.4.2 STS abstraction

Given the trace set $ST_i \in ST$, the generated STS $R(\mathcal{S}_i)$ can be used for analysis purpose, but it is still difficult to manually interpret, even for experts. This fifth *Autofunk* module aims to analyze $R(\mathcal{S}_i)$ in order to produce a new STS $\mathcal{S}_i^{\uparrow}$ whose level of abstraction is lifted by using more intelligible actions. This process is performed with inference rules, which encode the knowledge of the expert of the system. These rules are fired on the transitions of $R(\mathcal{S}_i)$ to deduce new transitions. We consider the same two types of rules as in Chapter 3 • Section 3.3.3 (page 62):

- The rules *replacing some transitions* by more comprehensive ones. These rules are of the form: *When Transition* $l_1 \xrightarrow{a(p),G,A}_{R(\mathcal{S}_i)} l_2$*, condition on* $a(p), G, A$*, Then add* $l_1 \xrightarrow{a'(p'),G',A'}_{\mathcal{S}_i^{\uparrow}} l_2$ *and retract* $l_1 \xrightarrow{a(p),G,A}_{R(\mathcal{S}_i)} l_2$;
- The rules that *aggregate some successive transitions* to a single transition compound of a more abstract action. These rules are of the form *When Transition* $l_1 \xRightarrow{(a_1,G_1,A_1)\ldots(a_n,G_n,A_n)} l_n$*, condition on* $(a_1, G_1, A_1) \ldots (a_n, G_n, A_n)$*, Then add* $l_1 \xrightarrow{a(p),G,A}_{\mathcal{S}_i^{\uparrow}} l_n$*, and retract* $l_1 \xRightarrow{(a_1,G_1,A_1)\ldots(a_n,G_n,A_n)} l_n$.

The generated STSs represent recorded scenarios modeled at a higher level of abstraction. These can be particularly useful for generating documentation or better understanding how the system behaves, especially when issues are experienced in production. On the other hand, it is manifest that the trace inclusion property is lost with the STSs constructed by this module because sequences are modified. In other words, such models cannot be used for testing purpose.

Example 4.4.2 If we take back our example, the actions of the STS depicted in Figure 4.7 can be replaced thanks to the inference rules given in Figure 4.8. Such rules change the labels 17011 and 17021 to more intelligible ones. The STS depicted on the left in Figure 4.10 is the result of the application of these rules on the model given in Figure 4.7.

The rule shown in Figure 4.9 aggregates two transitions into a unique transition indicating the movement of a product in its production line. $Transition$ are facts

modeling STS transitions as seen in Chapter 3. We need a variable $\$lfinal$ that retains the final location of the first transition $\$t1$, so that we can reuse it in the condition of the second transition $\$t2$. The final location of the first transition must be the initial location of the second transition ($Linit == \$lfinal$) to perform the aggregation. Because our STSs have a tree form, there is no need to ensure the absence of more than one transition starting from the final location of $\$t1$ ($\$lfinal$).

From 5 initial production events that are not self-explanatory, we generate a simpler STS constituted of one transition, clearly expressing a part of the functioning of the system. The result of this rule is shown in Figure 4.10 (on the right).

```
rule "Mark destination requests"
when:
  $t: Transition(action == "17011")
then
  $t.setAction("DestinationRequest")
end
```

```
rule "Mark destination responses"
when:
  $t: Transition(action == "17021")
then
  $t.setAction("DestinationResponse")
end
```

Fig. 4.8: Two rules adding value to existing transitions by replacing their actions by more intelligible ones. These names are part of the domain language used by Michelin experts.

4.5 Implementation and experimentation

In this section, we briefly describe the implementation of our model inference framework for Michelin. Then, we give an evaluation on real production systems.

4.5.1 Implementation

Our framework *Autofunk v2* is also developed in Java and still leverages Drools. In our industrial context, we have several bases of facts used throughout the different *Autofunk* modules, represented as Java objects such as: Events, Trace sets ST_i, Runs, Transitions, and STSs. We chose to target performance while implementing *Autofunk*. That is why most of the steps are implemented with parallel algorithms (except the

```
rule "Aggregate destination requests/responses"
when
  $t1: Transition(
    action == "DestinationRequest", $lfinal := Lfinal
  )
  $t2: Transition(
    action == "DestinationResponse" , Linit == $lfinal
  )
then
  insert(
    new Transition(
      "ProductAdvance",
      Guard($t1.Guard, $t2.Guard),
      Assign($t1.Assign, $t2.Assign),
      $t1.Linit,
      $t2.Lfinal
    )
  )
  retract($t1)
  retract($t2)
end
```

Fig. 4.9: An inference rule that aggregates two transitions of a Symbolic Transition System into a single transition. An example of its application is given in Figure 4.10.

production event parsing) combined with Java 8 parallel streams[1] and processing mechanisms. Figure 4.11 shows the initial setup of *Autofunk*. We have sets of log files to parse coming from Michelin's logging system. *Autofunk* reads these files, and transforms them into traces, that are then used to build models.

The input trace collection is constructed with a classical parser, which returns $Event$ Java objects. By now, we are not able to parallelize this part because of an issue we faced with Michelin's logging system. The resulting drawback is that the time to parse traces is longer than expected and heavily depends on the size of data to parse. The $Event$ base is then filtered with Drools inference rules as presented in Section 4.3.1. Then, we call a straightforward algorithm for reconstructing traces: it iterates over the $Event$ base and creates a set for each assignment of the identifier pid. These sets are sorted to construct traces given as $Trace$ Java objects. These objects correspond to $Traces(Sua)$. The generation of the trace subsets $ST = \{ST_1, \ldots, ST_N\}$ and of the first STSs are done with Drools inference rules applied in parallel with one thread per trace set.

The STS reduction, and specifically the generation of STS branches equivalence classes, has been implemented with a specific algorithm for better performance.

[1] https://docs.oracle.com/javase/tutorial/collections/streams/parallelism.html

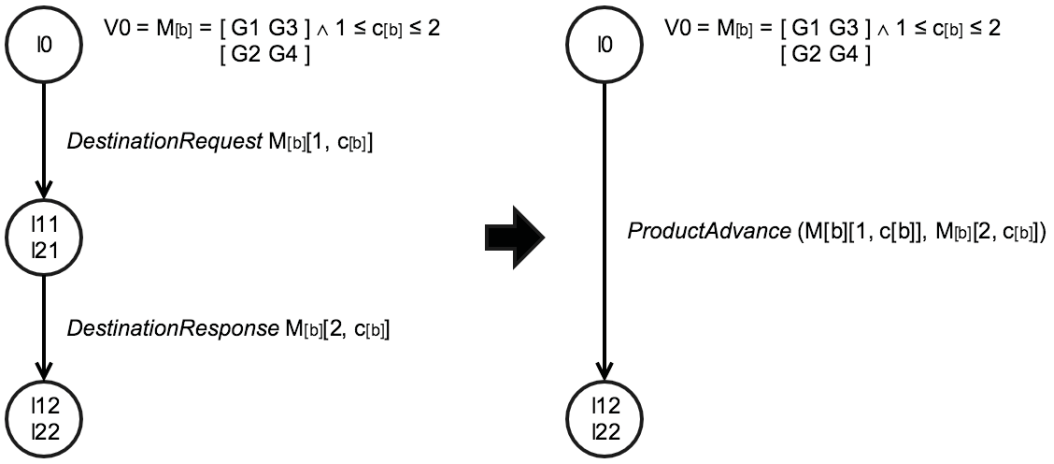

Fig. 4.10: The construction of the final Symbolic Transition System model. The final model is on the right, obtained after having applied all the abstraction rules given in Figures 4.8 and 4.9.

Indeed, comparing every action in STS branches in order to aggregate them is time consuming. Given a STS $\mathcal{S}$, this algorithm generates a signature for each branch b, *i.e.* a hash (SHA1[2] algorithm) of the concatenation of the signatures of the actions of b. The branches which have the same signature are gathered together and establish branch equivalence classes (as described in Section 4.4.1). Implementing equivalence classes using hashes allows to quickly reduce STS of any size, especially because this technique scales well. Thereafter, the reduced $R(\mathcal{S})$ is constructed thanks to the inference rule given in Section 4.4.1.

At the time of writing, we are working on the architecture depicted in Figure 4.12. It shows how we plan to integrate *Autofunk* with any Michelin production system. Events are collected on-the-fly from a system under analysis, and sent to a database (Elasticsearch[3] here) thanks to a message broker (namely RabbitMQ[4]). Initial benchmarks revealed that such an infrastructure allows to collect up to 10,000 events per second with a negligible overhead on the system. *Autofunk* then queries the database to fetch product events, and finally build models. This approach also solves the issue mentioned previously. Furthermore, it allows to leverage the data for other purposes, *e.g.*, visualization or more extensive analyses.

4.5.2 Evaluation

We conducted several experiments with real sets of production events, recorded in one of Michelin's factories at different periods of time. We executed our implemen-

[2] Secure Hash Algorithm 1
[3] `https://www.elastic.co/products/elasticsearch`
[4] `https://www.rabbitmq.com/`

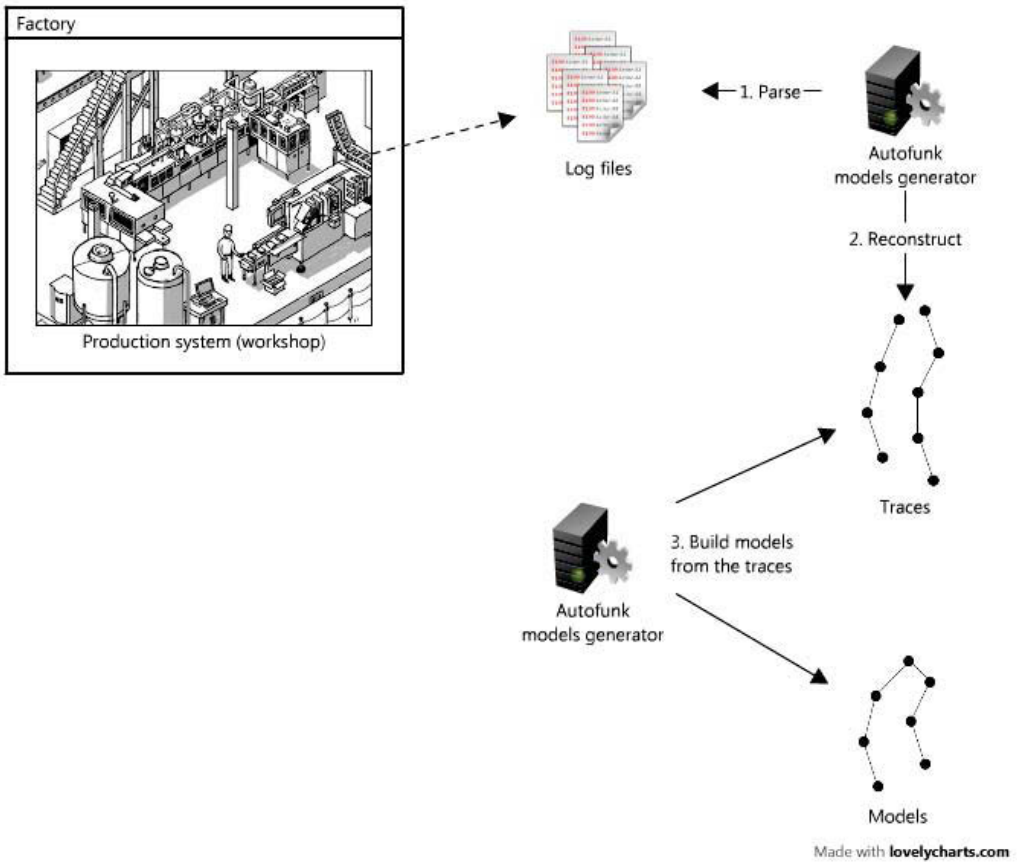

Fig. 4.11: The architecture of *Autofunk* that has been used to conduct our first experiments with Michelin log files. A newer (and better) architecture is given in Figure 4.12.

tation of *Autofunk v2* on a Linux (Debian) machine with 12 Intel(R) Xeon(R) CPU X5660 @ 2.8GHz and 64GB RAM.

We present here the results of 6 experiments on the same production system with different event sets collected during 1, 8, 11, 20, and 23 days. These results are depicted in Table 4.1. The third column gives the number of production events recorded on the system. The next column shows the trace number obtained after the parsing step. N and M represent the entry and exit points automatically computed with the statistical analysis. The column Trace Subsets shows how $Traces(Sua)$ is segmented into subsets $\{ST_1, \ldots, ST_N\}$ and the number of traces included in each subset. These numbers of traces also correspond to the numbers of branches generated in the STSs $\mathcal{S}_1, \ldots, \mathcal{S}_N$. The eighth column, $\#\ R(\mathcal{S}_i)$, represents the number of branches found in each reduced STSs $R(\mathcal{S}_1), \ldots, R(\mathcal{S}_N)$. Finally, execution times are rounded and expressed in minutes in the last column.

First, these results show that our framework can take millions of production events and still builds models quickly. With sets collected during one day up to one week (experiments A, B, C, and D), models are inferred in less than 10 minutes.

Experiment	Number of days	Number of events	$Card(Traces(Sua))$	N	M	# Trace subsets	# $R(\mathcal{S}_i)$	Exec. time (min)
A_1	1	660,431	16,602	2	3	4,822	332	1
A_2						1,310	193	
B_1	8	3,952,906	66,880	3	3	28,555	914	9
B_2						18,900	788	
B_3						6,681	51	
C_1	11	3,615,215	61,125	3	3	28,302	889	9
C_2						14,605	681	
C_2						7,824	80	
D_1	11	3,851,264	73,364	2	3	35,541	924	9
D_2						17,402	837	
E_1	20	7,635,494	134,908	2	3	61,795	1,441	16
E_2						35,799	1,401	
F_1	23	9,231,160	161,035	2	3	77,058	1,587	24
F_2						43,536	1,585	

Tab. 4.1: This table shows the results of 6 experiments on a Michelin production system with different event sets.

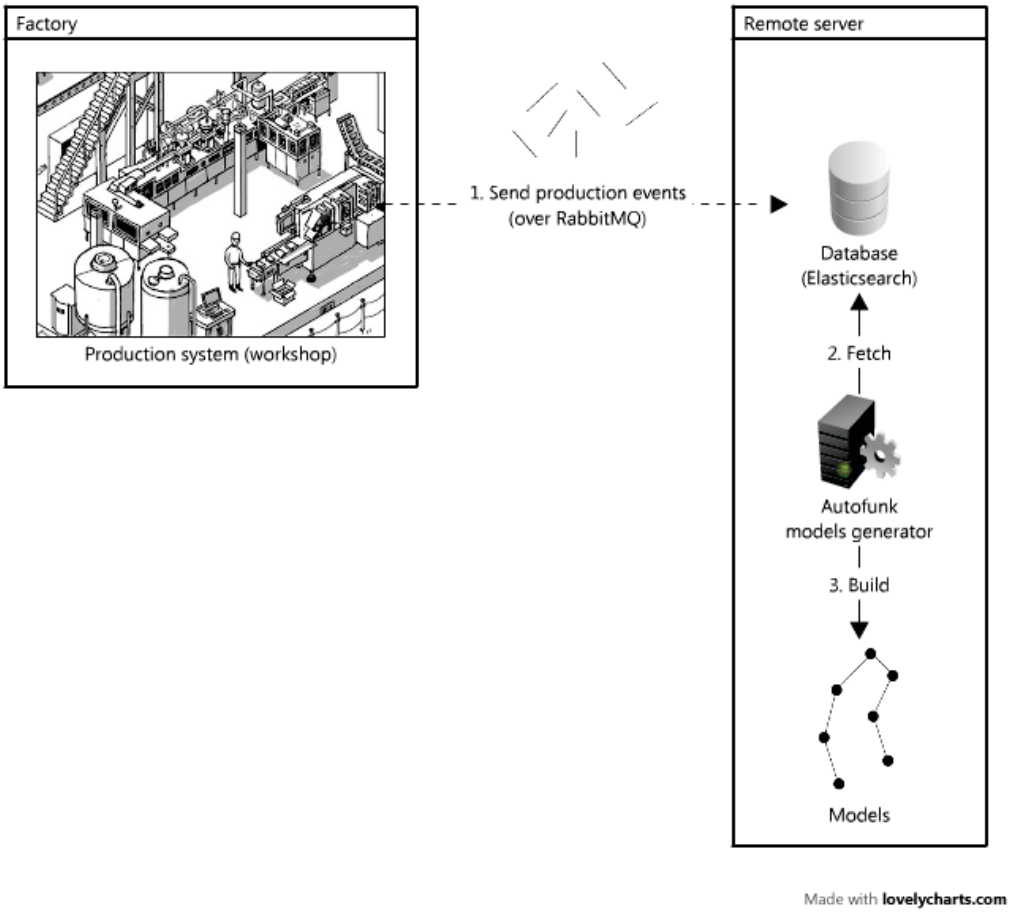

Fig. 4.12: The overall architecture built at Michelin to use *Autofunk* in a production environment. Events are sent over RabbitMQ to another server in order to minimize the overhead. Collected events are stored in a database. *Autofunk* can then fetch these events, and build the models.

Therefore, *Autofunk* can be used to quickly infer models for analysis purpose or to help diagnose faults in a system. Experiment F handled almost 10 million events in less than half an hour to build two models including around 1,600 branches. As mentioned earlier in this section, the parsing process is not parallelized yet, and it took up to 20 minutes to open and parse around 1,000 files (number of Michelin log files for this experiment). This issue should be solved by the architecture presented in Figure 4.12. The graph shown in Figure 4.13 summarizes the performances of our framework, and how fast it is at transforming production events into models (experiments B, C and D run in about 9 minutes). It also demonstrates that doubling the event set does not involve doubling its execution time. The linear regression reveals that the overall framework scales well, even with the current parsing implementation.

In Table 4.1, the difference between the number of trace subsets (7th column) and the number of branches included in the STSs $R(\mathcal{S}_i)$ (8th column) clearly shows that

our STS reduction approach is effective. For instance, with experiment B, we reduce the STSs by 91.88% against the initial trace set $Traces(Sua)$. In other words, 91% of the original behaviors are packed into matrices. As a reminder, such results are tied to the specific context of Michelin production systems. The reduction ratio may vary with other systems.

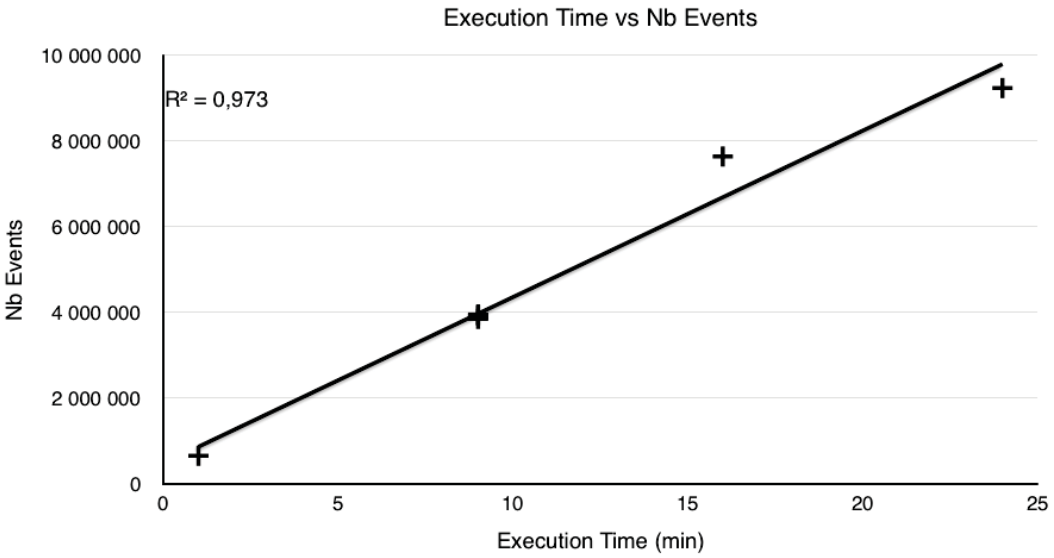

Fig. 4.13: Execution time vs events. According to the trend shown by the linear regression, we can state that our framework scales well.

