La collana della Scuola di Alta Formazione Dottorale accoglie le migliori tesi di dottorato dell'Università degli Studi di Bergamo, insignite della dignità di stampa e sottoposte a procedura di *blind peer review*.

Software testing is an important phase in the software development process, aiming at locating faults in artifacts, in order to achieve a degree of confidence that the software behaves according to its specification. The goal of this thesis is to study how model-based testing can be applied to repair models of configurable software systems.
The thesis presents the application cases of configuration constraints in combinatorial models, feature models, and clock guards in timed automata. Empirical evaluation shows that, on average, the devised processes achieve around 37% accuracy in combinatorial model repair, 79% in XSS vulnerability detection, 89% in feature model repair through mutations, and 100% accuracy on variability constraints and timed automata clock-guards repair.

Following the master's degree in computer engineering at the University of Bergamo, **Marco Radavelli** obtained his PhD in Engineering and Applied Sciences in the 2018-2019 Academic Year (XXXII cycle) at the same university, supervised by Prof. Angelo Gargantini. During his university studies he carried out research experiences in research institutions and universities abroad, including the Harvard Graduate School of Design, the University of Texas at Arlington, the National Institute of Informatics in Tokyo, and the SBA Research in Vienna. After earning his PhD, he currently lives in Hamburg (Germany), where he works as a consultant in the field of digitization and cloud computing.

ISBN: 978-88-97413-54-7
DOI: 10.13122/978-88-97413-54-7

Marco Radavelli

SOFTWARE TESTING TO REPAIR MODELS

35

Collana della Scuola di Alta Formazione Dottorale
- 35 -

Marco Radavelli

USING MODEL-BASED TESTING TO REPAIR MODELS Of Configurable Software Systems

UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

2021

Collana della Scuola di Alta Formazione Dottorale

- 35 -

Collana della Scuola di Alta Formazione Dottorale

Diretta da Paolo Cesaretti

Ogni volume è sottoposto a *blind peer review*.

ISSN: 2611-9927

Sito web: https://aisberg.unibg.it/handle/10446/130100

Marco Radavelli

USING MODEL-BASED TESTING TO REPAIR MODELS
# of Configurable Software Systems

Università degli Studi di Bergamo

2021

Using Model-Based Testing to Repair Models
of Configurable Software Systems / Marco Radavelli.
– Bergamo : Università degli Studi di Bergamo, 2021.
(Collana della Scuola di Alta Formazione Dottorale; 35)

ISBN: 978-88-97413-54-7
DOI: 10.13122/978-88-97413-54-7

Progetto grafico: Servizi Editoriali – Università degli Studi di Bergamo
© 2018 Università degli Studi di Bergamo
via Salvecchio, 19
24129 Bergamo
Cod. Fiscale 80004350163
P. IVA 01612800167

https://aisberg.unibg.it/handle/10446/200443

Acknowledgments

I would like to thank my supervisor Prof. Angelo Gargantini, for his patient guidance, support and help during my PhD. I also wish to thank Prof. Paolo Arcaini for his patient support and guidance, and for inviting me to work with him at ERATO Metamathematics for Systems Design Project, at NII in Tokyo. I thank also Prof. Ichiro Hasuo for accepting my visit and for his suggestions and kindness. Special thanks go to my family for the endless support, and to all the co-authors and colleagues at university, in particular Justyna Petke, Paolo Vavassori, Silvia Bonfanti, and Andrea Bombarda, for their contribution throughout my PhD, not only scientifically, but also on a human perspective. A special thank I wish to give to Yu Lei, his family and his team at University of Texas at Arlington for the encouragement and the support in research activities. Thanks to Dimitris Simos and his team at SBA Research in Vienna, for his guidance and openness in collaborating with me.

Thanks also to the co-supervisor of my M.Sc. thesis, Prof. Sarah Nadi, for having solicited in me a curiosity for the topics in software product lines, and giving me many clues on how to approach research.

Finally, I wish to thank the many kind people (researchers and not) with whom I had the pleasure to interact during the *journey* to produce this thesis.

I would like to thank the University of Bergamo for making it possible.