We also extracted the values of columns 4 and 7 in Table 4.1 to depict the stacked bar chart illustrated in Figure 4.14. This chart shows, for each experiment, the proportion of complete traces kept by *Autofunk* to build models, over the initial number of traces in $Traces(Sua)$. *Autofunk* has kept only 37% of the initial traces in Experiment A because its initial trace set is too small and contains many incomplete behaviors. During a day, most of the recorded traces do not start or end at entry or exit points, but rather start or end somewhere in production lines (cf. Section 4.2). That is why, on a single day, we can find so many incomplete traces. With more production events, such a phenomenon is limited because we absorb these storage delays.

We can also notice that experiments C and D have similar initial trace sets but experiment C owns more complete traces than experiment D by 12%, which is significant. Furthermore, experiments B and C take 3 entry points into account while the others only take 2 of them. This is related to the fixed limit of 10% we chose to ensure truly entry points to be automatically selected. The workshop we analyzed has three entry points, two of which are mainly used. The third entry point is employed to balance the production load between this workshop and a second one located close to it in the same factory. Depending on the period, this entry point may be more or less solicited, hence the difference between experiments B, C and experiment D. Increasing the limit of 10% to a higher value would change the value

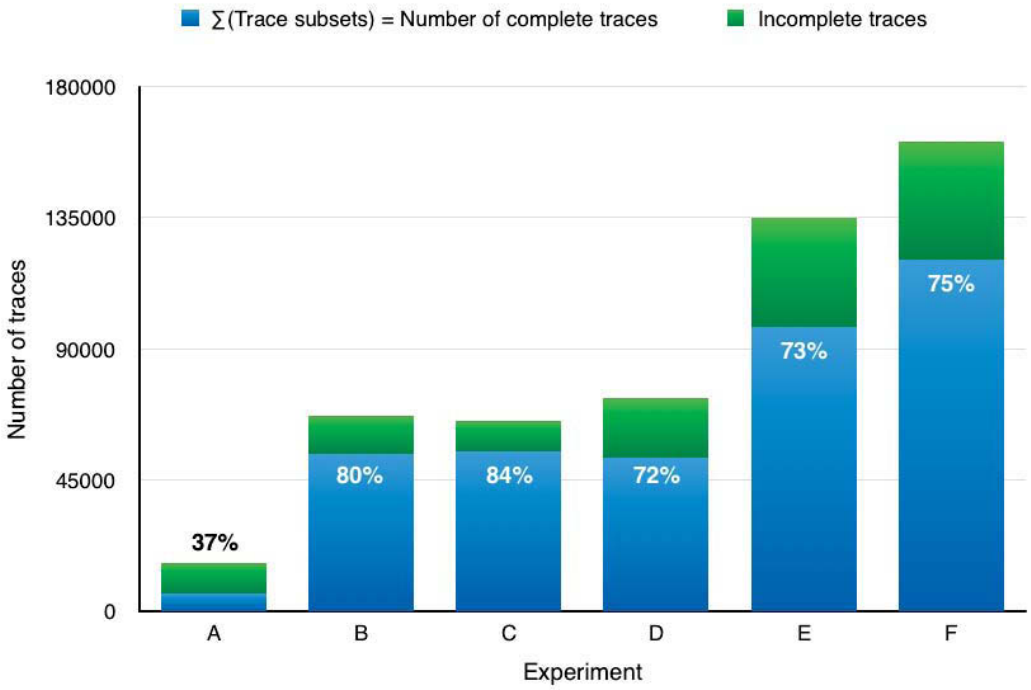

Fig. 4.14: Proportions of complete traces for the different experiments. This chart shows that *Autofunk* considers a lot of traces to infer models, but there is still room for improvement.

of N for experiments B and C, but would also impact experiment A by introducing false results since incorrect entry points could be selected. By means of a manual analysis, we concluded that 10% was the best ratio for removing incomplete traces in our experiments. 30% of initial traces have been removed, which is close to the reality.

Another potential issue with our parsing implementation is that every event has to be loaded in memory, so that we can perform computation and apply our algorithms on them. But working with millions of Java objects requires a lot of memory, *i.e.* memory consumption depends on the amount of initial traces. We compared execution time and memory consumption in Figure 4.15, showing that memory consumption tends to follow a logarithmic trend. In the next version of *Autofunk*, we plan to work on improving memory consumption even if it has been considered acceptable as is by Michelin.

4.6 A better solution to the trace segmentation and filtering problem with machine learning

In Section 4.3.2, we presented a naive statistical analysis to segment and filter the initial trace set used to infer models, part of *Autofunk v2*. Nevertheless, experiments

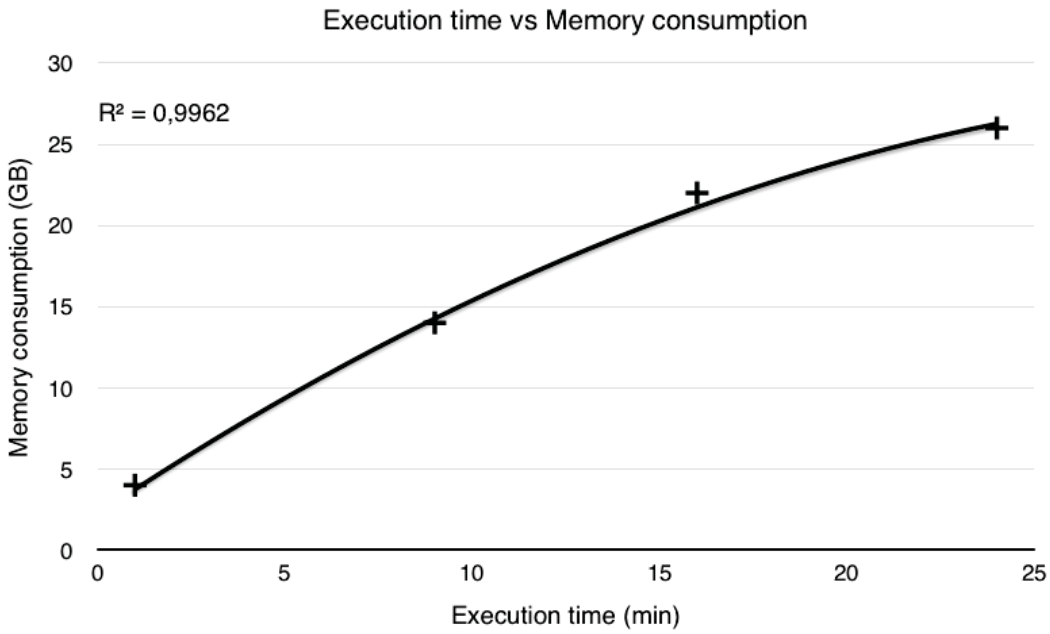

Fig. 4.15: Execution time vs memory consumption. This version 2 of *Autofunk* is still a prototype, and memory consumption remains an issue.

(as seen in the previous section) revealed that this empirical method was not stable, in other words the use of a configurable minimum limit (manually given), which is neither smart nor accurate. That is why we chose to work on a new solution to segment and filter traces, which gave birth to *Autofunk v3*.

Autofunk v3 now relies on a *machine learning* technique to segment a trace set into several subsets, one per entry point of the system Sua. We leverage this process to also remove incomplete traces, *i.e.* traces that do not express an execution starting from an entry point and ending to an exit point. These can be extracted by analyzing the traces and the variable $point$, which captures the product physical location.

In order to determine both entry and exit points of Sua, we rely on an outlier detection approach [HA04]. An outlier is an observation that deviates so much from the other observations as to arouse suspicions that it was generated by a different mechanism. More precisely, we chose to use the *k-means clustering* method [JAH79], a machine learning algorithm, which is both fast and efficient, and does not need to be trained before being effectively used (that is called unsupervised learning, and it is well-known in the machine learning field). *k-means clustering* aims to partition n observations into k clusters as shown in Figure 4.16.

In our context, observations are represented by the variable $point$ present in each trace of $Traces(Sua)$, which captures the product physical location, and $k = 2$ as we want to group the outliers together, and leave the other points in another cluster. Once the entry and exit points are found, we segment $Traces(Sua)$ and

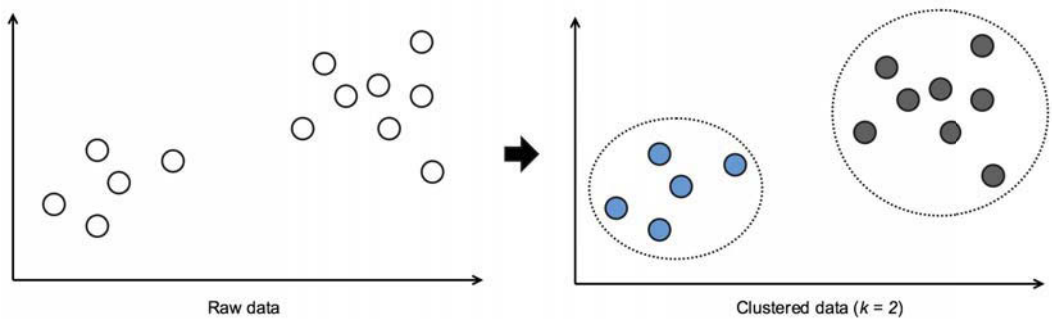

Fig. 4.16: k-means clustering explained: the intuition is to partition n observations into k clusters in which each observation belongs to the cluster with the nearest mean.

obtain a set $ST = \{ST_1, \ldots, ST_n\}$, whose union forms the set of complete traces $CTraces(Sua)$. Then, we apply the same generation and reduction steps as described in Section 4.3.3 and Section 4.4.1 so that we obtain the set of reduced models $R(\mathcal{S}) = \{R(\mathcal{S}_1), \ldots, R(\mathcal{S}_n)\}$.

In our implementation, we chose to rely on the MLlib[5] machine learning library from Apache Spark[6], a fast and general Java engine for large-scale data processing. The MLlib library provides an efficient Java implementation of k-means clustering.

4.7 Conclusion

In this chapter, we presented our revisited framework *Autofunk*, combining model inference, machine learning, and expert systems to generate models from production systems. Figure 4.17 shows the final architecture of our *third* version of *Autofunk* along with the technologies used in each module. Given a large set of production events, our framework infers exact models whose traces are included in the initial trace set of a system under analysis [PY06]. We chose to design *Autofunk* for targeting high performance. Our evaluation shows that this approach is suitable in the context of production systems since we quickly obtain STS trees reduced by 90% against the original trace sets of the system under analysis.

While there is still room for improvement on how we generate exact models, we decided to go deeper and use such models for testing purpose. In the next chapter, we describe our testing technique that leverages the work presented in this chapter.

[5] `https://spark.apache.org/mllib/`
[6] `https://spark.apache.org/`

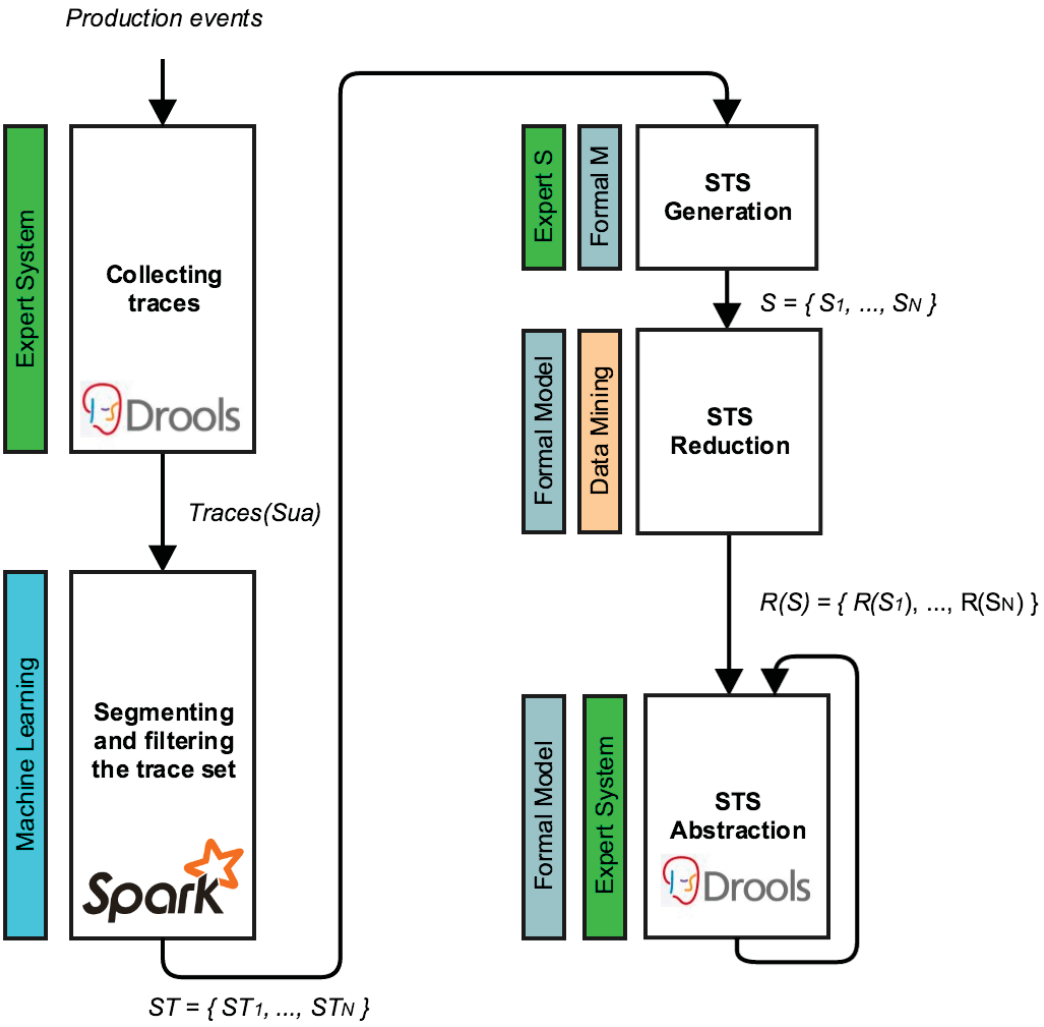

Fig. 4.17: Final design of our framework *Autofunk* (v3), designed for quickly inferring models of Michelin's production systems.

5 Testing applied to production systems

In this chapter, we tackle the problem of testing production systems, without disturbing them, and without having any specification *a priori*. Those two constraints sounds familiar as they have already been taken into consideration in Chapter 4. We present a passive testing technique built on-top of *Autofunk v3*. First, we infer reference models of a system under analysis Sua using the technique presented previously. Then, we check whether a system under test Sut conforms to these inferred models by means of two implementation relations. We use a slightly modified version of the trace preorder relation to strictly check conformance between the two systems. Because the inferred models are partial, they likely lack information, which implies that our first strict relation may be too "strong". That is why we propose a second (weaker) implementation relation to comply with Michelin's needs. Such a relation is less strict than the first one in order to accept non-standard behaviors down to a certain point. These two relations are leveraged in an algorithm for performing offline passive conformance testing. We end this chapter with a few results on this testing technique.

Contents

5.1 Introduction

Manual testing is, by far, the most popular technique for testing, but this technique is known to be error-prone as well. As already formulated in this thesis, production systems are usually composed of thousands of states (*i.e.* sets of conditions that exist at a given instant in time) and production events, which makes testing time consuming. In this context, we propose a *passive testing framework* for production systems that is compound of: (i) a model inference engine, already presented in Chapter 4, and (ii) a passive test engine, which is the purpose of this chapter. Both parts have to be fast and scalable to be used in practice.

The main idea of our proposal is that, given a running production system, for which active testing cannot be applied, we extract knowledge and models. Such models describe the functional behaviors of the system, and may serve different purposes, *e.g.*, testing another production system. The latter can be a new system roughly comparable to the first one in terms of features, but it can also be an updated version of the first one. Indeed, upgrades might inadvertently introduce faults, and it could lead to severe damages.

In our context, testing the updated system means detecting potential regressions before deploying changes in production. This is usually performed by engineers at Michelin, yet their "testing" phase is actually a manual task performed in a simulation room where they check a couple of known scenarios for a long period (about 6 months). Most of the time, such a process works well, but it takes a lot of time, and there is no guarantee regarding the number of covered scenarios. In addition, engineers who write or maintain the applications are likely the same who perform these manual testing phases, which can be problematic because they often know too well the applications. We designed our testing framework to help Michelin engineers focus on possible failures, automatically highlighted by the framework in a short amount of time. At the end, engineers still have to perform a few manual tasks, but they are more efficient as they have only a few behaviors to check (instead of all possible behaviors), given in a reliable and fast manner.

Generally speaking, a *passive tester* (also known as observer) aims at checking whether a system under test *conforms to* a model. It can be performed in either online or offline mode, as defined below:

- **Online testing:** it means that traces are computed and analyzed on-the-fly to provide verdicts, and no trace set is studied *a posteriori*;
- **Offline testing:** it means that a set of traces has been collected while the system is running. Then, the tester gives verdicts afterwards.

In this Chapter, we present an offline passive testing technique. We collect the traces of the system under test by reusing *Autofunk v3*'s Models generator, and we build a set of traces whose level of abstraction is the same as those considered for inferring models. Then, we use these traces to check whether the system under test conforms to the inferred models. In previous works, we used to work with fixed sets of traces. We noticed that, by taking large trace sets, we could build more complete models, and performing offline passive testing allows to use such large trace sets.

Conformance is defined with two implementation relations, which express precisely what the system under test should do. The first relation is based on the *trace preorder* [DNH84], which is a well-known relation based upon trace inclusion, and heavily used with passive testing. Nevertheless, our inferred models are partials, *i.e.* they do not necessarily capture all the possible behaviors that should happen. That is why we propose a second implementation relation, less restrictive on the traces that should be observed from the system under test.

In Section 5.2, we introduce an extra step of the model inference method described in Chapter 4 that is required to enable testing. In Section 5.3, we present our offline passive testing technique, built on-top of this model inference framework. We present key results on offline passive testing in Section 5.4. Finally, we conclude on this chapter in Section 5.5.

Publication. This work has been partially published in the Proceedings of the 13th International Conference on Formal Methods and Models for Co-Design (MEMOCODE'15) [DS15b].

5.2 Normalization of the inferred models

In order to perform testing, we reuse the reduced model set $R(\mathcal{S}) = \{R(\mathcal{S}_1), \ldots, R(\mathcal{S}_n)\}$ inferred with *Autofunk* that we *normalize* to get rid of some runtime-dependent information of the system under analysis Sua. Indeed, both models $\mathcal{S}_i$ and $R(\mathcal{S}_i)$ include parameters that are dependent to the products being manufactured. That is a consequence of generating models that describe behaviors of a continuous stream of products that are strictly identified. For instance, each action in a given sequence owns the assignment $(pid := val)$ (for the record, pid stands for *product identifier*). Collected traces are also time-dependent, which would make testing of another production system unfeasible. These information are typically known by a human domain expert, and we can use inference rules to identify and remove assignments one more time.

Given the model sets $\mathcal{S} = \{\mathcal{S}_1, \ldots, \mathcal{S}_n\}$ and $R(\mathcal{S}) = \{R(\mathcal{S}_1), \ldots, R(\mathcal{S}_n)\}$, we remove the assignments related to product identifiers and time stamps to produce normalized model sets, denoted by $\mathcal{S}^N = \{\mathcal{S}_1^N, \ldots, \mathcal{S}_n^N\}$ and $R(\mathcal{S}^N) = \{R(\mathcal{S}_1^N), \ldots, R(\mathcal{S}_n^N)\}$.

Furthermore, we label all the final locations with "Pass". We flag these locations as *verdict locations*, and gather them in the set $Pass \subseteq L_{\mathcal{S}_i^N}$. Both $\mathcal{S}_i^N$ and $R(\mathcal{S}_i^N)$ represent more generic models, *i.e.* they express *some possible complete behaviors that should happen*. These behaviors are represented by the traces $Traces_{Pass}(\mathcal{S}^N) = \bigcup_{1 \leq i \leq n} Traces_{Pass}(\mathcal{S}_i^N) = Traces_{Pass}(R(\mathcal{S}^N))$. We refer to these traces as *pass traces*, and we call the other traces *possibly fail traces*.

5.3 Passive testing with *Autofunk*

We consider both model sets $\mathcal{S}^N$ and $R(\mathcal{S}^N)$ of a system under analysis Sua, generated by our inference-based model generation framework, as *reference models*. In this section, we present the second part of our testing framework, dedicated to the passive testing of a system under test Sut.

Figure 5.1 depicts the design of our offline passive testing technique. A set of production events has been collected beforehand from Sut in the same way as for Sua. These events are grouped into traces to form the trace set $Traces(Sut)$, and then filtered to obtain a set of complete traces denoted by $CTraces(Sut)$ (cf. Chapter 4 • Section 4.6 (page 104)). Finally, we perform passive testing to *check* if Sut conforms to $\mathcal{S}^N$.

Our industrial partner wishes to check whether every complete execution trace of Sut matches a behavior captured by $\mathcal{S}^N$. In this case, the test verdict must reflect a successful result. On the contrary, if a complete execution of Sut is not captured by $\mathcal{S}^N$, one cannot conclude that Sut is faulty because $\mathcal{S}^N$ is a partial model, and it does not necessarily includes all the correct behaviors. Below, we formalize these verdict notions with two implementation relations. Such relations between models can only be written by assuming the following *test assumption*: the black-box system Sut can be described by a model, here with a Labeled Transition System as defined in Chapter 2.1 • Section 2.1.2 (page 17). For simplicity purpose, we also denote this model by Sut.

It is manifest that both the models and Sut must be *compatible*, *i.e.* the production events captured on Sua and Sut must lead to similar valued events. Otherwise, all traces of Sut will likely be rejected, and the testing process will become wasteful:

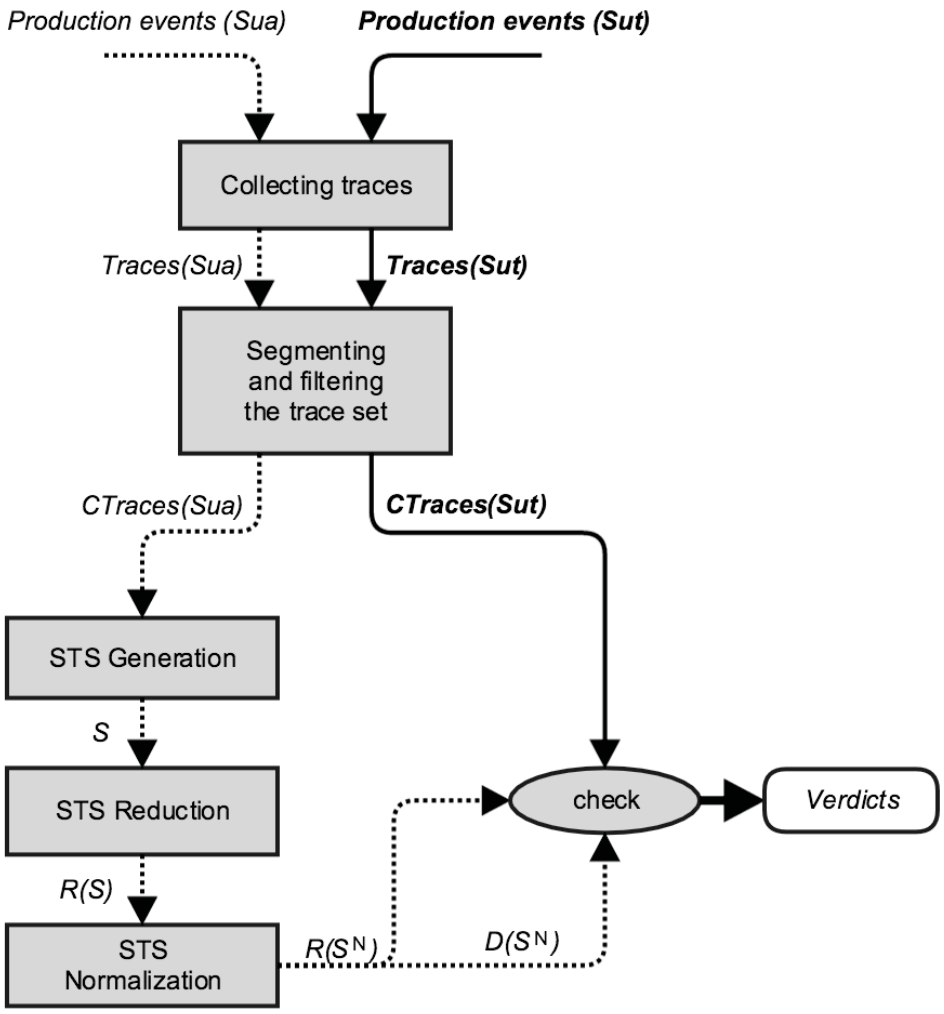

Fig. 5.1: Overview of *Autofunk v3* with the passive testing extension. While the previous *Autofunk* design has been kept, there are two new modules: "STS Normalization" and "check", representing the passive conformance testing part.

Definition 27 (Compatibility of $\mathcal{S}^N$ and Sut) *Let $\mathcal{S}^N = \{\mathcal{S}_1^N, \ldots, \mathcal{S}_n^N\}$ be a STS set, and Sut be a LTS (semantics), which is assumed to behave as the production system.*

$\mathcal{S}^N$ and Sut are said compatible *if and only if:* $\Sigma_{Sut} \subseteq \bigcup_{1 \leq i \leq n} \bigcup_{a(p) \in \Lambda_{\mathcal{S}_i^N}} a \times D_p$.

5.3.1 First implementation relation: $\leq_{ct}$

The first implementation relation, denoted by $\leq_{ct}$, refers to the trace preorder relation [DNH84; Vaa91] (cf. Example 2.1.4 on page 23). It aims at checking whether all the complete execution traces of Sut are pass traces of $\mathcal{S}^N = \{\mathcal{S}_1^N, \ldots, \mathcal{S}_n^N\}$. The first implementation relation can be written with the following definition:

Definition 28 (Implementation relation $\leq_{ct}$) *Let $\mathcal{S}^N$ be an inferred model of Sua, and Sut be the system under test. When Sut produces complete traces also captured by $\mathcal{S}^N$, we write: $Sut \leq_{ct} \mathcal{S}^N =_{def} CTraces(Sut) \subseteq Traces_{Pass}(\mathcal{S}^N)$.*

Pragmatically, the reduced model set $R(\mathcal{S}^N)$ sounds more convenient for passively testing Sut because it is strongly reduced in terms of size compared to $\mathcal{S}^N$. The test relation can also be written as below because both models $\mathcal{S}^N$ and $R(\mathcal{S}^N)$ are trace equivalent (cf. Proposition 25 in Chapter 4):

Proposition 29 $Sut \leq_{ct} \mathcal{S}^N \Leftrightarrow CTraces(Sut) \subseteq Traces_{Pass}(R(\mathcal{S}^N))$.

5.3.2 Second implementation relation: $\leq_{mct}$

As stated previously, the inferred model $\mathcal{S}^N$ of Sua is partial, and it might not capture all the behaviors that should happen on Sut. Consequently, our partner wants a *weaker implementation relation* that is less restrictive on the traces that should be observed from Sut. This second relation aims to check that, for every complete trace $t = (a_1(p), \alpha_1) \dots (a_m(p), \alpha_m)$ of Sut, we also have a set of traces of $Traces_{Pass}(\mathcal{S}^N)$ having the same sequence of symbols such that every variable assignment $\alpha_j(x)_{(1 \leq j \leq m)}$ of t is found in one of the traces of $Traces_{Pass}(\mathcal{S}^N)$ with the same symbol a_j.

Example 5.3.1 Figure 5.2 recalls the example considered in Chapter 4. The trace $t = (17011(\{nsys, nsec, point, pid\}), \{nsys := 1, nsec := 8, point := 1, pid := 1\})$ $17021(\{nsys, nsec, point, tpoint, pid\}, \{nsys := 1, nsec := 8, point := 4, tpoint := 9, pid := 1\})$ is not a *pass trace* of $\mathcal{S}^N$ because this trace cannot be extracted from one of the paths of the STS depicted in Figure 5.2, on account of the variables $point$ and $tpoint$, which do not take the expected values. Nonetheless, both variables are assigned with $(point := 4)$ and $(tpoint := 9)$ in the second path. This is interesting as it indicates that such values *may* be correct since they are actually used in a similar action in a similar path. Our second implementation relation aims at expressing that such a trace t captures a correct behavior as well.

The second implementation relation, denoted by $\leq_{mct}$, is defined as follows:

Definition 30 (Implementation relation $\leq_{mct}$) *Let $\mathcal{S}^N$ be an inferred model of Sua, and Sut be a system under test. We write: $Sut \leq_{mct} \mathcal{S}^N$ if and only if $\forall\ t = (a_1(p), \alpha_1) \dots (a_m(p), \alpha_m) \in CTraces(Sut)$, $\forall\ \alpha_j(x)_{(1 \leq j \leq m)}, \exists\ \mathcal{S}_i^N \in \mathcal{S}^N$ and $t' \in Traces_{Pass}(\mathcal{S}_i^N)$ such that $t' = (a_1(p), \alpha'_1) \dots (a_m(p), \alpha'_m)$ and $\alpha'_j(x) = \alpha_j(x)$.*

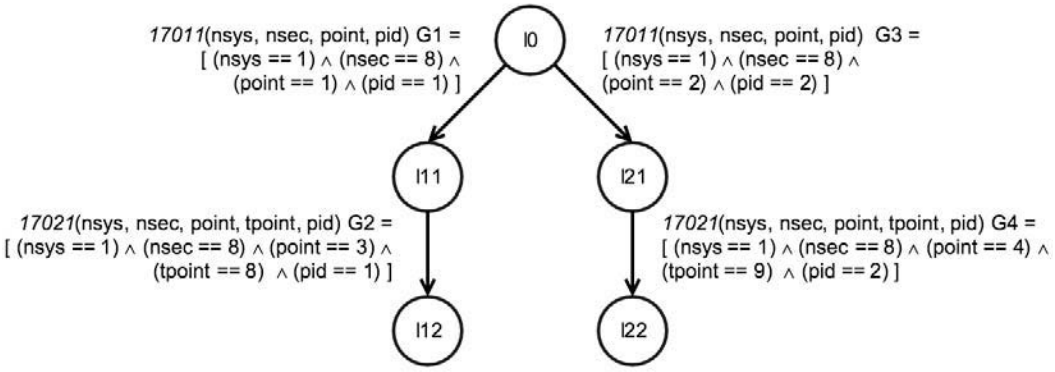

Fig. 5.2: The first Symbolic Transition System inferred in Chapter 4.

According to the above definition, the successive symbols and variable assignments of a trace $t \in CTraces(Sut)$ can be found into several traces of $Traces_{Pass}(\mathcal{S}_i^N)$, which have the same sequence of symbols $a_1 \dots a_m$ as the trace t. The reduced model $R(\mathcal{S}_i^N)$ was previously constructed to capture all these traces in $Traces_{Pass}(\mathcal{S}_i^N)$, having the same sequence of symbols. Indeed, given a STS $\mathcal{S}_i^N$, all the STS paths of $\mathcal{S}_i^N$, which have the same sequence of symbols labeled on the transitions, are packed into one STS path b in $R(\mathcal{S}_i^N)$ whose transition guards are stored into a matrix $M_{[b]}$.

Example 5.3.2 Given our trace example $t = (17011(\{nsys, nsec, point, pid\}), \{nsys := 1, nsec := 8, point := 1, pid := 1\})\ 17021(\{nsys, nsec, point, tpoint, pid\}, \{nsys := 1, nsec := 8, point := 4, tpoint := 9, pid := 1\})$, and the reduced model depicted in Figure 5.3, t is a pass trace with respect to $\leq_{mct}$ because each assignment $\alpha_j(x)$ satisfies at least one guard of the matrix line j. For instance, the assignment $(point := 4)$, which is given with the second valued event of t, satisfies one of the guards of the second line of the matrix $M_{[b]}$.

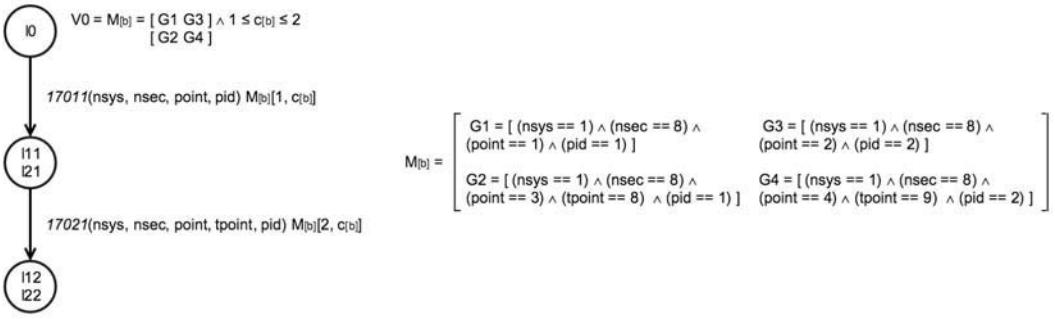

Fig. 5.3: Reduced Symbolic Transition System model (with its matrix) obtained from the model depicted in Figure 5.2.

Given a trace $(a_1(p), \alpha_1) \dots (a_m(p), \alpha_m) \in CTraces(Sut)$ and a STS path b of $R(\mathcal{S}_i^N)$ having the same sequence of symbols $a_1 \dots a_m$, the relation can now be formulated

as follows: for every valued event $(a_j(p), \alpha_j)$, each variable assignment $\alpha_j(x)$ must satisfies at least one of the guards of the matrix line j in $M_{[b]}[j, *]$.

Consequently, we propose to rewrite the implementation relation $\leq_{mct}$ as:

Proposition 31 *$Sut \leq_{mct} \mathcal{S}^N$ if and only if $\forall\ t = (a_1(p), \alpha_1) \dots (a_m(p), \alpha_m) \in CTraces(Sut)$, $\exists\ R(\mathcal{S}_i^N) \in R(\mathcal{S}^N)$ and $b = l0_{R(\mathcal{S}_i^N)} \xRightarrow{(a_1(p_1), M_{[b]}[1, c_{[b]}]) \dots (a_j(p_j), M_{[b]}[j, c_{[b]}])} l_m$ with $(1 \leq c_{[b]} \leq k)$ such that $\forall\ \alpha_j(x)_{(1 \leq j \leq m)}, \alpha_j(x) \models M_{[b]}[j, 1] \vee \dots \vee M_{[b]}[j, k]$, and $l_m \in Pass$.*

The disjunction of guards $M_{[b]}[j, 1] \vee \dots \vee M_{[b]}[j, k]$, found in the matrix $M_{[b]}$, could be simplified by gathering all the equalities $(x == val)$ together with disjunctions for every variable x that belongs to the parameter set p_j of $a_j(p_j)$. Such equalities can be extracted with the projection operator *proj* (see Definition 12). We obtain one guard of the form $\bigwedge_{x \in p_j} ((x == val_1) \vee \dots \vee (x == val_k))$. This can be expressed by deriving new STSs from $R(\mathcal{S})$.

STS $D(\mathcal{S})$

Given a STS $R(\mathcal{S}_i^N)$, the STS $D(\mathcal{S}_i^N)$ is constructed with the disjunction of guards described in the previous section:

Definition 32 (Derived STS $D(\mathcal{S}_i^N)$) *Let $R(\mathcal{S}_i^N) =< L_R, l0_R, V_R, V0_R, I_R, \Lambda_R, \rightarrow_R >$ be a STS of $R(\mathcal{S}^N)$. We denote by $D(\mathcal{S}_i^N)$ the STS $< L_D, l0_D, V_D, V0_D, I_D, \Lambda_D, \rightarrow_D >$ derived from $R(\mathcal{S}_i^N)$ such that:*

- $L_D = L_R$;
- $l0_D = l0_R$;
- $I_D = I_R$;
- $\Lambda_D = \Lambda_R$;
- *$V_D, V0_D$ and $\rightarrow_D$ are defined by the following inference rule:*

$$\frac{b = l0_R \xRightarrow{(a_1(p_1), M_{[b]}[1, c_{[b]}]) \dots (a_m(p_m), M_{[b]}[m, c_{[b]}])}_R l_m, (1 \leq c_{[b]} \leq k) \text{ in } V0_R}{\begin{array}{c} l0_D \xRightarrow{(a_1(p_1), M_b[1]) \dots (a_m(p_m), M_b[m])}_D l_m, V0_D = V0_D \wedge M_b, \\ M_b[j]_{(1 \leq j \leq m)} = \bigwedge_{x \in p_j} (proj_x(M_{[b]}[j, 1]) \vee \dots \vee proj_x(M_{[b]}[j, k])) \end{array}}$$

The set of derived models is denoted by $D(\mathcal{S}^N) = \{D(\mathcal{S}_1^N), \dots, D(\mathcal{S}_n^N)\}$.

The second implementation relation $\leq_{mct}$ can now be expressed by:

Proposition 33 *$Sut \leq_{mct} \mathcal{S}^N$ if and only if $\forall\ t = (a_1(p), \alpha_1) \dots (a_m(p), \alpha_m) \in CTraces(Sut)$, $\exists\ D(\mathcal{S}_i^N) \in D(\mathcal{S}^N)$ and $l0_{D(\mathcal{S}_i^N)} \xRightarrow{(a_1(p_1),G_1)\dots(a_m(p_m),G_m)} l_m$ such that $\forall\ \alpha_{j\ (1\leq j\leq m)}, \alpha_j \models G_j$ and $l_m \in Pass$.*

The implementation relation $\leq_{mct}$ now means that a trace of Sut must also be a pass trace of the model set $D(\mathcal{S}^N) = \{D(\mathcal{S}_1^N), \dots, D(\mathcal{S}_n^N)\}$. This notion of trace inclusion can also be formulated with the first implementation relation $\leq_{ct}$ as follows:

Proposition 34 *$Sut \leq_{mct} \mathcal{S}^N$ if and only if $CTraces(Sut) \subseteq Traces_{Pass}(D(\mathcal{S}^N))$, and $Sut \leq_{mct} \mathcal{S}^N \Leftrightarrow Sut \leq_{ct} D(\mathcal{S}^N)$.*

Now, the implementation relation $\leq_{mct}$ is expressed with the first relation $\leq_{ct}$, which implies that our passive testing algorithms shall be the same for both relations except that they shall take different reference models.

Furthermore, because $D(\mathcal{S}^N)$ is derived from $R(\mathcal{S}^N)$, *i.e.* the guards of the model $D(\mathcal{S}_i^N)$ are the disjunctions of the guards of the model $R(\mathcal{S}_i^N)$ (cf. Definition 32), we have $Traces_{Pass}(R(\mathcal{S}^N)) \subseteq Traces_{Pass}(D(\mathcal{S}^N))$. Therefore, if $Sut \leq_{ct} \mathcal{S}^N$, we have $Sut \leq_{mct} \mathcal{S}^N$:

Proposition 35 $Sut \leq_{ct} \mathcal{S}^N \implies Sut \leq_{mct} \mathcal{S}^N$.

In the next section, we introduce the offline passive testing algorithm that uses these two implementation relations.