Table of Contents

Introduction

Traditional testing techniques provide test generation, fault localization and program repair. If we assume a *fault* to be a discrepancy between a model (*problem space*) and the system implementation (*solution space*) for a particular test case, we can distinguish two scenarios: (1) the model is correct and the implementation is faulty, and (2) the implementation is correct and the model is faulty. While the most common case is that the model reflects the specification, and the error resides in the system implementation, this is not always the case. The specification may change throughout the software system life cycle, and it may happen that the implementation is updated but the model(s) remain(s) outdated, leading to inconsistencies that reduce system maintainability [176]. Such inconsistencies are often due to budget constraints, as updating models is a difficult and error-prone activity to perform manually. In other cases, the model is derived from the software implementation, often even long time after the program is put in operation. Moreover, with the increase of complexity of software systems, also their models are becoming larger: the Linux Kernel, for instance, has currently more than 13,000 features (in release 3.9) [164]. The goal of this thesis is to study how model-based testing techniques can be applied to drive automatic repair of software models. Model *repair* can be useful whenever the model is outdated w.r.t. the implementation, or a defined oracle or the specification. Sometimes an engineer also wants to detect which is the *correct* model of a current aspect of an implemented system, and can use testing techniques not only to find bugs in a system, but also to *repair* or to entirely *learn* a model from the system implementation. Outcomes of this project may have an impact in reducing the effort of software maintenance, which is one of the costliest phases in software life cycle [217].

Chapter 1. Research Objective

1.1 Research Questions and Objective

The goal of this thesis is to study how model-based software testing, and in particular *combinatorial interaction testing* (CIT), can be applied to drive automatic repair of models of software variability. The thesis is divided into seven chapters. Each chapter (introduction excluded) tries to answer a research question (RQ).

RQ1: *What are the main techniques and applications of software testing and program repair?*

This is shown in Chapter 2 in which relevant books and papers are analyzed. Different types of software models are presented (namely software product lines and combinatorial models, with a reference to timed automata), and currently available techniques to support engineers in automatically updating and repairing programs and models are analyzed. In particular, techniques such as mutation, together with evolutionary algorithms, are discussed. In this chapter is also given a background on existing ways to encode and reason on configuration space, such as SAT and SMT solvers, and binary decision diagrams.

RQ2: *How to increase software engineer productivity by automatically repair non-conformant models?*

Often during software development lifecycle, models may become non-conformant with respect to the system implementation. Automatically repair the model in order to maintain conformance, by applying small edits to it, is the goal of the approach presented in Chapter 3. The devised process uses information coming from tests on the real system, and it represents the *framework* shared by all the *specific* methods described in the following chapters. Chapter 3 answers this research question, and discusses how also academic tools can improve engineers perform the different phases of this model repair process.

RQ3: *How to repair configuration constraints using software testing?*

Part of the goal of the thesis is the application of model repair technique directly to *configuration constraints*, i.e., constraints under which (and only under which) the system is supposed to work correctly, or shows a certain property. Chapther 4 describes a possible approach to automatically repair constraints using combinatorial interaction testing, and a case study on the application of this method to software security, for the detection of XSS vulnerabilities in web applications.

RQ4: *How to repair feature models?*

A widely used notation to model software variability is *feature model*. Feature models are used in software product lines (SPL) engineering. Many software products have configuration files with parameter-value pairs, and such configuration files can be, for example, modeled by feature models. Although feature models are not yet widely used to model configurable software applications, there are many companies, in particular in the automotive sector, that use feature models to represent domain knowledge for SPLs (see this list [71] of some companies using feature models). Model repair is a challenging problem as the number of configurations grows exponentially with the number of modeled features (i.e., parameters), and some software have hundreds or even thousands of features: the Linux Kernel, for instance, has currently over 18 000 features [198]. Chapter 5 presents two approaches for automated repair of feature models: the first is targeted towards repairing the hierarchical relations among features and the simple constraints of type *requires* and *excludes*, while the second is targeted towards repairing arbitrary constraints among features.

RQ5: *How to repair timed automata clock guards?*

Many software systems have a behavior that involves timed constraints, and can be modeled by timed automata (TA), and parametric timed automata (PTA) [21, 13, 22]. They can range from electrical automated machines such as coffee machines and washing machines (where time automata model, e.g., the maximum amount of seconds after which, if the user has not selected any type of coffee, the inserted coins have to be given back to the user), to protocol implementation in the railway, automotive and aerospace industry. Chapter 6 describes a process to automatically repair this particular types of model, i.e., timed automata, by generating (from that initial model) and executing test cases (i.e., sequences in the automata), and iteratively repair the potential inconsistencies between the model and the system.

RQ6: *How combinatorial testing editing and generation can be made easy to perform?*

As part of the thesis, we have also the goal to investigate a tool that supports the activity of combinatorial test generation, that is often used in the presented processes for model repair. Chapter 7 presents a comparative study among the existing tools for combinatorial testing, and proposes a SaaS (Software-as-a-Service) web-based tool, that does not require any installation, to make the test generation activity easy for engineers.

RQ7: *How can the detection of failure-inducing combinations be improved?*

One of the steps of the test-driven processes to automatically repair models is the detection of the combinations of parameters that cause inconsistencies between the model and the system: the failure-inducing combinations (*fccs*). There are tools for failure-inducing combination detection, such as BEN [96, 97], IterAIFL [208], AIFL [184], FIC [219], SOFOT [155], and ICT [159]. In Chapter 8 we present MixTgTe, a new method to detect failure-inducing combinations guaranteeing to achieve complete accuracy up to a certain strength t. This tool could be used not only to improve the constraint repair method devised (described in Chapter 4), but also as an alternative fault localization technique.

1.2 Contributions

We give below an overview of the contributions of this PhD thesis in terms of published or accepted peer-reviewed papers, in international conferences or scientific journals. The list is in chronological order, and for each paper is shown a short summary and a description of the role of M. Radavelli in the work.

A. [88] Angelo Gargantini, Justyna Petke, Marco Radavelli, and Paolo Vavassori. Validation of constraints among configuration parameters using search-based combinatorial interaction testing. In *Search Based Software Engineering - 8th International Symposium, SSBSE 2016, Raleigh, NC, USA, October 8-10, 2016, Proceedings*, pages 49–63, 2016

   In this paper, we study the effectiveness of different combinatorial test generation criteria in detecting faults in system whose combinatorial model contains constraints among parameters. M. Radavelli contributed in the discussion of ideas between co-authors, in implementing part of the experiments (in particular, the evaluation with the benchmark *Django*), and in writing about it. The co-authors further contributed in writing the paper, implementing the approach, in writing the code to report data from evaluation, and in further discussions and proof reading.

B. [87] Angelo Gargantini, Justyna Petke, and Marco Radavelli. Combinatorial Interaction Testing for Automated Constraint Repair. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 239–248. IEEE, March 2017

   In this paper, we present a process for finding and fixing conformance faults in models of parameter configuration of software systems. We compare effectiveness among the CIT policies introduced in the previous paper [88] in repairing the constraints, and also in completely inferring the constraints, for the benchmark *Django*. M. Radavelli contributed to the discussion of ideas between co-authors, to the implementation of part of the evaluation, and in writing the paper.

The co-authors further contributed in writing the paper, implementing the approach, reporting the data from evaluation, and in further discussions and proof reading.

C. [26] Paolo Arcaini, Angelo Gargantini, and Marco Radavelli. An evolutionary process for product-driven updates of feature models. In *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems*, VAMOS 2018, pages 67–74, New York, NY, USA, 2018. ACM, ACM

This paper presents an approach to automatically repair feature models. Given a set of change requirements in terms of features and products to add/remove from a feature model, the evolutionary process tries to mutate that feature model such that the obtained model exactly captures the specified requirements. M. Radavelli contributed in the process definition, implemented the process in Java, selected the benchmarks, performed the experiments on them, and produced tables and charts. M. Radavelli contributed in writing the paper. The co-authors further contributed in discussions, writing the paper, and proof reading.