5.3.3 Offline passive testing algorithm

Our offline passive testing algorithm, which aims to check whether the two previous implementation relations hold, is given in Algorithm 4. It takes the complete traces of Sut, as well as the model sets $R(\mathcal{S}^N)$ and $D(\mathcal{S}^N)$, with regard to Proposition 29 and Proposition 34. It returns the verdict "Pass$\leq_{ct}$" if the relation $\leq_{ct}$ is satisfied, and the verdict "Pass$\leq_{mct}$" if $\leq_{mct}$ is satisfied, along with the possibly fail traces with respect to $\leq_{ct}$, gathered in the set T_1. Otherwise, it returns both T_1, and the possibly fail traces with respect to $\leq_{ct}$, gathered in the set T_2.

Algorithm 4 relies upon the function $check(Trace\ trace, STS\ \mathcal{S})$ to check whether the trace $trace = (a_1(p), \alpha_1) \dots (a_m(p), \alpha_m)$ is a trace of $\mathcal{S}$. If a STS path b is composed of the same sequence of symbols as $trace$ (line 27), the function tries to find a matrix column (*i.e.* a vector) $M = M_{[b]}[*, c_{[b]}]$ $(1 \leq c_{[b]} \leq k)$ such that every

variable assignment α_j satisfies the guard $M[j]$. If such a column of guards exists, the function returns $True$, and $False$ otherwise (cf. Proposition 37).

Algorithm 4 covers every trace $trace$ of $CTraces(Sut)$, and tries to find a STS $R(\mathcal{S}_i^N)$ such that $trace$ is also a trace of $R(\mathcal{S}_i^N)$ with $check(trace, R(\mathcal{S}_i^N))$ (line 7). If no model $R(\mathcal{S}_i^N)$ is found, $trace$ is added to the set T_1 (line 11), which gathers the possibly fail traces with respect to $\leq_{ct}$. Thereafter, this algorithm performs the same step but using the STS $D(\mathcal{S}^N)$ (line 13). One more time, if no model $D(\mathcal{S}_i^N)$ is found, the trace $trace$ is added to the set T_2 (line 17), which gathers the possibly fail traces with respect to the relation $\leq_{mct}$. Finally, if T_1 is empty, the verdict "Pass$\leq_{ct}$" is returned, which means that the first implementation relation holds. Otherwise, T_1 is provided. If T_2 is empty, the verdict "Pass$\leq_{mct}$" is returned. Otherwise T_2 is returned.

When one of the implementation relations does not hold, this algorithm offers the advantage of providing the possibly fail traces of $CTraces(Sut)$. Such traces can be later analyzed to check if Sut is correct or not as these traces may be false positives, because of the partialness of the reference models for instance. That is very helpful for Michelin engineers because it allows them to only focus on what are potentially faulty behaviors, reducing debugging time, and making engineers more efficient.

Soundness of Algorithm 4

Proposition 36 *Let $\mathcal{S}^N$ be a STS set, and Sut be a LTS compatible to $\mathcal{S}^N$.*

$Sut \leq_{ct} \mathcal{S}^N \implies$ Algorithm 4 returns "Pass$\leq_{ct}$", "Pass$\leq_{mct}$".
$Sut \leq_{mct} \mathcal{S}^N \implies$ Algorithm 4 returns "Pass$\leq_{mct}$".

Sketch of proof: Proposition 36 can be split into three points:

1. $Sut \leq_{ct} \mathcal{S}^N \implies$ Algorithm 4 returns "Pass$\leq_{ct}$";
2. $Sut \leq_{mct} \mathcal{S}^N \implies$ Algorithm 4 returns "Pass$\leq_{mct}$";
3. $Sut \leq_{ct} \mathcal{S}^N \implies$ Algorithm 4 returns "Pass$\leq_{ct}$", "Pass$\leq_{mct}$".

For each point, Algorithm 4 relies on the function $check$:

Proposition 37 *Let $t \in CTraces(Sut)$ be a trace, and $\mathcal{S}^N$ a STS set such that $Sut \leq_{ct} \mathcal{S}^N$. $\leq_{ct}$ means that there exists a model $\mathcal{S}_i^N$ such that $t \in Traces_{Pass}(\mathcal{S}_i^N)$.*

$(\exists\ (1 \leq i \leq n) :\ t \in Traces_{Pass}(\mathcal{S}_i^N)) \implies$ *the function $check(t, \mathcal{S}_i^N)$ returns $True$.*

Sketch of proof: $\exists\ (1 \leq i \leq n),\ t \in Traces_{Pass}(\mathcal{S}_i^N)$ implies $\exists\ (1 \leq i \leq n),\ t \in Traces_{Pass}(R(\mathcal{S}_i^N))$. The $Traces$ and LTS semantics definitions imply: $\exists\ p = l0_{R(\mathcal{S}_i^N)} \xrightarrow{(a_1(p_1),G_1,A_1)\ldots(a_n(p_n),G_n,A_n)} l_n,\ l_n \in Pass$, and $t \in Traces_{Pass}(p)$.

Given the trace $t = (a_1(p), \alpha_1) \ldots (a_n(p), \alpha_n)$, the function $check(t, R(\mathcal{S}_i^N))$:

- seeks $b = l0_{R(\mathcal{S}_i^N)} \xrightarrow{(a_1(p_1),G'_1,A'_1)\ldots(a_n(p_n),G'_n,A'_n)} l_n,\ l_n \in Pass$ (line 27), with $M_{[b]}$ be the matrix $n \times k$ of b such that $G'_j = M_{[b]}[j, c_{[b]}]$ $(1 \leq c_{[b]} \leq k$ and $1 \leq j \leq n)$;
- ensures that $\exists\ (1 \leq c_{[b]} \leq k) : \alpha_j \models M_{[b]}[j, c_{[b]}]\ (1 \leq j \leq n)$ (lines 31-36).

The function $check(t, \mathcal{S}_i^N)$ seeks b such that $t \in Traces_{Pass}(b)$, and returns $True$ if b exists (line 38). Therefore, $\exists\ p = l0_{R(\mathcal{S}_i^N)} \xrightarrow{(a_1(p_1),G_1,A_1)\ldots(a_n(p_n),G_n,A_n)} l_n,\ l_n \in Pass \mid\ t \in Traces_{Pass}(p)$ implies the function $check(t, \mathcal{S}_i^N)$ returns $True$. □

Sketch of proof for (1): $Sut \leq_{ct} \mathcal{S}^N \Leftrightarrow CTraces(Sut) \subseteq Traces_{Pass}(R(\mathcal{S}^N))$ (Proposition 29) can be written as follows: $\forall\ t \in CTraces(Sut), \exists\ (1 \leq i \leq n) :\ t \in Traces_{Pass}(R(\mathcal{S}_i^N))$.

Given that, and according to Proposition 37, the function $check(t, \mathcal{S}_i^N)$ returns $True$ for every trace $t \in CTraces(Sut)$ (lines 6-9). Therefore the set T_1 is empty (line 18), and Algorithm 4 returns "Pass$\leq_{ct}$" (line 19). □

Sketch of proof for (2): $Sut \leq_{mct} \mathcal{S}^N \Leftrightarrow CTraces(Sut) \subseteq Traces_{Pass}(D(\mathcal{S}^N))$ (Proposition 34) can be written as follows: $\forall\ t \in CTraces(Sut), \exists\ (1 \leq i \leq n) :\ t \in Traces_{Pass}(D(\mathcal{S}_i^N))$.

Given that, and according to Proposition 37, the function $check(t, \mathcal{S}_i^N)$ returns $True$ for every trace $t \in CTraces(Sut)$ (lines 12-15). Therefore the set T_2 is empty (line 21), and Algorithm 4 returns "Pass$\leq_{mct}$" (line 22). □

Sketch of proof for (3): $Sut \leq_{ct} \mathcal{S}^N \implies Sut \leq_{mct} \mathcal{S}^N$ (Proposition 35), therefore Algorithm 4 returns "Pass$\leq ct$", "Pass$\leq_{mct}$". □

Complexity of Algorithm 4

The complexity of the function $check(t, \mathcal{S}_i^N)$ is $\mathcal{O}(m \times k)$ with m the number of valued events in the trace t (*i.e.* its length), and k the number of columns in $M_{[b]}$, which is likely large as reduced models still express all complete behaviors found in the

traces of a system under analysis. Finding a branch in a model is negligible thanks to the hash mechanism, hence we only take the matrix traversal into account.

The complexity of Algorithm 4 is $\mathcal{O}(t \times n \times (m \times k))$ with t the number of complete traces of Sut, n the number of models, and $(m \times k)$ the complexity of the $check(t, \mathcal{S}_i^N$ function. Compared to the number of traces and columns, the number of models n is negligible (*i.e.* $n << t$ and $n << k$), which means that the overall complexity is $\mathcal{O}(t \times m \times k)$.

In our first experiments, we found that $m << k$, hence the complexity of the function $check(t, \mathcal{S}_i^N)$ can be updated to $\mathcal{O}(k)$, and the overall complexity becomes: $\mathcal{O}(t \times k)$ (with $t \approx k$).

Algorithm 4: Offline passive testing algorithm

Input : $R(\mathcal{S}^N), D(\mathcal{S}^N), CTraces(Sut)$
Output : Verdicts and/or possibly fail trace sets T_1, T_2

BEGIN;
$T_1 = \emptyset$;
$T_2 = \emptyset$;
foreach $trace \in CTraces(Sut)$ **do**
  $check = False$;
  for $i = 1, \ldots, n$ **do**
    if `check` *(trace,* $R(\mathcal{S}_i^N)$*)* **then**
      $check = True$;
      break;
  if $check == False$ **then**
    $T_1 = T_1 \cup \{trace\}$;
    for $i = 1, \ldots, n$ **do**
      if `check` *(trace,* $D(\mathcal{S}_i^N)$*)* **then**
        $check = True$;
        break;
    if $check == False$ **then**
      $T_2 = T_2 \cup \{trace\}$;
if $T_1 == \emptyset$ **then**
  return "Pass$\leq_{ct}$";
else
  if $T_2 == \emptyset$ **then**
    return "Pass$\leq_{mct}$" and T_1;
  else
    return T_1 and T_2;
END;

Function *check(Trace trace, STS* $\mathcal{S}$*) : boolean* **is**
  if $\exists\ b = l0_{\mathcal{S}} \xrightarrow{(a_1(p_1),G_1,A_1)\ldots(a_m(p_m),G_m,A_m)} l_m \mid trace = (a_1(p), \alpha_1) \ldots (a_n(p), \alpha_m)$ *and* $l_m \in Pass$ **then**
    $M_{[b]} = Mat(b)$ is the matrix $m \times k$ of b;
    $c_{[b]} = 1$;
    while $c_{[b]} \leq k$ **do**
      $M = M_{[b]}[*, c_{[b]}]$;
      $sat = True$;
      for $j = 1, \ldots, m$ **do**
        if $\alpha_j \not\models M[j]$ **then**
          $sat = False$;
          break;
      if $sat == True$ **then**
        return $True$;
      $c_{[b]} + +$;
  return $False$;

5.4 Implementation and experimentation

In this section, we summarize the work done on the different *Autofunk* implementations for Michelin. Then, we give a few results on our offline passive testing technique.

5.4.1 Implementation

By adding a testing module to *Autofunk v3*, we have developed a complete tool for testing production systems at Michelin.

According to the *cloc* tool[1], the latest version of *Autofunk* has 2831 lines of code written in Java 8, along with 4 Drools rules used in Layer 1 of the model inference module. According to *JaCoCo*[2] (which stands for Java Code Coverage) tool, the test suite (compound of 119 test cases) covers 90% of the code, with both unit and functional tests. Our algorithms are extensively tested.

Users of *Autofunk v3* interact with it using the command line. We designed parallelizable algorithms that we have used in combination with Java 8 streams and parallel processing abilities. This gives interesting performance results on multi-core processors, without affecting code readability. *Autofunk v3* still embeds Drools, and can either run a local Spark instance or connect to a Spark cluster (for the k-means clustering), which is better for performance. Models are persisted on disk using the *Kryo*[3] serialization library. Finally, this tool is highly configurable thanks to the (Typesafe) *Config*[4] library.

Table 5.1 presents the differences across all *Autofunk* versions. All versions use Drools to perform model inference, but only *Autofunk v3* can perform testing as explained in this chapter. *Autofunk v1* does not segment the trace set because it is primarily used for web applications, for which this task does not apply. *Autofunk v3* is heavily based on *Autofunk v2*, hence both use a context-specific reduction technique by means of branch equivalence classes. Yet, *Autofunk v3* uses MLlib for its k-means implementation, and is more extensively tested.

5.4.2 Experimentation

We conducted some experiments with real sets of production events, recorded in one of Michelin's factories at different periods of time. The results given in this

[1] `https://github.com/AlDanial/cloc`
[2] `https://github.com/jacoco/jacoco`
[3] `https://github.com/EsotericSoftware/kryo`
[4] `https://github.com/typesafehub/config`

Autofunk	Use Drools?	Perform inference?	Perform testing?	Segmentation algorithm	Reduction algorithm	Code coverage
v1	✓	✓	×	None	Bisimulation minimization	<50%
v2	✓	✓	×	Statistical analysis	Branch equivalence classes	70%
v3	✓	✓	✓	K-means (Spark / MLlib)	Branch equivalence classes	90%

Tab. 5.1: Summary of the different *Autofunk* versions. *Autofunk v3* is based on *Autofunk v2*, which has been developed from scratch (even though inspired by *Autofunk v1*).

section are focused on our offline passive testing technique, built on-top of *Autofunk v3*. We executed our implementation on a Linux (Debian) machine with 12 Intel(R) Xeon(R) CPU X5660 @ 2.8GHz and 64GB RAM.

Table 5.2 shows the results of three experiments on the same production system with different trace sets, recorded at different periods of time, with the latest *Autofunk* version. The first column shows the experiment number (#), columns 2 and 3 respectively give the sizes of the trace sets of the system under analysis Sua and of the system under test Sut. The two next columns show the percentage of pass traces with respect to the relations $\leq_{ct}$ and $\leq_{mct}$. The last column indicates the execution time (in minutes) for the testing phase.

#	$Card(CTraces(Sua))$	$Card(CTraces(Sut))$	$Card(T_1)$	$Card(T_2)$	Time
1	2,075	2,075	0	0	1
2	53,996	2,075	62	1452	4
3	53,996	25,047	500	500	10

Tab. 5.2: This table shows the results of our offline passive testing method based on a same specification.

In Experiment 1, we decided to use the same production events for both inferring models, *i.e.* specifications, and testing. This experiment shows that our implementation behaves correctly when trace sets are similar, *i.e.* when behaviors of both Sua and Sut are equivalent. That is why there are no possibly fail traces.

Experiment 2 has been run with traces of Sut that are older than those of Sua, which is unusual as the *de facto* usage of our framework is to build specifications from a production system Sua, and to take a newer or updated system as Sut. Here, only 30% (1452 traces in T_2) of the traces of Sut are pass traces with respect to the second implementation relation (same sequence of symbols with different values). There are two explanations: (i) the system has been strongly updated between the two periods of record (4 months), and (ii) production campaigns, *i.e.* grouping of planned orders and process orders to produce a certain amount of products over a certain period of time, were different (revealed by *Autofunk*, indicating that values for some key parameters were unknown).

Finally, experiment 3 shows good results as the specification models are rich enough, *i.e.* built from a larger set of traces (10 days) than the one collected on Sut. Such an experiment is a typical usage of our framework at Michelin. The traces of Sut have been collected for 5 days, and it took only 10 minutes to check conformance. While 98% of the traces are pass traces, the remaining 2% (500 traces) are new behaviors that never occurred before. Such a piece of information is essential for Michelin engineers to detect potential regressions. Even though 2% may represent

a large set to analyze (500 traces in this experiment), *Autofunk* eases the work of Michelin engineers by highlighting the traces to focus on. Instead of having to check 25,000 traces manually, they only have to check 500 traces, which is a significant improvement on a daily basis. Yet, such a subset may contain false positives depending on the richness of the models, but using large sets of traces to infer the models usually reduces the number of false positives.

5.5 Conclusion

In this chapter, we presented a fast passive testing framework built on-top of our model inference framework *Autofunk v3*, which combines different techniques such as model inference, expert systems, and machine learning.

In this work, we focus on complete traces exclusively because production systems run continuously, and a few irrelevant behaviors are likely to happen, *e.g.*, collecting traces can be turned either on or off at any time, but also human operators in a factory can act on the products. Tracking such irrelevant behaviors would be inefficient as it would mean more processing time for results of no interest.

Given a large set of production events, our framework infers exact models whose traces are included in the initial trace set of a system under analysis. Such models are then reused as specifications to perform offline passive testing using a second set of traces recorded on a system under test. Using two implementation relations based on complete trace inclusion, *Autofunk* is able to determine what has changed between the two systems. This is particularly useful for our industrial partner Michelin because potential regressions can be detected while deploying changes in production. Initial results on this offline method are encouraging, and Michelin engineers see a real potential in this framework.

Given the preliminary results, We know that 2% of a large trace set (as mentioned in the previous section) still represents many traces, which may be difficult to analyze. Nonetheless in our manufacturing context, this is still valuable because, before *Autofunk*, engineers had to manually test everything by hand. Now, *Autofunk* performs most of the work automatically, and engineers only have to manually check a small subset of traces compared to the initial trace set, which saves a lot of time.

In the next chapter, we give our thoughts on how to improve *Autofunk* as well as perspectives for future work.

Conclusions and future work

6

Contents

6.1 Summary of achievements

This thesis has proposed a novel approach to infer models of software systems in order to perform conformance testing of production systems, with a technique that leverages such models. The original aims and objectives of the thesis were as follows:

- To infer partial yet exact models of production systems in a fast and efficient manner, based on the data exchanged in a (production) environment;
- To design a conformance testing technique based on the inferred models, targeting production systems. The main idea was to detect regressions across similar production systems (*e.g.*, a software update or a hardware upgrade).

Chapters 3 and 4 addressed the first of the objectives by proposing two approaches combining model inference, machine learning, and expert systems to infer exact models for web applications and production systems, wrapped into the *Autofunk* framework. The expert system is composed of rules, capturing the knowledge of human experts, and used either to filter the trace set to remove the undesired ones,

to infer Symbolic Transition Systems (STSs), or to build more abstract STSs. In Chapter 4, the state merging is replaced with a context-specific state reduction based on an event sequence-based abstraction. This state reduction can be seen as the kTail algorithm [BF72] where k is as high as possible for every initial branch (or path) of the original Symbolic Transition System (STS). This state reduction ensures that the resulting models do not over-approximate the system under analysis, but it is also very context-specific, and cannot be generalized. We also showed that our approach is scalable: it can take thousands and thousands of traces and can still build models quickly thanks to our specific state merging process.

The second objective was achieved by enhancing *Autofunk* with a passive testing technique, presented in Chapter 5. Given a large set of production events, *Autofunk* reuses the inferred models as specifications to perform offline passive testing, using a second set of traces recorded on a system under test, and two implementation relations to determine what has changed between the two systems. This is particularly useful for our industrial partner Michelin because potential regressions can be detected while deploying changes in production.

The next section introduces some perspectives for future work on model inference. Section 6.3 is dedicated to future work on the testing part of our work. Section 6.4 closes this thesis.

6.2 New challenges in model inference

Model inference is a research field that has received a lot of attention over the past three decades, and it is still gaining ground with the emergence of new kinds of applications. Many recent works consider model inference to later perform analyses of the models or automatic testing in order to check different aspects of the software system such as robustness, security, and even regression testing, as presented in this thesis. Nonetheless, model inference still has a few drawbacks, which require further investigation, and we believe that the next three major directions could be very beneficial.