D. [89] Angelo Gargantini and Marco Radavelli. Migrating combinatorial interaction test modeling and generation to the web. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 308–317, 2018

This paper presents CTWEDGE, a novel tool that offers a complete SaaS (Software as a Service) environment for combinatorial testing modeling and generation. This tool is installation-free and download-free, easy to use, and extensible to support more generators. M. Radavelli contributed in the discussion towards this paper, in the tool implementation using Xtext Web [82, 211], in the comparison with existing tools, and in writing the paper. The co-author further contributed in discussions, writing the paper, and proof reading.

E. [27] Paolo Arcaini, Angelo Gargantini, and Marco Radavelli. Achieving change requirements of feature models by an evolutionary approach. *Journal of Systems and Software*, 150:64–76, 2019

This journal paper further extends [26] by improving the process and by performing a more extensive evaluation using real evolutions of feature models. M. Radavelli contributed in the discussions and in writing the paper (in particular the experiments and the related works), implemented the process and performed the experiments. The co-authors further contributed in discussions, writing the paper, and proof reading.

F. [28] Paolo Arcaini, Angelo Gargantini, and Marco Radavelli. Efficient and guaranteed detection of t-way failure-inducing combinations. In *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 200–209. IEEE, IEEE, April 2019

This paper presents MixTgTe, an iterative process to fault localization aiming at finding minimal failure-inducing combinations alternating combinatorial test generation and execution. With respect to existing methods, under the assumption that the maximum strength of the failure-inducing combinations is known and the process is run up to that strength, it guarantees to find all and only those minimal failure-inducing combinations. This kind of fault localization is used in the processes for model repair devised in our papers [87, 89, 93], and the method described in this paper could be used to improve those processes. M. Radavelli contributed in the discussions between co-authors, in writing the paper (all sections, although in particular the experiments and the related work), implemented the process, and performed the evaluation. The co-authors further contributed in discussions, writing the paper, and proof reading.

G. [93] Bernhard Garn, Marco Radavelli, Angelo Gargantini, Manuel Leithner, and Dimitris E. Simos. A fault-driven combinatorial process for model evolution in XSS vulnerability detection. In Franz Wotawa, Gerhard Friedrich, Ingo Pill, Roxane Koitz-Hristov, and Moonis Ali, editors, *Advances and Trends in Artificial Intelligence. From Theory to Practice*, pages 207–215, Cham, 2019. Springer International Publishing

This paper applies the automated iterative process for constraints repair described in [87] to the field of security testing. It identifies (from an empty model) constraints among XSS attack parameters that trigger XSS vulnerabilities in web applications. Empirical evaluation on six real-world web applications shows that the process achieves on average 78.8% accuracy in detecting XSS vulnerability triggering conditions. M. Radavelli contributed in the discussions among co-authors, in writing the paper (in particular the sections regarding the process description, the evaluation, and related work), and implemented part of the process.The co-authors further contributed in discussions, in implementing part of the process (web server for test case concretization, and XSS vulnerability detection), writing part of the paper, and proof reading.

H. [20] Étienne André, Paolo Arcaini, Angelo Gargantini, and Marco Radavelli. Repairing timed automata clock guards through abstraction and testing. In Dirk Beyer and Chantal Keller, editors, *Tests and Proofs*, pages 129–146, Cham, 2019. Springer International Publishing

This paper proposes an iterative approach for automatically repairing timed automata, in the case where clock guards shall be checked for conformance w.r.t. the system implementation, and be repaired if needed. Our approach abstracts the initial TA by adding some parameters in the clock guards, generates some tests, and then refines the abstraction by identifying only those TAs that correctly evaluate all the tests. M. Radavelli contributed in the discussions among co-authors, in the implementation of the approach in Java, and he performed the evaluation among the three benchmarks, as well as the comparison of the process with existing techniques. Co-authors contributed in discussions (especially precious has been André's expertise on Timed Automate), in implementing the process, in writing the paper and in proof reading.

I. [29] Paolo Arcaini, Angelo Gargantini, and Marco Radavelli. A process for fault-driven repair of constraints among features. In *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume B*, SPLC '19, pages 71:1–71:9, New York, NY, USA, 2019. ACM

   In this paper, we proposed a process that, given a (faulty or outdated) variability model, and the faults in terms of failure-containing combinations, identifies the constraints involved in the fault, repairs them according to the oracle value, and simplifies them to make the edit minimal. The process can be seen as an integration to the evolutionary algorithm we proposed to repair feature models in [26, 27], that deals with constraints among features expressed in propositional logic. M. Radavelli contributed in the discussions among co-authors, in the implementation of the approach in Java, performed the empirical evaluation among the 7 benchmark models and the different simplification strategies, and wrote the paper. Co-authors contributed in discussions, in writing the paper, and in proof reading.

1.2.1 Other publications

Aside the papers that characterize the contributions of this thesis, two extended abstracts that present the overall contents of this thesis have been accepted for presentation in two software engineering conferences (A and B in the list below). Furthermore, M. Radavelli contributed to a paper that we here only mention, as it goes beyond the scope of this thesis. It presents a framework for testing a medical protocol using ASM model refinement (paper C in the list below), and it is the fruit of a collaboration between the FMSE lab at the University of Bergamo [1], and the team of Prof. Yu Lei at University of Texas at Arlington.

[1] Formal Methods and Software Engineering Lab at University of Bergamo: `https://foselab.unibg.it/`

A. [172] M. Radavelli. Using testing to repair models. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 489–491, April 2019

B. [173] Marco Radavelli. Using software testing to repair models. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, pages 1253–1255, New York, NY, USA, 2019. ACM

C. [46] Silvia Bonfanti, Andrea Bombarda, Angelo Gargantini, Marco Radavelli, Feng Duan, and Yu Lei. Combining model refinement and test generation for conformance testing of the ieee phd protocol using abstract state machines. In *Testing Software and Systems - 31th IFIP WG International Conference, ICTSS 2019, Paris, France, October 15-17, 2018, Proceedings*, 2019

1.3 Note on conventions

Parts of this thesis are based on the aforementioned publications. To be able to easily distinguish a text included verbatim from the publications, the corresponding paragraphs have wider lateral margins and a reference is reported at the very beginning of the quoted text in the chapter.

> *[X]* This paragraph is an example of the way we mark the text, which is a verbatim copy. It means that the original text appeared in the paper [X].

Chapter 2. State of the Art

To make this thesis more self-contained, this chapter presents an overview of the context of model repair, i.e., model-driven software engineering, with some background over the specific software models targeted in the repair processes, i.e., combinatorial constraints, and feature models. (Sect. 2.1), and of the main common techniques used in the approaches for model repair proposed in this thesis, i.e., software testing and fault localization, evolutionary algorithms, SAT and SMT solvers, and BDD and MDD representations (Sect. 2.2).

More specific concepts, such as timed automata, and further definitions, are described in the corresponding chapters throughout the thesis.

2.1 The Role of Models in Software Engineering

MDSE (Model-Driven Software Engineering) or simply MDE (Model-Driven Engineering) provides a comprehensive vision for system development. MDSE can be defined as a methodology for applying the advantages of modeling to software engineering activities [49, 103]. MDSE seeks for solutions according to two dimensions: conceptualization (columns in the Fig. 1, taken from [49]) and implementation (rows in the figure).

Different levels to model reality are defined in the *conceptualization* dimension, while the *implementation* issue deals with the mapping of the models to some existing or future running systems. While the conceptualization distinguish a model from its meta-model (model of the model), the *implementation* consists of linking the modeling level with the realization level via a transformation (or *automation*) level, where the mappings from the modeling to the realization levels are put in place. For the application level, such mappings can be from model to artifacts (*code generation*, or *model-to-text*, M2T, transformation), or from artifacts to model (*model inference*, or T2M transformation). In the other levels, we can have model-to-model transformations (M2M).

There is a literature on how to design high quality models [123], however keeping all the different models conformant to specification and implementation while code-base evolves, is one of the main challenges in model-based software engineering.

There are different ways to reach this goal of conformance: methods to check model validity [163, 35], tools to detect and visualize, and in some cases automatically fix, inconsistent models [77, 80, 76], even inconsistency directly in requirements [79, 78], reliability analysis, are examples of such methods.