6.2.1 Building exact, or rather, more precise models

When the inferred models are used for analysis purpose (verification or testing), they must be as precise as possible. From the literature, we observed that the main feature leading to over-approximation is the state merging process. A trivial solution would be to use minimization techniques instead, *e.g.*, a bisimulation minimization as described in Chapter 3 of this thesis. As a reminder, the bisimulation relation associated with a minimization technique merges the state sets that are bisimilar

equivalent. This relation is stronger than a classical trace equivalence relation, but it may be considered too strong since the bisimulation minimization usually does not merge enough states, and thus may still produce large models. If another more suitable relation can be used, and if verification or testing techniques can work well with larger yet more precise models, which results can we expect? In this thesis, we provided a preliminary answer to this question (which is context-specific), but there is still a lot of work that could be done.

Another solution to limit non-approximation is to define and estimate *quality metrics* [Ton+12; Lo+12] to guide the model construction. In [Ton+12], three metrics, related to over- and under-approximation rates and model size, are measured to balance over-approximation and under-approximation of the inferred models with two search-based algorithms: (i) a multi-objective genetic algorithm, and (ii) the non-dominated sorting genetic algorithm *NSGA-II* [Deb+02]. But this process is time-consuming and can only be applied to small systems because the complete models, *i.e.* the models compound of all the observations, are incrementally re-generated from scratch to improve the metrics. An iterative process performed by adding the observations one after another in the model could also be considered. Other metrics could also be chosen depending on the context of the software system. In *Autofunk*, a similar improvement would be to build submodels of a production system. By now, we consider a whole workshop as a production system to infer models. We distinguish the production lines, but apart from that, we do not make any distinction among the different parts of the workshop. Nonetheless, there are parts that are more critical than the others, at least in Michelin's workshops. Being able to focus on specific locations of a workshop would be interesting to build smaller models, and this should bring significant improvements to the end users of *Autofunk*.

6.2.2 Scalability as a first-class citizen

In this thesis, we tackled the problem of inferring models from production systems. Such systems are distributed over several devices, and generate thousands of events a day. Collecting, storing, and analyzing such amount of data becomes more and more complicated, and model inference algorithms have to take these points into account. To our knowledge, too few studies [Yan+06; PG09] take scalability into account. That is also why we have proposed techniques that scale well in this thesis. We shew that adopting contextual algorithms was interesting for performance (*e.g.*, our context-specific reduction). We also believe that the use of specific parallel programming paradigms and heuristics would be an interesting addition to quickly build models or to find state equivalence classes. That is what we did in our Java implementation. Besides, such challenges are close to those of what we now call "big

data" [Hu+14]. This term not only defines the large volume of data but also new processing algorithms and applications since the traditional ones are inadequate.

To pursue this path, we could consider other kinds of systems because applications in the Industry become more and more complex, especially with the rise of *Service-Oriented Architectures* (SOA), distributed systems, and more recently *microservices*[1] [Tho15]. For instance, microservices are emerging as a new architectural style, aiming at creating software systems as a set of small services, *i.e.* small applications, each developed and deployed on its own. This package of services form the complete software system. Inferring models of such systems is not doable with most of the existing model inference techniques. On the other hand, *Autofunk* can use many data sources to infer models. We chose to gather heterogeneous events between different devices and software, which could be a path to follow if one would want to infer models of microservices as each service owns its data.

6.2.3 Bringing together different methods and research fields

Some papers chose to combine different algorithms for optimizing model inference. For instance, several works [Alu+05; Raf+05; Mer+11] replaced teachers and oracles with testers to answer queries. Other works [AN13; Yan+13] combined static analyses of source code with crawlers to increase code coverage rates, and reduce exploration time. Other research domains, such as machine learning and data mining, have also been considered to avoid the classical state merging stage. Ammons *et al.* [Amm+02] developed a machine learning approach, called specification mining to infer state machines. The authors focused on the most frequent interaction patterns found in a scenario set. In this thesis, we also adopted machine learning to automatically slice a trace set into several subsets so that we can infer several models of a production system in a workshop (cf. Chapter 4 • Section 4.6 (page 104)). Leveraging different domains such as the ones mentioned here sounds promising for optimizing model inference. As an example, state merging might be replaced with a kind of mechanism that would be automatically extracted from the characteristics or the context of the software system.

In the next section, we give some perspectives related to our implementation of *Autofunk*, the second main line of our work (*i.e.* software testing), and several ideas for future works.

[1] `http://martinfowler.com/articles/microservices.html`

6.3 Testing and beyond

Despite the promising results obtained by *Autofunk* in Chapter 5, there is room for improvement. The next section is focused on *Autofunk*'s usability, *i.e.* how, we think, *Autofunk* could be enhanced to be more widely used.

On the other hand, we only covered offline passive testing of production systems in this thesis. To go further, we give the insight of an online passive version in Section 6.3.2, and we discuss the integration of some active testing concepts into *Autofunk* in Section 6.3.3. Section 6.3.4 introduces our thoughts on data mining, which are semi-related to the previous section on active testing. Finally, Section 6.3.5 discusses our main assumption used throughout this thesis, *i.e.* we infer models of systems that behave correctly, and what we could do to reject it.

6.3.1 Improving usability

Our implementation of *Autofunk* has primarily been built to validate our work, but also to fill the gap between research and industrial applicability, thanks to our partner Michelin. At the time of writing, *Autofunk* is a Java console application with about 3,000 lines of code, and 119 unit tests covering 90% of the code.

Nevertheless, it is clear that this tool is still a prototype, and not a production-ready tool. As pointed out in this thesis, memory consumption remains an issue for example, because we load all objects in memory in order to act on them. At the time of writing, we partially fixed this problem by reducing *Autofunk*'s memory footprint with better object representations in memory, but it is not future-proof. With the rise of big data technologies and tools, it should be possible to find a better solution to this issue. For instance, *Autofunk v3* includes *Apache Spark*[2], a framework for large-scale data processing, which, among all, provides an implementation of the k-means clustering algorithm. Such a framework is designed to handle large data sets. It should be possible to make *Autofunk* more efficient by adapting our algorithms on-top of Apache Spark or any similar big data framework, *e.g.*, with a *Map-Reduce* approach [DG08]. This is a programming model that allows to process large data sets with distributed parallel algorithms. Most of the algorithms presented in this thesis are already executable in parallel, but not distributed yet. Nonetheless, running *Autofunk* on a computer cluster, *i.e.* a set of interconnected computers seen as a single logical unit, should bring significant performance improvements, but it should also refine the overall scalability.

[2] `https://spark.apache.org/`

By now, inference rules written with the Drools rule language, which are at the heart of *Autofunk*, have to be packaged within the Java application. It would be better to allow the configuration of such rules at runtime, *e.g.*, using a graphical user interface. This could also be helpful to write and manage the different sets of inference rules, which is an issue we already mentioned earlier in this thesis. An interface may mitigate such a drawback, but writing such inference rules remains a delicate task. That is why we would like to investigate different approaches to avoid such a labor. As we are already familiar with machine learning, we would like to pursue in this path by proposing a machine learning technique that replaces some of the inference rules. For instance, because the events in a production system are text-based and readable, the inference rules needed during the filtering step might be replaced by a method inspired by the works on automated text categorization [Seb02]. Yet, it is manifest that the inference rules used to lift abstraction of the models, as presented in Chapter 3 • Section 3.3.3 (page 62), still have to be written, and cannot be easily replaced, because they strongly depend on the business.

Another point that would be worth working on is the *visualization* of the inferred models and test results. At the time of writing, *Autofunk* is able to generate graphs representing the generated models, but they are not really usable in practice because of the size of the models. In most cases, the models represent behaviors of a production system. The models fit a system's physical layout in a factory. When a possibly fail trace is raised, it would be nice to highlight the possibly faulty behavior directly using the layout of the production system under test. That way, an engineer would have all the information required to quickly determine what caused such a behavior, and state whether it is a bug or a false positive. We could relate this idea to the research field on *fault localization* [Jon+02; WD10], except that we would locate faults in a physical manner in a factory.

Finally, we would like to reduce the number of false positives yield by our test engine as presented in Chapter 5 • Section 5.4 (page 122). For the record, false positives are behaviors that are considered possibly faulty, even though they are correct, because such behaviors are not part of the reference models. We already know that inferring reference models from large sets of traces reduces the number of false positives, but there might be other methods to overcome this issue. For instance, we proposed a weaker implementation relation that works well to avoid false positives with similar behaviors that are (partially) known. Unfortunately, this is not simply a problem of *safe* (regression) test selection [Ors+04] because we cannot plainly reject all new behaviors (as we could do in some more traditional testing scenarios). A naive approach would be to teach the test engine to recognize the new yet correct behaviors, but it seems cumbersome. Instead, we believe that improving *Autofunk*'s testing module with an online mode and some active testing concepts may be more effective.

6.3.2 Online passive testing

In Chapter 5, we presented an offline passive testing technique, which we started to adapt in order to propose an online passive testing technique as well. Both offline and online modes are not completely unalike, they also serve different purposes. Our online passive testing approach records traces on a system under test on-the-fly, and then checks whether those traces satisfy specifications, still generated from a system under analysis. It enables what we call *just-in-time fault detection*. Faults can be revealed in near real-time on a running system so that users can be notified as soon as possible. Here, our approach only indicates when a fault has been detected.

In online mode, we do not have complete traces (such as $CTraces(Sut)$), but traces that are constructed on-the-fly by an *instance* of tester, every time an event is received. Yet, we still consider a set of filtered traces, and not $Traces(Sut)$ directly, because it is possible to reuse the set of rules of Layer 1 in the model inference process (cf. Chapter 4 • Section 4.3.1 (page 86)), so that irrelevant events are filtered out.

Each new production event passes through a *proxy* whose role is to distribute the incoming events across the different instances of tester. The proxy functioning is given in Algorithm 5. Each event is filtered (line 4) and transformed into a filtered valued event (line 6), from which we extract its product identifier pid (line 7). The valued event is then forwarded to the two right testers (lines 14 and 15), *i.e.* the two instances for this pid, one for each STS set ($R(\mathcal{S}^N)$ and $D(\mathcal{S}^N)$). If there is no instance for a given model set yet, we create a new tester for this pid first (lines 8-13).

At this point, there should be two instances of tester running the same algorithm per pid, *i.e.* per product being manufactured. Such an algorithm is of type *checker-state*, which, for each valued event received (sent by the proxy), constructs runs based on the models in either $R(\mathcal{S}^N)$ or $D(\mathcal{S}^N)$, until it reaches:

- a verdict location (cf. Chapter 5 • Section 5.2 (page 111)), which means that the trace complies with the reference model set chosen (either $R(\mathcal{S}^N)$ or $D(\mathcal{S}^N)$);
- an unexpected event that has been received or a guard that has not been satisfied, which leads to a $Fail$ verdict;
- a *deadlock*, which happens when no event is received after a certain delay, which may lead to a $Fail$ verdict too. This situation may occur when a product is removed from the production line by a human operator for instance.

Algorithm 5: Online passive testing proxy

Input : Production events, model sets $R(\mathcal{S}^N)$ and $D(\mathcal{S}^N)$
Output : Verdicts: $Fail_{\leq ct}$ or $Fail_{\leq mct}$

```
1  BEGIN;
2  Instances = ∅;
3  while production event event do
4      if event is filtered by Layer 1 rules then
5          continue;
6      transform event into a valued event (a(p), α);
7      extract the product identifier pid' from (a(p), α);
8      if ∄ i1_pid ∈ Instances | pid == pid' then
9          create new tester i1_pid(R(S^N)) with pid = pid';
10         Instances = Instances ∪ {i1_pid};
11     if ∄ i2_pid ∈ Instances | pid == pid' then
12         create new tester i2_pid(D(S^N)) with pid = pid';
13         Instances = Instances ∪ {i2_pid};
14     forward (a(p), α) to i1_pid;
15     forward (a(p), α) to i2_pid;
16     if i1_pid has returned T ≠ ∅ then
17         return Fail_{≤ct};
18     if i2_pid has returned T ≠ ∅ then
19         return Fail_{≤mct};
20 END;
```

When it cannot construct any run (or when a deadlock is detected), it means that the observed trace does not lead to a $Pass$ location, hence this trace is considered a possibly fail trace. We introduce two new verdicts $Fail_{\leq ct}$ and $Fail_{\leq mct}$ that indicate whether a trace does not comply with either $R(\mathcal{S}^N)$ or $D(\mathcal{S}^N)$. Such verdicts are returned by the proxy given in Algorithm 5, depending on the result of the execution of the testers (lines 16-19).

Algorithm 6 is executed by each tester instance. Its aim is to construct a trace $trace$ on-the-fly, and to returns a trace set T, which is empty if the current trace complies with the model set $\mathcal{S}^N$, or not empty when there is either a deadlock or a guard that has not been satisfied, *i.e.* when a fault has been detected.

Each model $\mathcal{S}_i^N$ has its own set of runs $RUNS_i$, which contains tuples of runs (r) and column indices (col). Each run set is initialized with a tuple $(q_0, 0) \mid r = q_0, col = 0$ (line 3). When receiving a new valued event, the algorithm constructs a trace (line 5). For each model $\mathcal{S}_i^N$, the algorithm tries to find a tuple (r, col) in $RUNS_i$ so that the location l associated with its run's state q_{j-1} has, at least, one transition with the symbolic action $a(p)$ (line 8).

If the current tuple is the initial one $(q_0, 0)$, the algorithm loops over all transitions t (in all branches b) having the symbolic action $a(p)$ with $G = M_{[b]}[j, *]$. Here, the goal is to find the columns $c_{[b]}$ in $M_{[b]}$ for which $\alpha \cup v$ satisfies the guard $M_{[b]}[j, c_{[b]}]$ (lines 10-12). When it finds a column that is satisfied, the algorithm computes a new state q_{next}, and constructs a new run by completing r with the current valued event and the new state: $r \cdot (a(p), \alpha) \cdot q_{next}$ (line 13). This new run is added to the run set $RUNS'$ (line 14), which contains all runs found for a current valued event.

When the current tuple is not $(q_0, 0)$, the algorithm is already aware of the column (col) in $M_{[b]}$, hence it does not need to retrieve it. For each transition t with $G = M_{[b]}[j, col]$, the algorithm determines whether $\alpha \cup v$ satisfies $M_{[b]}[j, col]$ (lines 16-17). In this case, it also computes a new state q_{next}, and completes the current run r as before, which leads to a new run added to $RUNS'$ (line 19).

Once all runs have been covered, we replace the runs in $RUNS_i$ with those in $RUNS'$ (line 20). Once all models $\mathcal{S}_i^N$ have been covered, we check the presence of, at least, one run in any of $RUNS_i$ (line 21). If there is no run, the algorithm has not been able to find a transition that can be fired with the last received valued event, *i.e.* there is no more transition or the guards have not been satisfied. That is why it has not been able to create any run, and all run sets $RUNS_i$ are therefore empty. Such a situation means that the trace does not comply with $\mathcal{S}^N$, and we return a non-empty trace set $T = \{trace\}$ (line 22), which is handled by the proxy to return the right verdict (cf. Algorithm 5).

When we do not receive any valued event anymore, the algorithm has reached a *deadlock* situation. There are two cases depending on the location l of the last state q of the run r, for all $(r, col) \in RUNS_i$ (line 25). If l is not a verdict location, the trace does not comply with $\mathcal{S}^N$, and we return a non-empty trace set $T = \{trace\}$. Otherwise, we return an empty set T (line 27).

Algorithm 6: Online passive testing algorithm

Input : A STS set $\mathcal{S}^N = \{\mathcal{S}_1^N, \ldots, \mathcal{S}_n^N\}$, valued events $(a(p), \alpha)$
Output : A set of traces T, which may be empty

BEGIN;
$trace = \emptyset$;
$RUNS_i = \{(q_0, 0) \mid q_0 = (l0_{\mathcal{S}_i^N}, V0_{\mathcal{S}_i^N})\},\ (1 \leq i \leq n)$;
while *valued event* $(a(p), \alpha)$ **do**
    $trace = trace \cdot (a(p), \alpha)$;
    for $i = 1, \ldots, n$ **do**
        $RUNS' = \emptyset$;
        foreach $(r, col) \in RUNS_i \mid r = q_0(a_1(p), \alpha_1) \ldots q_{j-1}$ *with* $q_{j-1} = (l, v) \in L_{\mathcal{S}_i^N}$ *and* $l \overset{a(p)}{\Longrightarrow}$ **do**
            if $(r, col) == (q_0, 0)$ **then**
                foreach $t = l \xrightarrow{a(p),G,A} l_{next} \in \rightarrow_{\mathcal{S}_i^N}$ *with* $G = M_{[b]}[j, *]$ **do**
                    for $c_{[b]} = 1, \ldots, k$ **do**
                        if $\alpha \cup v \models M_{[b]}[j, c_{[b]}]$ **then**
                            $q_{next} = (l_{next}, v_{next} = A(v \cup \alpha))$;
                            $RUNS' = RUNS' \cup \{(r \cdot (a(p), \alpha) \cdot q_{next}, c_{[b]})\}$;
            else
                foreach $t = l \xrightarrow{a(p),G,A} l_{next} \in \rightarrow_{\mathcal{S}_i^N}$ *with* $G = M_{[b]}[j, col]$ **do**
                    if $\alpha \cup v \models M_{[b]}[j, col]$ **then**
                        $q_{next} = (l_{next}, v_{next} = A(v \cup \alpha))$;
                        $RUNS' = RUNS' \cup \{(r \cdot (a(p), \alpha) \cdot q_{next}, col)\}$;
        $RUNS_i = RUNS'$;
    if $\forall\ i \in \{1, \ldots, n\},\ RUNS_i == \emptyset$ **then**
        return $\{trace\}$;
// No more valued event received (deadlock)
$T = \emptyset$;
if $\forall\ i \in \{1, \ldots, n\}, \exists (r, col) \in RUNS_i \mid r$ *ends with* $q = (l, v)$ *and* $l \notin Pass$ **then**
    $T = T \cup \{trace\}$;
return T;
END;

At the time of writing, there are still open-ended questions regarding this online method to devise a complete online passive algorithm. First of all, we need to find a way to properly identify deadlocks. We know that products can stay for days in a production system, hence it is complicated to set a timed limit. Another approach would be to compute average delays between the events in the models $R(\mathcal{S}_i)$ thanks to the event's time stamps, which can be retrieved in the valued events.

The current algorithm also does not support the presence of repetitive patterns. For the record, a repetitive pattern is "a sequence of valued events that should contain at least one valued event", which we chose to remove during the model inference because they do "not express a new and interesting behavior". It was not an issue in offline passive testing because we applied the same process on the traces of both Sua and Sut, but in online mode, it is not possible anymore as we construct a trace on-the-fly. It might be possible to deal with the repetitive patterns in our online technique by considering a set of patterns P, defined *a priori*, for which the algorithm would be able to detect them on-the-fly, and to construct a special run with a *cycle* to express the detected pattern.

The issue above is linked to the fact that we get a stream of production events from Sut, which we filter to remove irrelevant events only (*i.e.* is incomplete traces). Yet, we are not able to remove incomplete traces on-the-fly. This is worth noting because it will likely lead to more false positives in online mode than in offline mode (for which we consider complete traces of Sut exclusively).

Last but not least, running two testers per product (pid) will result in many instances executed in parallel as there are thousands of products in a factory at any given time. Such a situation may not scale well, especially with the notion of deadlocks mentioned previously. Indeed, we might have too many instances executed in parallel, which would freeze the testing process. One option to avoid too many instances at the same time would be to maintain two fixed pools of instances for both the model sets $R(\mathcal{S}^N)$ and $D(\mathcal{S}^N)$, and to store the runs $RUNS_i$ out of these instances (*e.g.*, in a in-memory database). That way, the instances could be reused. Nonetheless, there are many events exchanged at the same time in a production system, and we need to find a non-blocking solution to accept and handle these events.

6.3.3 Integrating active testing with *Autofunk*

Active testing, as defined in Chapter 2.1 • Section 2.1.3 (page 24), works by stimulating a system under test. We chose not to take this direction because stimulating a production system might break it if one sends incorrect data. Indeed, as there are physical devices behind software, it could lead to severe damages.

Nonetheless, and according to our partner Michelin, it should be possible to reproduce a production environment in a simulation room, and thus to simulate a whole production system without the physical devices (only the logical controllers). We could then leverage our inferred models, which contain the data collected from a real environment, to construct a set of inputs, which we would reuse on the system under test in simulation room. It is worth mentioning that this principle is not strictly active testing because we do not generate the test cases that lead to verdicts. Yet, we believe that adding such an active approach to *Autofunk* would speed up the testing process significantly by avoiding to collect traces of a system under test for a long period, and it would also make its adoption easier.

A simplistic way to integrate active testing with *Autofunk* would be to "replay" [TH00; OK05] the data previously "recorded", *i.e.* the data available in the inferred models. This is a path Michelin would like to explore. Figure 6.1 gives the insight of their use case. A system under analysis Sua in a production environment is used to build a first set of models. A system under test Sut, which is likely a different version of the system under analysis, is set up in a simulation environment (as already mentioned before). Replaying the data from Sua in Sut should produce new events that would be used to infer models of Sut. At this point, it should be possible to reuse the passive testing technique described in this thesis. Nevertheless, the main unanswered question is how to replay the data in a safe manner? Such an approach also implies that the initial conditions are exactly the same between the system under analysis and the system under test. This is yet another strong assumption we would prefer not to make.

Instead, we would like to leverage our inferred models to extract input test data, for instance, by mining *realistic domains* as defined in [End+11]. Realistic domains are data domains found in concrete implementations. A data domain should come with a practical way to generate values in it. This would be particularly useful to actively interact with a system as it would ease the process of generating test data by means of a sampler for instance, *i.e.* a value generator.

Mining realistic domains for testing purpose is not the only use case in which we believe. Indeed, our inferred models own a lot of interesting information related to the behaviors of a production system running in production, and it could be interesting to apply data mining techniques on them.

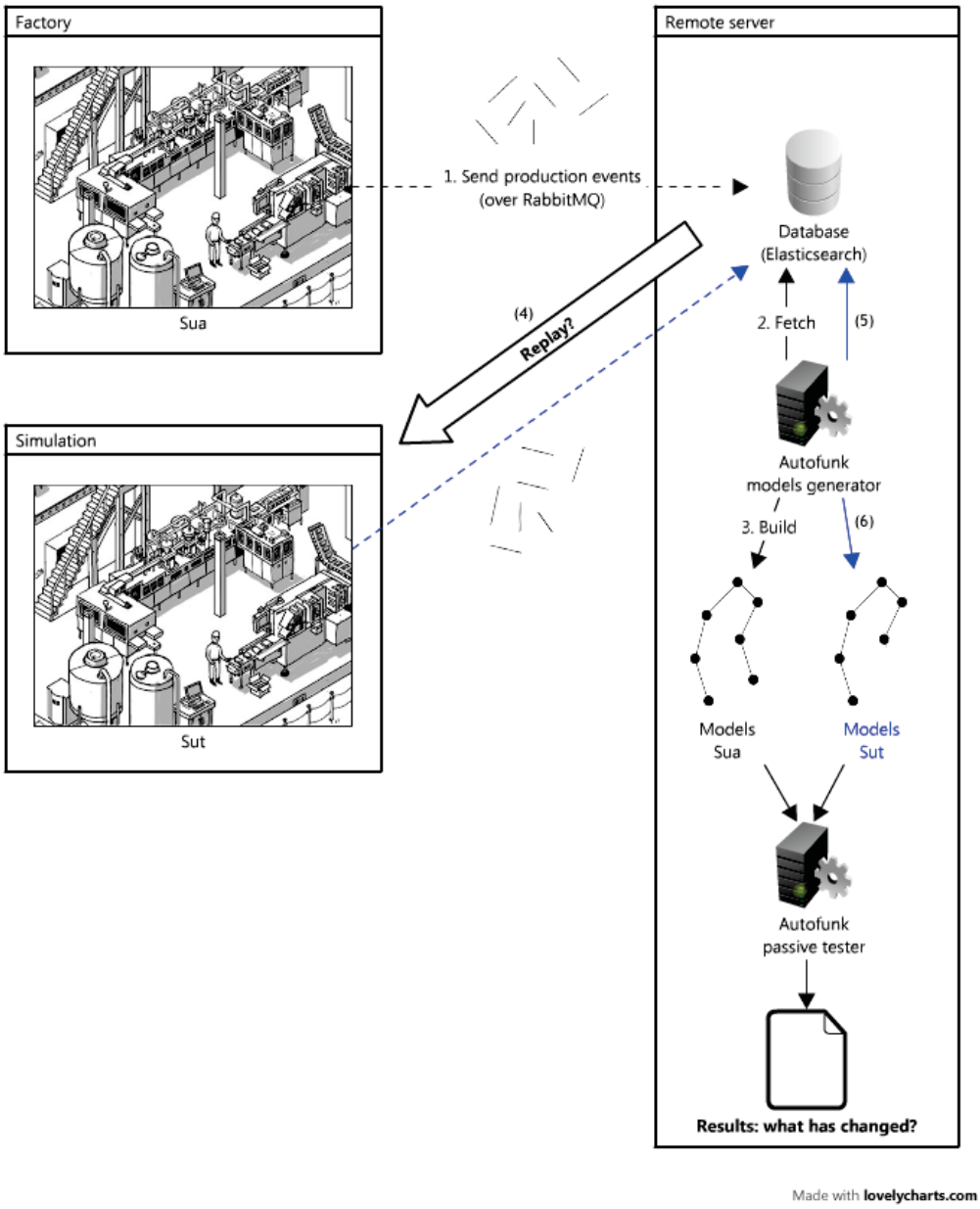

Fig. 6.1: Insight of an approach discussed with Michelin to use a replay technique with *Autofunk* in order to track what has changed between two versions of a production system.

6.3.4 Data mining

Data mining [Cha+06] is an interdisciplinary research domain whose goal is to extract information from a data set. For example, visualization, which we already mentioned in a previous section, is a component of data mining. In our case, given the collected trace sets and/or the inferred models, we have a lot of information available for data mining.

As an example, a side project we quickly set up with Michelin engineers was to visualize the data collected by *Autofunk*. We relied on a tool called *Kibana*[3] to show different business metrics, such as the usage rates of some stores in a workshop, the number of manufactured products per day, but also the usage rates of the production machines themselves as depicted in Figure 6.2. Such information could be used to create models that might predict maintenance operations for instance. The main objective of *predictive maintenance* [Mob02] is to decide when to maintain a system according to its state, which we could deduce by mining the data contained in the inferred models.

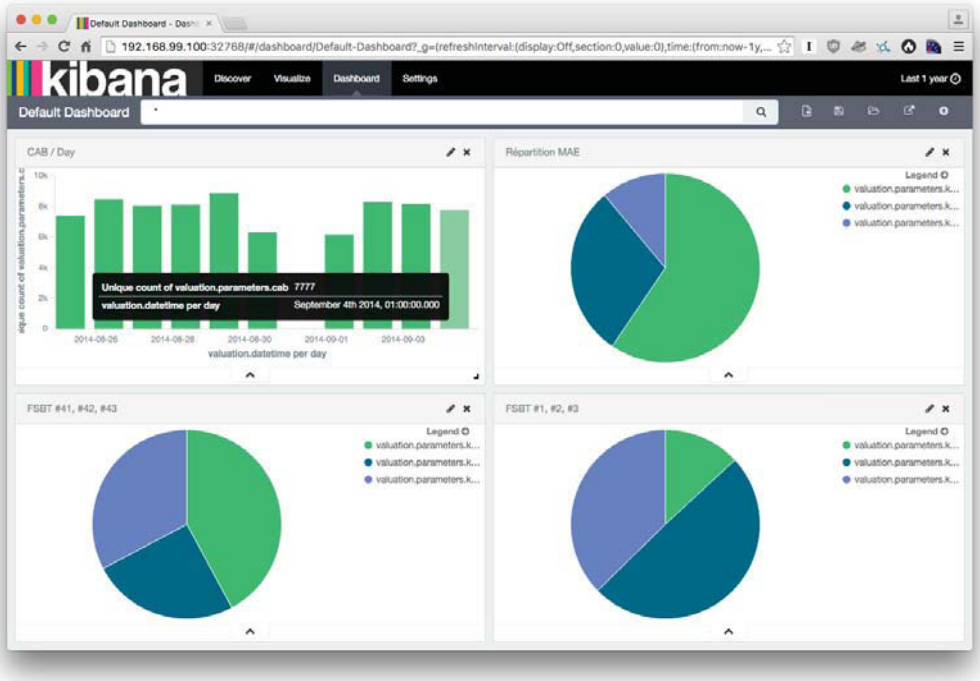

Fig. 6.2: Dashboard displaying various business metrics created with Kibana, a visualization tool.

We could also extract time data from the models. A potential use case would be to detect slowness in the production lines. That way, we might highlight imperceptible abnormal functioning slowing down the whole manufacturing process. This is

[3] `https://www.elastic.co/products/kibana`

somehow related to the *workload-based performance testing* approach introduced by Avritzer *et al.* [Avr+02], and more generally, to the *Performance Testing* [VW98] and *Knowledge Management* [PS15] fields.

6.3.5 Refuting our main hypothesis

As stated in the introduction of this thesis (cf. Chapter 1 • Section 1.2 (page 3)), we consider a system under analysis as a system which behaves correctly. In other words, such a system does not produce any fault. It is not entirely unrealistic since this assumption has been validated with our industrial partner Michelin. In fact, Michelin's production systems run continuously with only a few scheduled downtimes, *i.e.* periods when either the whole factory or only a workshop is unavailable, *e.g.*, for maintenance. Otherwise systems are fully operational.

That being said, their need for a reliable method to perform upgrades, which led to the work presented in this thesis, demonstrates that such systems are not error-proof. Putting it differently, inferring models representing behaviors of a software under analysis from production data is compelling, but it comes at a price: it is likely that *Autofunk* will infer erroneous behaviors due to a fault that happened in a production environment, which will not be revealed by our testing module. It is not an issue when performing, for instance, robustness testing, but it should not be used for conformance testing. We were able to perform conformance testing only because of Michelin's conditions, which we could extend to most of the existing industrial and manufacturing contexts. Nevertheless, it cannot be applied in all cases.

Based on this state, we would like to reject such a hypothesis to perform conformance testing based on our inferred models, but also to make the models more accurate. We already highlighted a few paths to improve the accuracy of the inferred models in Section 6.2.1. To go a step further, we could apply *model checking* [BK+08] if we consider our inferred models as the system models. Model checking is a verification technique that explores all possible system states, described in a *system model*, in a brute-force manner thanks to a model checker. It is useful to show whether a given system model truly satisfies a certain property. Nonetheless, such properties have to be provided, and we hit a known issue again: the lack of up-to-date documentation and/or specification of legacy systems.

6.4 Final thoughts

In this thesis, we proposed a solution to a practical problem that led to two main research directions, each opening the door to many different fields as well as many new challenges. The collaboration with Michelin has been successful, yet we believe

that working with other industrial companies would be a huge benefit, and moving toward a consortium would be interesting for both Academia and Industry people.

Over the last three years, we met several researchers who were working on legacy systems. Such systems may be outdated or simply old, given how software are deeply established in our lives, computer scientists are just at the beginning of the research realm on these systems.

Bibliography

[Abd+06] Parosh Aziz Abdulla, Lisa Kaati, and Johanna Hogberg. *Bisimulation Minimization of Tree Automata*. Tech. rep. In Proc. 11th Int. Conf. Implementation and Application of Automata, volume 4094 of LNCS, 2006 (cit. on p. 59).

[Abr87] Samson Abramsky. "Observation Equivalence As a Testing Equivalence". In: *Theoretical Computer Science* 53.2–3 (1987), pp. 225 –241 (cit. on p. 23).

[AD94] Rajeev Alur and David L. Dill. "A Theory of Timed Automata". In: *Theoretical Computer Science* 126 (1994), pp. 183–235 (cit. on p. 16).

[Alc+04] Baptiste Alcalde, Ana Cavalli, Dongluo Chen, Davy Khuu, and David Lee. "Network Protocol System Passive Testing for Fault Management: A Backward Checking Approach". In: *Formal Techniques for Networked and Distributed Systems–FORTE 2004*. Springer, 2004, pp. 150–166 (cit. on p. 26).

[Alu+05] Rajeev Alur, Pavol Černý, P. Madhusudan, and Wonhong Nam. "Synthesis of Interface Specifications for Java Classes". In: *SIGPLAN Not.* 40.1 (Jan. 2005), pp. 98–109 (cit. on pp. 42, 130).

[Ama+08] D. Amalfitano, A.R. Fasolino, and P. Tramontana. "Reverse Engineering Finite State Machines from Rich Internet Applications". In: *Reverse Engineering, 2008. WCRE '08. 15th Working Conference On*. 2008, pp. 69–73 (cit. on pp. 33, 35, 37).

[Ama+11] D. Amalfitano, A.R. Fasolino, and P. Tramontana. "A GUI Crawling-Based Technique for Android Mobile Application Testing". In: *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference On*. 2011, pp. 252–261 (cit. on pp. 33–35, 37).

[Ama+12] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. "Using GUI Ripping for Automated Testing of Android Applications". In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ASE 2012. New York, NY, USA: ACM, 2012, pp. 258–261 (cit. on pp. 34, 35, 37, 71).

[Ama+14] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Bryan Dzung Ta, and Atif M. Memon. "MobiGUITAR – A Tool for Automated Model-Based Testing of Mobile Apps". In: *IEEE Software* (2014) (cit. on pp. 27, 33–35, 37).

[Amm+02] Glenn Ammons, Rastislav Bodík, and James R. Larus. "Mining Specifications". In: *SIGPLAN Not.* 37.1 (Jan. 2002), pp. 4–16 (cit. on pp. 27, 41, 130).

[AN13] Tanzirul Azim and Iulian Neamtiu. “Targeted and Depth-First Exploration for Systematic Testing of Android Apps”. In: *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*. OOPSLA ’13. Indianapolis, Indiana, USA: ACM, 2013, pp. 641–660 (cit. on pp. 34, 37, 130).

[Ana+12] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. “Automated Concolic Testing of Smartphone Apps”. In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. FSE ’12. New York, NY, USA: ACM, 2012, 59:1–59:11 (cit. on p. 71).

[And+12] César Andrés, Mercedes G Merayo, and Manuel Núnez. “Formal Passive Testing of Timed Systems: Theory and Tools”. In: *Software Testing, Verification and Reliability* 22.6 (2012), pp. 365–405 (cit. on p. 25).

[Ang81] Dana Angluin. “A Note on the Number of Queries Needed to Identify Regular Languages”. In: *Information and Control* 51.1 (1981), pp. 76 –87 (cit. on p. 31).

[Ang87] Dana Angluin. “Learning Regular Sets from Queries and Counterexamples”. In: *Information and Computation* 75.2 (1987), pp. 87 –106 (cit. on pp. 27, 28, 30).

[Ant+11] J. Antunes, N. Neves, and P. Verissimo. “Reverse Engineering of Protocols from Network Traces”. In: *Reverse Engineering (WCRE), 2011 18th Working Conference On*. 2011, pp. 169–178 (cit. on p. 27).

[Avr+02] Alberto Avritzer, Joe Kondek, Danielle Liu, and Elaine J Weyuker. “Software Performance Testing Based on Workload Characterization”. In: *Proceedings of the 3rd International Workshop on Software and Performance*. ACM. 2002, pp. 17–24 (cit. on p. 141).

[Bar+11] Mike Barnett, Manuel Fahndrich, K. Rustan M. Leino, et al. “Specification and Verification: The Spec# Experience”. In: *Communications of the ACM* 54.6 (2011), pp. 81–91 (cit. on p. 16).

[BD04] P. Bourque and R. Dupuis. “Guide to the Software Engineering Body of Knowledge 2004 Version”. In: *Guide to the Software Engineering Body of Knowledge, 2004. SWEBOK* (2004) (cit. on p. 12).

[Beo+15] Harsh Beohar, Mahsa Varshosaz, and Mohammad Reza Mousavi. “Basic Behavioral Models for Software Product Lines: Expressiveness and Testing Pre-Orders”. In: *Science of Computer Programming* (2015), pp. – (cit. on p. 23).

[Ber+04] Antonia Bertolino, Andrea Polini, Paola Inverardi, and Henry Muccini. “Towards Anti-Model-Based Testing”. In: *Proc. DSN 2004 (Ext. abstract)* (2004), pp. 124–125 (cit. on p. 2).

[Ber+06] Therese Berg, Bengt Jonsson, and Harald Raffelt. “Regular Inference for State Machines with Parameters”. English. In: *Fundamental Approaches to Software Engineering*. Ed. by Luciano Baresi and Reiko Heckel. Vol. 3922. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, pp. 107–121 (cit. on p. 30).

[Ber+08] Therese Berg, Bengt Jonsson, and Harald Raffelt. “Regular Inference for State Machines Using Domains with Equality Tests”. English. In: *Fundamental Approaches to Software Engineering*. Ed. by JosÃ©Luiz Fiadeiro and Paola Inverardi. Vol. 4961. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, pp. 317–331 (cit. on p. 30).

[Ber+09] Antonia Bertolino, Paola Inverardi, Patrizio Pelliccione, and Massimo Tivoli. “Automatic Synthesis of Behavior Protocols for Composable Web-Services”. In: *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ESEC/FSE ’09. New York, NY, USA: ACM, 2009, pp. 141–150 (cit. on pp. 27, 43).

[Ber06] Therese Berg. “Regular inference for reactive systems”. In: (2006) (cit. on p. 29).

[Ber07] Antonia Bertolino. “Software Testing Research: Achievements, Challenges, Dreams”. In: *Future of Software Engineering, 2007. FOSE ’07*. 2007, pp. 85–103 (cit. on p. 1).

[Ber91] Gilles Bernot. “Testing Against Formal Specifications: A Theoretical View”. In: *Proceedings of the International Joint Conference on Theory and Practice of Software Development on Advances in Distributed Computing (ADC) and Colloquium on Combining Paradigms for Software Development (CCPSD): Vol. 2*. TAPSOFT ’91. Brighton, United Kingdom: Springer-Verlag New York, Inc., 1991, pp. 99–119 (cit. on p. 23).

[BF72] A.W. Biermann and J.A. Feldman. “On the Synthesis of Finite-State Machines from Samples of Their Behavior”. In: *Computers, IEEE Transactions on* C-21.6 (1972), pp. 592–597 (cit. on pp. 38, 128).

[BK+08] Christel Baier, Joost-Pieter Katoen, et al. *Principles of Model Checking*. MIT press Cambridge, 2008 (cit. on p. 141).

[BM83] D.L. Bird and C.U. Munoz. “Automatic Generation of Random Self-Checking Test Cases”. In: *IBM Systems Journal* 22.3 (1983), pp. 229–245 (cit. on p. 14).

[Boh79] Barry W. Bohem. “Software Engineering; R & D Trends and Defense Needs”. In: *Research Directions in Software Technology (Ch. 22)*. Ed. by P. Wegner. Cambridge, MA: MIT Press, 1979, pp. 1–9 (cit. on p. 10).

[Boo91] G. Booch. *Object Oriented Design: With Applications*. The Benjamin/Cummings Series in Ada and Software Engineering. Benjamin/Cummings Pub., 1991 (cit. on p. 1).

[BR70] J. N. Buxton and B. Randell, eds. *Software Engineering Techniques: Report of a Conference Sponsored by the NATO Science Committee, Rome, Italy, 27-31 Oct. 1969, Brussels, Scientific Affairs Division, NATO*. 1970 (cit. on p. 10).

[Bri89] Ed Brinksma. “Formal Approach to Conformance Testing”. In: *Proc. Int. Workshop on Protocol Test Systems*. North-Holland. 1989, pp. 311–325 (cit. on p. 22).

[Cav+03] Ana Cavalli, Caroline Gervy, and Svetlana Prokopenko. “New approaches for passive testing using an extended finite state machine specification”. In: *Information and Software Technology* 45.12 (2003), pp. 837–852 (cit. on p. 26).