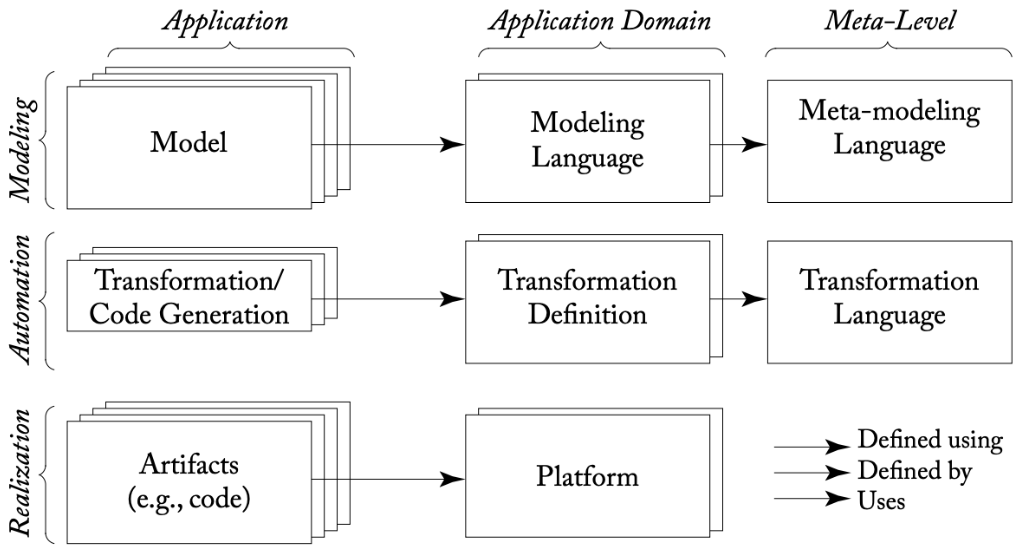


Figure 1: Conceptualization level and implementation level in Model-Driven Software Engineering

The model repair approaches proposed in this thesis try to improve the state of the art for model repair, by giving more tools or methods for software engineers to apply in the industrial practice, or to further extend.

Whereas with the recent advent of agile methodologies, such as the scrum development process, the usage of models is sometimes interpreted as being discouraged, empirical evidence shows that a trade-off on which models to keep, should be found [45]. As code evolves, models should be updated too to maintain conformance with code, and this operations take time: therefore which models to keep and which not, should be a careful decision to make [45]. Indeed, models have the positive effect of being more easily understandable than code and, among their functions, models are particularly important in helping engineers to transfer domain knowledge, and collaborate with others. A trade-off should be found, but models remain covering an important function in the software development process, therefore all the model-driven engineering activities are of importance.

In the rest of the section, we describe the three model types that we chose as target for the repair operations identified in this thesis: combinatorial constraints, feature models, and timed automata.

2.1.1 Combinatorial Models

In most configurable systems, dependencies exist between configuration parameters. Such constraints may be introduced for several reasons, e.g., to model inconsistencies between certain hardware components, limitations of the possible system configurations, or simply design choices [63]. Software systems can be configured by setting specific parameter values also at different stages of the software testing process.

- **Compile time** Configurations can be set at compile time. An example is shown in Figure 3b. Depending on the value settings of the Boolean variables `HELLO` and `BYE` different messages will be displayed when the program is run.

- **Design time** Configurations can also be set at design time. For example, in case of a SPL, a configurability model is built during the design.

- **Runtime** Another way of setting parameter configurations is at runtime. This can be usually done by means of a graphical user interface (GUI). In a chat client, e.g., you can change your availability status as the program is running.

- **Launch time** We also differentiate the case where parameters are read from a separate configuration file or given as program arguments, *before* the system is run. We say that these parameters are set at launch time of the given application. They decide which features of the system should be activated at startup. Examples of such systems include chat clients, web browsers and others.

A combinatorial model M is composed by a set P of parameters, and a set C of constraints among them. M can be formally defined as follows.