[Cav+09a] A Cavalli, Azzedine Benameur, Wissam Mallouli, and Keqin Li. “A Passive Testing Approach for Security Checking and Its Pratical Usage for Web Services Monitoring”. In: *NOTERE 2009* (2009) (cit. on p. 25).

[Cav+09b] Ana Cavalli, Stephane Maag, and Edgardo Montes de Oca. “A Passive Conformance Testing Approach for a MANET Routing Protocol”. In: *Proceedings of the 2009 ACM Symposium on Applied Computing*. SAC ’09. New York, NY, USA: ACM, 2009, pp. 207–211 (cit. on p. 25).

[Cav+15] Ana R Cavalli, Teruo Higashino, and Manuel Núñez. “A survey on formal active and passive testing with applications to the cloud”. In: *annals of telecommunications-annales des télécommunications* 70.3-4 (2015), pp. 85–93 (cit. on p. 26).

[Cha+06] Soumen Chakrabarti, Martin Ester, Usama Fayyad, et al. “Data Mining Curriculum: A Proposal (Version 1.0)”. In: *Intensive Working Group of ACM SIGKDD Curriculum Committee* (2006) (cit. on p. 140).

[Cho+13] Wontae Choi, George Necula, and Koushik Sen. “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 and Applications*. OOPSLA ’13. New York, NY, USA: ACM, 2013, pp. 623–640 (cit. on pp. 35, 37).

[Cla+99] Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999 (cit. on p. 10).

[CM13] Xiaoping Che and Stephane Maag. “Passive testing on performance requirements of network protocols”. In: *Advanced Information Networking and Applications Workshops (WAINA), 2013 27th International Conference on*. IEEE. 2013, pp. 1439–1444 (cit. on p. 26).

[Com+10] David Combe, Colin De La Higuera, and Jean-Christophe Janodet. “Zulu: An Interactive Learning Competition”. In: *Finite-State Methods and Natural Language Processing*. Springer, 2010, pp. 139–146 (cit. on p. 31).

[Cot+07] Domenico Cotroneo, Roberto Pietrantuono, Leonardo Mariani, and Fabrizio Pastore. “Investigation of Failure Causes in Workload-Driven Reliability Testing”. In: *Fourth International Workshop on Software Quality Assurance: In Conjunction with the 6th ESEC/FSE Joint Meeting*. ACM. 2007, pp. 78–85 (cit. on p. 39).

[Cup+05] Frederic Cuppens, Nora Cuppens-Boulahia, and Thierry Sans. “Nomad: A Security Model with Non Atomic Actions and Deadlines”. In: *Computer Security Foundations, 2005. CSFW-18 2005. 18th IEEE Workshop*. IEEE. 2005, pp. 186–196 (cit. on p. 25).

[Dal+10] Valentin Dallmeier, Nikolai Knopp, Christoph Mallon, Sebastian Hack, and Andreas Zeller. “Generating Test Cases for Specification Mining”. In: *Proceedings of the 19th International Symposium on Software Testing and Analysis*. ISSTA ’10. New York, NY, USA: ACM, 2010, pp. 85–96 (cit. on p. 42).

[Dal+12] Valentin Dallmeier, Martin Burger, Tobias Orth, and Andreas Zeller. “WebMate: A Tool for Testing Web 2.0 Applications”. In: *Proceedings of the Workshop on JavaScript Tools*. JSTools ’12. New York, NY, USA: ACM, 2012, pp. 11–15 (cit. on p. 37).

[Deb+02] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. “A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II”. In: *Evolutionary Computation, IEEE Transactions on* 6.2 (2002), pp. 182–197 (cit. on p. 129).

[DG08] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters”. In: *Communications of the ACM* 51.1 (2008), pp. 107–113 (cit. on p. 131).

[DN81] Joe W. Duran and Simeon Ntafos. “A Report on Random Testing”. In: *Proceedings of the 5th International Conference on Software Engineering*. ICSE ’81. Piscataway, NJ, USA: IEEE Press, 1981, pp. 179–183 (cit. on p. 14).

[DNH84] Rocco De Nicola and Matthew CB Hennessy. “Testing Equivalences for Processes”. In: *Theoretical computer science* 34.1 (1984), pp. 83–133 (cit. on pp. 5, 23, 111, 113).

[DR99] Joseph S Dumas and Janice Redish. *A Practical Guide to Usability Testing*. Intellect Books, 1999 (cit. on p. 11).

[DS14a] William Durand and Sébastien Salva. “Inférence De Modeles Dirigée Par La Logique Métier”. In: *Actes de la 13eme édition d’AFADL, atelier francophone sur les Approches Formelles dans l’Assistance au Développement de Logiciels, juin 2014.* (2014), p. 31 (cit. on pp. 5, 6, 48).

[DS14b] William Durand and Sébastien Salva. “Inferring Models with Rule-Based Expert Systems”. In: *Proceedings of the Fifth Symposium on Information and Communication Technology, SoICT ’14, Hanoi, Vietnam, December 4-5, 2014*. 2014, pp. 92–101 (cit. on pp. 5, 6, 48).

[DS15a] William Durand and Sébastien Salva. “Autofunk: An Inference-Based Formal Model Generation Framework for Production Systems”. In: *FM 2015: Formal Methods - 20th International Symposium, Oslo, Norway, June 24-26, 2015, Proceedings*. 2015, pp. 577–580 (cit. on pp. 5, 6, 81).

[DS15b] William Durand and Sebastien Salva. “Passive Testing of Production Systems Based on Model Inference”. In: *Formal Methods and Models for Codesign (MEMOCODE), 2015 ACM/IEEE International Conference On*. 2015, pp. 138–147 (cit. on pp. 5, 7, 111).

[Dup96] Pierre Dupont. “Incremental Regular Inference”. In: *Proceedings of the Third ICGI-96*. Springer, 1996, pp. 222–237 (cit. on p. 31).

[End+11] Ivan Enderlin, Frédéric Dadeau, Alain Giorgetti, and Abdallah Ben Othman. “Praspel: A Specification Language for Contract-Based Testing in PHP”. In: *Proceedings of the 23rd IFIP WG 6.1 International Conference on Testing Software and Systems*. ICTSS’11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 64–79 (cit. on pp. 16, 138).

[Ern+07] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, et al. “the Daikon System for Dynamic Detection of Likely Invariants”. In: *Science of Computer Programming* 69.1–3 (2007). Special issue on Experimental Software and Toolkits, pp. 35 –45 (cit. on p. 40).

[Ern+99] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. “Dynamically Discovering Likely Program Invariants to Support Program Evolution”. In: *Proceedings of the 21st International Conference on Software Engineering*. ICSE ’99. New York, NY, USA: ACM, 1999, pp. 213–224 (cit. on p. 39).

[Fer89] Jean-Claude Fernandez. “An Implementation of an Efficient Algorithm for Bisimulation Equivalence”. In: *Science of Computer Programming* 13 (1989), pp. 13–219 (cit. on pp. 23, 59).

[Fit12] Melvin Fitting. *First-order logic and automated theorem proving*. Springer Science & Business Media, 2012 (cit. on p. 10).

[FR14] R. Fielding and J. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*. RFC 7230. `http://www.rfc-editor.org/rfc/rfc7230.txt`. RFC Editor, 2014 (cit. on pp. 48, 54).

[Fra+05] L. Frantzen, J. Tretmans, and T.A.C. Willemse. “Test Generation Based on Symbolic Specifications”. In: *FATES 2004*. Ed. by J. Grabowski and B. Nielsen. Lecture Notes in Computer Science 3395. Springer, 2005, pp. 1–15 (cit. on pp. 18–20, 56).

[Fra+06] L. Frantzen, J. Tretmans, and T. A. C. Willemse. “A Symbolic Framework for Model-Based Testing”. In: *Proceedings of the First Combined International Conference on Formal Approaches to Software Testing and Runtime Verification*. FATES’06/RV’06. Seattle, WA: Springer-Verlag, 2006, pp. 40–54 (cit. on p. 56).

[Ghe+09] Carlo Ghezzi, Andrea Mocci, and Mattia Monga. “Synthesizing Intensional Behavior Models by Graph Transformation”. In: *Proceedings of the 31st International Conference on Software Engineering*. ICSE ’09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 430–440 (cit. on p. 40).

[God+08] Patrice Godefroid, Michael Y. Levin, and David Molnar. “Automated Whitebox Fuzz Testing”. In: *In NDSS*. 2008 (cit. on p. 14).

[Gol67] E Mark Gold. “Language Identification in the Limit”. In: *Information and control* 10.5 (1967), pp. 447–474 (cit. on p. 27).

[Gra+01] Todd L Graves, Mary Jean Harrold, Jung-Min Kim, Adam Porter, and Gregg Rothermel. “An Empirical Study of Regression Test Selection Techniques”. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 10.2 (2001), pp. 184–208 (cit. on p. 13).

[Gro+08] Roland Groz, Keqin Li, Alexandre Petrenko, and Muzammil Shahbaz. “Modular system verification by inference, testing and reachability analysis”. In: *Testing of Software and Communicating Systems*. Springer, 2008, pp. 216–233 (cit. on p. 27).

[Gro+12] Roland Groz, Muhammad-Naeem Irfan, and Catherine Oriat. “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”. In: ed. by Tiziana Margaria and Bernhard Steffen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. Chap. Algorithmic Improvements on Regular Inference of Software Models and Perspectives for Security Testing, pp. 444–457 (cit. on p. 27).

[HA04] Victoria J. Hodge and Jim Austin. “A Survey of Outlier Detection Methodologies”. In: *Artificial Intelligence Review* 22 (2 2004), pp. 85–126 (cit. on p. 105).

[Hal+91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. “The synchronous dataflow programming language Lustre”. In: *Proceedings of the IEEE* 79.9 (1991), pp. 1305–1320 (cit. on p. 16).

[Ham77] R.G. Hamlet. “Testing Programs with the Aid of a Compiler”. In: *Software Engineering, IEEE Transactions on* SE-3.4 (1977), pp. 279–290 (cit. on p. 14).

[HL95] Matthew Hennessy and Huimin Lin. “Symbolic Bisimulations”. In: *Theoretical Computer Science* 138.2 (1995), pp. 353–389 (cit. on p. 18).

[HM80] Matthew Hennessy and Robin Milner. *On Observing Nondeterminism and Concurrency*. Springer, 1980 (cit. on p. 23).

[Hor51] Alfred Horn. “On Sentences Which Are True of Direct Unions of Algebras”. In: *The Journal of Symbolic Logic* 16 (01 Mar. 1951), pp. 14–21 (cit. on p. 55).

[How+12a] Falk Howar, Bernhard Steffen, Bengt Jonsson, and Sofia Cassel. “Inferring Canonical Register Automata”. English. In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Viktor Kuncak and Andrey Rybalchenko. Vol. 7148. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 251–266 (cit. on p. 30).

[How+12b] Falk Howar, Malte Isberner, Bernhard Steffen, Oliver Bauer, and Bengt Jonsson. “Inferring Semantic Interfaces of Data Structures”. English. In: *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 7609. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 554–571 (cit. on p. 30).

[HP13] Wen-ling Huang and Jan Peleska. “Exhaustive Model-Based Equivalence Class Testing”. English. In: *Testing Software and Systems*. Ed. by Hüsnü Yenigün, Cemal Yilmaz, and Andreas Ulrich. Vol. 8254. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 49–64 (cit. on p. 14).

[HR04] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning About Systems*. Cambridge University Press, 2004 (cit. on p. 18).

[HT97] Lex Heerink and Jan Tretmans. “Refusal Testing for Classes of Transition Systems with Inputs and Outputs”. In: *Formal Description Techniques and Protocol Specification, Testing and Verification*. Springer, 1997, pp. 23–39 (cit. on p. 23).

[Hu+14] Han Hu, Yonggang Wen, Tat-Seng Chua, and Xuelong Li. “Toward Scalable Systems for Big Data Analytics: A Technology Tutorial”. In: *Access, IEEE* 2 (2014), pp. 652–687 (cit. on p. 130).

[Hun+02] Hardi Hungar, Tiziana Margaria, and Bernhard Steffen. “Model Generation for Legacy Systems”. In: *Radical Innovations of Software and Systems Engineering in the Future, 9th International Workshop, RISSEF 2002, Venice, Italy, October 7-11, 2002, Revised Papers*. 2002, pp. 167–183 (cit. on pp. 34, 37).

[Hun+04] Hardi Hungar, Tiziana Margaria, and Bernhard Steffen. “Model Generation for Legacy Systems”. English. In: *Radical Innovations of Software and Systems Engineering in the Future*. Ed. by Martin Wirsing, Alexander Knapp, and Simonetta Balsamo. Vol. 2941. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, pp. 167–183 (cit. on p. 40).

[Irf+12] Muhammad-Naeem Irfan, Roland Groz, and Catherine Oriat. “Improving Model Inference of Black Box Components Having Large Input Test Set.” In: *ICGI*. Citeseer. 2012, pp. 133–138 (cit. on p. 30).

[JAH79] M. A. Wong J. A. Hartigan. “Algorithm AS 136: A K-Means Clustering Algorithm”. In: *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 28.1 (1979), pp. 100–108 (cit. on p. 105).

[JM12] Mona Erfani Joorabchi and Ali Mesbah. “Reverse Engineering IOS Mobile Applications”. In: *Proceedings of the 2012 19th Working Conference on Reverse Engineering*. WCRE ’12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 177–186 (cit. on p. 37).

[Jon+02] James A Jones, Mary Jean Harrold, and John Stasko. “Visualization of Test Information to Assist Fault Localization”. In: *Proceedings of the 24th International Conference on Software Engineering*. ACM. 2002, pp. 467–477 (cit. on p. 132).

[Jor95] Paul C. Jorgensen. *Software Testing: A Craftsman’s Approach*. 1st. Boca Raton, FL, USA: CRC Press, Inc., 1995 (cit. on pp. 2, 14, 15).

[Jér06] Thierry Jéron. “Model-Based Test Selection for Infinite State Reactive Systems”. In: *From Model-Driven Design to Resource Management for Distributed Embedded Systems*. Springer, 2006, pp. 35–44 (cit. on p. 21).

[Kan03] Cem Kaner. “What is a good test case”. In: *Star East* (2003), p. 16 (cit. on p. 1).

[KF94] Michael Kaminski and Nissim Francez. “Finite-Memory Automata”. In: *Theoretical Computer Science* 134.2 (1994), pp. 329 –363 (cit. on p. 30).

[Kim+07] Jangbok Kim, Kyunghee Choi, D.M. Hoffman, and Gihyun Jung. “White Box Pairwise Test Case Generation”. In: *Quality Software, 2007. QSIC ’07. Seventh International Conference On*. 2007, pp. 286–291 (cit. on p. 14).

[KK12] Mohd Ehmer Khan and Farmeena Khan. “A Comparative Study of White Box, Black Box and Grey Box Testing Techniques”. In: *Editorial Preface* 3.6 (2012) (cit. on p. 13).

[Krk+10] Ivo Krka, Yuriy Brun, Daniel Popescu, Joshua Garcia, and Nenad Medvidovic. “Using Dynamic Execution Traces and Program Invariants to Enhance Behavioral Model Inference”. In: *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*. ICSE ’10. New York, NY, USA: ACM, 2010, pp. 179–182 (cit. on pp. 27, 40).

[LA+00] Tal Lev-Ami, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. “Putting static analysis to work for verification: A case study”. In: *ACM SIGSOFT Software Engineering Notes*. Vol. 25. 5. ACM. 2000, pp. 26–38 (cit. on p. 10).

[Lak09] Kiran Lakhotia. “Search–Based Testing”. PhD thesis. King’s College London, 2009 (cit. on p. 1).

[Lan+98] Kevin J. Lang, Barak A. Pearlmutter, and Rodney A. Price. “Results of the Abbadingo One DFa Learning Competition and a New Evidence-Driven State Merging Algorithm”. In: *Proceedings of the 4th International Colloquium on Grammatical Inference*. ICGI ’98. London, UK, UK: Springer-Verlag, 1998, pp. 1–12 (cit. on p. 40).

[Lan96] Kevin Lano. *The B Language and Method: A Guide to Practical Formal Development*. 1st. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1996 (cit. on p. 16).

[Lea+99] GaryT. Leavens, AlbertL. Baker, and Clyde Ruby. “JML: A Notation for Detailed Design”. English. In: *Behavioral Specifications of Businesses and Systems*. Ed. by Haim Kilov, Bernhard Rumpe, and Ian Simmonds. Vol. 523. The Springer International Series in Engineering and Computer Science. Springer US, 1999, pp. 175–188 (cit. on p. 16).

[Lee+06] D. Lee, Dongluo Chen, Ruibing Hao, et al. “Network Protocol System Monitoring-a Formal Approach with Passive Testing”. In: *Networking, IEEE/ACM Transactions on* 14.2 (2006), pp. 424–437 (cit. on p. 25).

[LK06] D. Lo and Siau-Cheng Khoo. “QUARK: Empirical Assessment of Automaton-Based Specification Miners”. In: *Reverse Engineering, 2006. WCRE ’06. 13th Working Conference On*. 2006, pp. 51–60 (cit. on p. 38).

[Lo+09] David Lo, Leonardo Mariani, and Mauro Pezzè. “Automatic Steering of Behavioral Model Inference”. In: *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ESEC/FSE ’09. New York, NY, USA: ACM, 2009, pp. 345–354 (cit. on p. 38).

[Lo+12] David Lo, Leonardo Mariani, and Mauro Santoro. “Learning Extended {FSA} from Software: An Empirical Assessment”. In: *Journal of Systems and Software* 85.9 (2012). Selected papers from the 2011 Joint Working IEEE/IFIP Conference on Software Architecture (WICSA 2011), pp. 2063 –2076 (cit. on pp. 39, 129).

[Lor+08] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. “Automatic Generation of Software Behavioral Models”. In: *Proceedings of the 30th International Conference on Software Engineering*. ICSE ’08. New York, NY, USA: ACM, 2008, pp. 501–510 (cit. on pp. 27, 38).

[LS+14] Pablo Lamela Seijas, Simon Thompson, Ramsay Taylor, Kirill Bogdanov, and John Derrick. “Synapse: Automatic Behaviour Inference and Implementation Comparison for Erlang”. In: *Proceedings of the Thirteenth ACM SIGPLAN Workshop on Erlang*. Erlang ’14. New York, NY, USA: ACM, 2014, pp. 73–74 (cit. on p. 41).

[LS09] Martin Leucker and Christian Schallhart. “A Brief Account of Runtime Verification”. In: *The Journal of Logic and Algebraic Programming* 78.5 (2009). The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS’07), pp. 293 –303 (cit. on pp. 10, 25).

[LS11] Thierry Le Sergent. “SCADE: A Comprehensive Framework for Critical System and Software Engineering”. In: *Proceedings of the 15th International Conference on Integrating System and Software Modeling*. SDL’11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 2–3 (cit. on p. 16).

[LW89] Hareton KN Leung and Lee White. “Insights into Regression Testing [software Testing]”. In: *Software Maintenance, 1989., Proceedings., Conference On*. IEEE. 1989, pp. 60–69 (cit. on p. 11).

[MA01] R.E. Miller and K.A. Arisha. “Fault Management Using Passive Testing for Mobile IPv6 Networks”. In: *Global Telecommunications Conference, 2001. GLOBECOM ’01. IEEE*. Vol. 3. 2001, 1923–1927 vol.3 (cit. on p. 26).

[Mal+08] W. Mallouli, F. Bessayah, A. Cavalli, and A. Benameur. “Security Rules Specification and Analysis Based on Passive Testing”. In: *Global Telecommunications Conference, 2008. IEEE GLOBECOM 2008. IEEE*. 2008, pp. 1–6 (cit. on p. 26).

[Mea55] George H. Mealy. “A Method for Synthesizing Sequential Circuits”. In: *Bell System Technical Journal, The* 34.5 (1955), pp. 1045–1079 (cit. on p. 29).

[Mei04] Karl Meinke. “Automated Black-Box Testing of Functional Correctness Using Function Approximation”. In: *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA ’04. New York, NY, USA: ACM, 2004, pp. 143–153 (cit. on p. 32).

[Mei10] Karl Meinke. “CGE: A Sequential Learning Algorithm for Mealy Automata”. English. In: *Grammatical Inference: Theoretical Results and Applications*. Ed. by JoséM. Sempere and Pedro García. Vol. 6339. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 148–162 (cit. on p. 32).

[Mem+03] Atif Memon, Ishan Banerjee, and Adithya Nagarajan. “GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing”. In: *Proceedings of the 10th Working Conference on Reverse Engineering*. WCRE ’03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 260– (cit. on p. 71).

[Mer+11] Maik Merten, Bernhard Steffen, Falk Howar, and Tiziana Margaria. “Next Generation LearnLib”. English. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by ParoshAziz Abdulla and K.RustanM. Leino. Vol. 6605. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, pp. 220–223 (cit. on pp. 30, 130).

[Mes+12] Ali Mesbah, Arie van Deursen, and Stefan Lenselink. “Crawling Ajax-Based Web Applications Through Dynamic Analysis of User Interface State Changes”. In: *ACM Transactions on the Web (TWEB)* 6.1 (2012), 3:1–3:30 (cit. on pp. 33–35, 37, 71).

[Mey92] Bertrand Meyer. “Applying "Design by Contract"”. In: *Computer* 25.10 (Oct. 1992), pp. 40–51 (cit. on p. 16).

[MG00] Christoph Michael and Anup Ghosh. “Using Finite Automata to Mine Execution Data for Intrusion Detection: A Preliminary Report”. English. In: *Recent Advances in Intrusion Detection*. Ed. by Hervé Debar, Ludovic Mé, and S.Felix Wu. Vol. 1907. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2000, pp. 66–79 (cit. on p. 27).

[MH07] Mike Lewicki Martial Hebert. *Artificial Intelligence, Logic and Reasoning*. Lecture. Available from https://www.cs.cmu.edu/afs/cs/academic/class/15381-s07/www/slides/022707reasoning.pdf. 2007 (cit. on p. 55).

[Mil80] Robin Milner. *A Calculus for Communicating Processes, Volume 92 of Lecture Notes in Computer Science*. 1980 (cit. on pp. 17, 23).

[Mil89] Robin Milner. *Communication and Concurrency*. Vol. 84. Prentice hall New York etc., 1989 (cit. on p. 23).

[MM10] Sonali Mathur and Shaily Malik. "Advancements in the V-Model". In: *International Journal of Computer Applications* 1.12 (2010) (cit. on p. 15).

[Mob02] R Keith Mobley. *An Introduction to Predictive Maintenance*. Butterworth-Heinemann, 2002 (cit. on p. 140).

[Moo56] Edward F. Moore. "Gedanken Experiments on Sequential Machines". In: *Automata Studies*. Princeton U., 1956, pp. 129–153 (cit. on p. 2).

[Mor+10] G. Morales, S. Maag, A. Cavalli, et al. "Timed Extended Invariants for the Passive Testing of Web Services". In: *Web Services (ICWS), 2010 IEEE International Conference On*. 2010, pp. 592–599 (cit. on p. 25).

[MP07] Leonardo Mariani and Mauro Pezze. "Dynamic Detection of COTS Component Incompatibility". In: *IEEE Software* 24.5 (2007), pp. 76–85 (cit. on p. 38).

[MP08] L. Mariani and F. Pastore. "Automated Identification of Failure Causes in System Logs". In: *Software Reliability Engineering, 2008. ISSRE 2008. 19th International Symposium On*. 2008, pp. 117–126 (cit. on pp. 27, 39).

[MS11] Karl Meinke and MuddassarA. Sindhu. "Incremental Learning-Based Testing for Reactive Systems". English. In: *Tests and Proofs*. Ed. by Martin Gogolla and Burkhart Wolff. Vol. 6706. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, pp. 134–151 (cit. on pp. 27, 32).

[Mye79] Glenford J. Myers. *Art of Software Testing*. New York, NY, USA: John Wiley & Sons, Inc., 1979 (cit. on pp. 1, 10).

[Ngu+13] BaoN. Nguyen, Bryan Robbins, Ishan Banerjee, and Atif Memon. "GUITAR: An Innovative Tool for Automated Testing of GUI-Driven Software". English. In: *Automated Software Engineering* (2013), pp. 1–41 (cit. on pp. 33–35, 37).

[Nie03] Oliver Niese. "An Integrated Approach to Testing Complex Systems". PhD thesis. Dortmund University of Technology, 2003 (cit. on p. 29).

[OK05] Alessandro Orso and Bryan Kennedy. "Selective Capture and Replay of Program Executions". In: *SIGSOFT Softw. Eng. Notes* 30.4 (May 2005), pp. 1–7 (cit. on p. 138).

[Ors+04] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. "Scaling Regression Testing to Large Software Systems". In: *ACM SIGSOFT Software Engineering Notes*. Vol. 29. 6. ACM. 2004, pp. 241–251 (cit. on p. 132).

[Par+98] Rajesh Parekh, Codrin Nichitiu, and Vasant Honavar. "A Polynomial Time Incremental Algorithm for Learning DFA". English. In: *Grammatical Inference*. Ed. by Vasant Honavar and Giora Slutzki. Vol. 1433. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1998, pp. 37–49 (cit. on p. 31).

[Par81] David Park. "Concurrency and Automata on Infinite Sequences". In: *Proceedings of the 5th GI-Conference on Theoretical Computer Science*. London, UK, UK: Springer-Verlag, 1981, pp. 167–183 (cit. on pp. 47, 59).

[PG09] Michael Pradel and Thomas R. Gross. “Automatic Generation of Object Usage Specifications from Large Method Traces”. In: *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. ASE ’09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 371–382 (cit. on pp. 27, 42, 129).

[Phi86] Iain Phillips. “Refusal Testing”. In: *Automata, Languages and Programming, 13th International Colloquium, ICALP86, Rennes, France, July 15-19, 1986, Proceedings*. 1986, pp. 304–313 (cit. on p. 23).

[Phi87] Iain Phillips. “Refusal Testing”. In: *Theoretical Computer Science* 50.3 (1987), pp. 241–284 (cit. on p. 23).

[PS15] Stella Pachidi and Marco Spruit. “The Performance Mining Method: Extracting Performance Knowledge from Software Operation Data”. In: *International Journal of Business Intelligence Research (IJBIR)* 6.1 (2015), pp. 11–29 (cit. on p. 141).

[PY06] Alexandre Petrenko and Nina Yevtushenko. “Conformance Tests As Checking Experiments for Partial Nondeterministic FSM”. English. In: *Formal Approaches to Software Testing*. Ed. by Wolfgang Grieskamp and Carsten Weise. Vol. 3997. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, pp. 118–133 (cit. on pp. 5, 59, 60, 85, 88, 89, 91, 92, 95, 106).

[Raf+05] Harald Raffelt, Bernhard Steffen, and Therese Berg. “LearnLib: A Library for Automata Learning and Experimentation”. In: *Proceedings of the 10th International Workshop on Formal Methods for Industrial Critical Systems*. FMICS ’05. New York, NY, USA: ACM, 2005, pp. 62–71 (cit. on pp. 30, 130).

[Ram03] Muthu Ramachandran. “Testing Software Components Using Boundary Value Analysis”. In: *Proceedings of the 29th Conference on EUROMICRO*. EUROMICRO ’03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 94– (cit. on p. 14).

[RH97] Gregg Rothermel and Mary Jean Harrold. “A Safe, Efficient Regression Test Selection Technique”. In: *ACM Trans. Softw. Eng. Methodol.* 6.2 (Apr. 1997), pp. 173–210 (cit. on p. 13).

[RK04] J.A. Rehg and H.W. Kraebber. *Computer-Integrated Manufacturing*. Pearson Prentice Hall, 2004 (cit. on p. 81).

[Roo86] Paul Rook. “Controlling Software Projects”. In: *Software Engineering Journal* 1.1 (1986), p. 7 (cit. on p. 15).

[RR01] S.P. Reiss and M. Renieris. “Encoding Program Executions”. In: *Software Engineering, 2001. ICSE 2001. Proceedings of the 23rd International Conference On*. 2001, pp. 221–230 (cit. on p. 38).

[Rus+00] Vlad Rusu, Lydie du Bousquet, and Thierry Jéron. “An Approach to Symbolic Test Generation”. In: *Proceedings of the Second International Conference on Integrated Formal Methods*. IFM ’00. London, UK, UK: Springer-Verlag, 2000, pp. 338–357 (cit. on pp. 21, 49).

[Rus+05] Vlad Rusu, Hervé Marchand, and Thierry Jéron. “Automatic Verification and Conformance Testing for Validating Safety Properties of Reactive Systems”. In: *FM 2005: Formal Methods*. Springer, 2005, pp. 189–204 (cit. on p. 56).

[Sal+05] Maher Salah, Trip Denton, Spiros Mancoridis, and Ali Shokouf. “Scenariographer: A Tool for Reverse Engineering Class Usage Scenarios from Method Invocation Sequences”. In: *In ICSM*. IEEE Computer Society, 2005, pp. 155–164 (cit. on pp. 27, 42).

[SD15] Sébastien Salva and William Durand. “Autofunk, a Fast and Scalable Framework for Building Formal Models from Production Systems”. In: *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems, DEBS ’15, Oslo, Norway, June 29 - July 3, 2015*. 2015, pp. 193–204 (cit. on pp. 5, 6, 81).

[Seb02] Fabrizio Sebastiani. “Machine Learning in Automated Text Categorization”. In: *ACM Comput. Surv.* 34.1 (Mar. 2002), pp. 1–47 (cit. on p. 132).

[Set09] Burr Settles. *Active Learning Literature Survey*. Computer Sciences Technical Report 1648. University of Wisconsin–Madison, 2009 (cit. on p. 29).

[Sho+07] Sharon Shoham, Eran Yahav, Stephen Fink, and Marco Pistoia. “Static Specification Mining Using Automata-Based Abstractions”. In: *Proceedings of the 2007 International Symposium on Software Testing and Analysis*. ISSTA ’07. New York, NY, USA: ACM, 2007, pp. 174–184 (cit. on p. 43).

[SL15] Sébastien Salva and Patrice Laurençot. “Model Inference and Automatic Testing of Mobile Applications”. In: *International Journal On Advances in Software*. Vol. 8. 1,2. Iaria, June 2015 (cit. on pp. 33–37).

[SM12] Muddassar A. Sindhu and Karl Meinke. “IDS: An Incremental Learning Algorithm for Finite Automata”. In: *CoRR* abs/1206.2691 (2012) (cit. on p. 32).

[SS97] Ian Sommerville and Pete Sawyer. *Requirements Engineering: A Good Practice Guide*. 1st. New York, NY, USA: John Wiley & Sons, Inc., 1997 (cit. on p. 17).

[Ste+11] Bernhard Steffen, Falk Howar, and Maik Merten. “Introduction to Active Automata Learning from a Practical Perspective”. English. In: *Formal Methods for Eternal Networked Software Systems*. Ed. by Marco Bernardo and Valérie Issarny. Vol. 6659. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, pp. 256–296 (cit. on p. 29).

[Tan+95] Q. M. Tan, A. Petrenko, G. v Bochmann, and G. Luo. *Testing Trace Equivalence for Labeled Transition Systems*. Université de Montréal, Département d’informatique et de recherche opérationnelle, 1995 (cit. on p. 23).

[TH00] Henrik Thane and Hans Hansson. “Using Deterministic Replay for Debugging of Distributed Real-Time Systems”. In: *Real-Time Systems, 2000. Euromicro RTS 2000. 12th Euromicro Conference On*. IEEE. 2000, pp. 265–272 (cit. on p. 138).

[TH08] Nikolai Tillmann and Jonathan de Halleux. “Pex - White Box Test Generation for .NET”. In: *Proc. of Tests and Proofs (TAP’08)*. Vol. 4966. LNCS. Prato, Italy: Springer Verlag, 2008, 134–153 (cit. on p. 14).

[Tho15] Johannes Thones. “Microservices”. In: *Software, IEEE* 32.1 (2015), pp. 116–116 (cit. on p. 130).

[TL98] Kuo chung Tai and Yu Lei. “A Test Generation Strategy for Pairwise Testing”. In: *IEEE Transactions on Software Engineering* 28 (1998), p. 2002 (cit. on p. 14).

[Ton+12] Paolo Tonella, Alessandro Marchetto, Cu Duy Nguyen, et al. “Finding the Optimal Balance Between over and Under Approximation of Models Inferred from Execution Logs”. In: *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference On*. IEEE. 2012, pp. 21–30 (cit. on p. 129).

[Ton+13] Paolo Tonella, Cu Duy Nguyen, Alessandro Marchetto, Kiran Lakhotia, and Mark Harman. “Automated Generation of State Abstraction Functions Using Data Invariant Inference”. In: *8th International Workshop on Automation of Software Test, AST 2013, San Francisco, CA, USA, May 18-19, 2013*. 2013, pp. 75–81 (cit. on p. 41).

[TR03] Mary Frances Theofanos and Janice (Ginny) Redish. “Bridging the Gap: Between Accessibility and Usability”. In: *interactions* 10.6 (Nov. 2003), pp. 36–51 (cit. on p. 11).

[Tre08] Jan Tretmans. “Formal Methods and Testing”. In: ed. by Robert M. Hierons, Jonathan P. Bowen, and Mark Harman. Berlin, Heidelberg: Springer-Verlag, 2008. Chap. Model Based Testing with Labelled Transition Systems, pp. 1–38 (cit. on pp. 17, 23).

[Tre92] Gerrit Jan Tretmans. “A Formal Approach to Conformance Testing”. In: (1992) (cit. on p. 22).

[Tre94] Jan Tretmans. “A Formal Approach to Conformance Testing”. In: *Proceedings of the IFIP TC6/WG6.1 Sixth International Workshop on Protocol Test Systems VI*. Amsterdam, The Netherlands, The Netherlands: North-Holland Publishing Co., 1994, pp. 257–276 (cit. on p. 22).

[Tre96a] Jan Tretmans. “Conformance Testing with Labelled Transition Systems: Implementation Relations and Test Generation”. In: *Computer networks and ISDN systems* 29.1 (1996), pp. 49–79 (cit. on pp. 23, 24).

[Tre96b] Jan Tretmans. “Test Generation with Inputs, Outputs, and Quiescence”. In: (1996), pp. 127–146 (cit. on pp. 17, 23).

[Ura+07] Hasan Ural, Zhi Xu, and Fan Zhang. “An Improved Approach to Passive Testing of FSM-Based Systems”. In: *Proceedings of the Second International Workshop on Automation of Software Test*. AST ’07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 6– (cit. on p. 26).

[Vaa91] Frits W Vaandrager. “On the Relationship Between Process Algebra and Input/output Automata”. In: *Logic in Computer Science, 1991. LICS’91., Proceedings of Sixth Annual IEEE Symposium On*. IEEE. 1991, pp. 387–398 (cit. on pp. 23, 113).

[VW98] Filippos I Vokolos and Elaine J Weyuker. “Performance Testing of Software Systems”. In: *Proceedings of the 1st International Workshop on Software and Performance*. ACM. 1998, pp. 80–87 (cit. on p. 141).

[Wal+07] Neil Walkinshaw, Kirill Bogdanov, Mike Holcombe, and Sarah Salahuddin. “Reverse Engineering State Machines by Interactive Grammar Inference”. In: *In Proceedings of the 14th Working Conference on Reverse Engineering (WCRE’07*. IEEE, 2007 (cit. on p. 40).

[Wal+10] Neil Walkinshaw, Kirill Bogdanov, John Derrick, and Javier Paris. “Increasing Functional Coverage by Inductive Testing: A Case Study”. In: *Proceedings of the 22Nd IFIP WG 6.1 International Conference on Testing Software and Systems*. ICTSS’10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 126–141 (cit. on p. 14).

[Wal+95] Gwendolyn H. Walton, J. H. Poore, and Carmen J. Trammell. “Statistical Testing of Software Based on a Usage Model”. In: *Softw. Pract. Exper.* 25.1 (Jan. 1995), pp. 97–108 (cit. on p. 14).

[Was+07] Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. “Detecting Object Usage Anomalies”. In: *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ESEC-FSE ’07. New York, NY, USA: ACM, 2007, pp. 35–44 (cit. on p. 43).

[WD10] W Eric Wong and Vidroha Debroy. “Software Fault Localization.” In: *Encyclopedia of Software Engineering* 1 (2010), pp. 1147–1156 (cit. on p. 132).

[Wey82] Elaine J Weyuker. “On testing non-testable programs”. In: *The Computer Journal* 25.4 (1982), pp. 465–470 (cit. on p. 1).

[WF89] Dolores R Wallace and Roger U Fujii. “Software Verification and Validation: An Overview”. In: *IEEE Software* 3 (1989), pp. 10–17 (cit. on p. 10).

[Wha+02] John Whaley, Michael C. Martin, and Monica S. Lam. “Automatic Extraction of Object-Oriented Component Interfaces”. In: *SIGSOFT Softw. Eng. Notes* 27.4 (July 2002), pp. 218–228 (cit. on pp. 41, 42).

[WK99] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999 (cit. on p. 16).

[Won+97] W Eric Wong, Joseph R Horgan, Saul London, and Hira Agrawal. “A Study of Effective Regression Testing in Practice”. In: *Software Reliability Engineering, 1997. Proceedings., the Eighth International Symposium On*. IEEE. 1997, pp. 264–274 (cit. on p. 11).

[Yan+06] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. “Perracotta: Mining Temporal API Rules from Imperfect Traces”. In: *Proceedings of the 28th International Conference on Software Engineering*. ICSE ’06. New York, NY, USA: ACM, 2006, pp. 282–291 (cit. on pp. 42, 129).

[Yan+13] Wei Yang, Mukul R. Prasad, and Tao Xie. “A Grey-Box Approach for Automated GUI-Model Generation of Mobile Applications”. In: *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering*. FASE’13. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 250–265 (cit. on pp. 33–35, 37, 71, 130).

[Zho+11] Hao Zhong, Lu Zhang, Tao Xie, and Hong Mei. “Inferring Specifications for Resources from Natural Language API Documentation”. In: *Autom. Softw. Eng.* 18.3-4 (2011), pp. 227–261 (cit. on pp. 27, 44).

[IEE90] IEEE. “IEEE Standard Glossary of Software Engineering Terminology”. In: *IEEE Std 610.12-1990* (1990), pp. 1–84 (cit. on p. 12).

[ISO01] ISO. *ISO/IEC 9126-1:2001, Software Engineering – Product Quality – Part 1: Quality Model*. Tech. rep. International Organization for Standardization, 2001 (cit. on pp. 11, 12).

[ISO05] E.N. ISO. "9000: 2005". In: *Quality management systems-Fundamentals and vocabulary (ISO 9000: 2005)* (2005) (cit. on p. 1).

[ISO10a] ISO/IEC. *ISO/IEC 25010 - Systems and Software Engineering - Systems and Software Quality Requirements and Evaluation (SQuaRE) - System and Software Quality Models*. Tech. rep. 2010 (cit. on pp. 11, 12).

[ISO10b] ISO/IEC/IEEE. "Systems and Software Engineering – Vocabulary". In: *ISO/IEC/IEEE 24765:2010(E)* (2010), pp. 1–418 (cit. on pp. 10, 12, 13, 15).

Colophon

This thesis was typeset with LaTeX 2$_\varepsilon$. It uses the *Clean Thesis* style developed by Ricardo Langner. The design of the *Clean Thesis* style is inspired by user guide documents from Apple Inc.

Download the *Clean Thesis* style at `http://cleanthesis.der-ric.de/`.