Definition 2.1. *Let $P = \{p_1, \ldots, p_m\}$ be the set of parameters. Every parameter p_i assumes values in the domain $D_i = \{v^i_1, \ldots, v^i_{o_i}\}$. Every parameter has its name (it can have also a type with its own name) and every enumerative value has an explicit name. We denote with $C = \{c_1, \ldots, c_n\}$ the set of constraints.*

Constraints c_i are given in general form, using the language of propositional logic with equality and arithmetic.

2.1.2 Feature Models

Software Product Line (SPL) engineering is an approach for the systematic reuse of software artifacts across a very large number of similar products. Various domains apply SPL engineering successfully to improve the overall project success, by increasing quality, saving costs for development and maintenance, and decreasing time-to-market [7]. SPLs offer a systematic reuse of software artifacts within a range of products sharing a common set of features (i.e., units of functionality) [102]. According to IEEE a feature is a distinguishing characteristic of a software item (e.g. concerning its performance, portability, or functionality) [6].

The central aspect of systematic reuse is the concept of variability. This concept provides the possibility to define particular artifacts (features) for the entire SPL as not necessarily being part of each product.

Variability specifies the point at which features are selected in combination with other features [66]. The stakeholders of the SPL generally define where variability occurs and decide which features are variable e.g. optional or alternate.

Feature models represent variability in software product lines, and are employed in various domains, in particular for cyber-physical systems, in the IoT and in the automotive sectors [110]. However, variability model consistency checking, and repair, in the context of systematic variability management in software, is still challenging and a relatively new problem [121]. One main reason for the increasing need of systematic variability management in software is the fact that the majority of modern features in a car are based on software. Thus, variability moves from mechanics and hardware to software [47, 160].

An example of feature model is given in Fig. 16, and, although there exist multiple, slightly different, formal definitions for feature and feature models, one is given in Section 3.4.2.

2.1.3 Model Repair and Program Repair

To the best of our knowledge, few studies address the problem of repairing the *model* artifacts, instead of the implementation code, and the problem of repair *automatically* existing models of software systems w.r.t. the implementation is still an open issue. Example of works on automated repair of programs are in [154, 132, 36].

In model-driven engineering, there are approaches aiming at repairing inconsistencies between models [144]; Tran et al. show a way to address the problem *manually* for the Linux Kernel [201]; Nadi et al. present a method to automatically *mine* conditions under which a system behaves in a certain way [153], and there are methods to statistically infer constraints from data [60, 8], but they are not directly applicable to repair existing models made of sets of constraints and they do not guarantee complete accuracy. A quality-based model refactoring framework assessing quality of merging operations among SPL models, expressed in UML [177], supports maintainability of models describing relations among features. It represents an approach to model repair, although it is specific to one kind of models. The need of a fault-driven constraint repair process was already envisioned in [107], but any empirical experiment were yet performed.

2.2 Software Engineering Techniques for Model Repair

Software systems are an integral part of life, from business applications (e.g., banking) to consumer products (e.g., cars). Most people have had an experience with software that did not work as expected. Software that does not work correctly can lead to many problems, including loss of money, time, or business reputation, and even injury or death. Software testing is a way to assess the quality of the software and to reduce the risk of software failure in operation.

Software testing is expensive and labor intensive. Software testing requires up to 50% of software development costs, and even more for safety-critical applications [18]. Therefore, software testing has been an active area both in research and in industry. There are several studies and techniques to adapt software testing to specific scenarios and applications, to improve test suite generation and execution, and to apply testing techniques to various tasks in software engineering, such as program repair and model inference and repair. [132, 60, 73, 212]

One of the goals of software testing is to automate as much as possible, thereby significantly reducing its cost, minimizing human error, and making regression testing easier. [18]

2.2.1 Combinatorial Interaction Testing

Experiments and industrial evidence suggest that software failures are usually caused by interactions among inputs or parameters of the system [124]. For this reason, combinatorial interaction testing (CIT), which consists in testing all the interactions of a given strength, is widely used and efficient in detecting bugs. Combinatorial Interaction Testing (CIT) is a particular type of model-based, black-box testing technique that aims at covering the interactions of parameters in a system under test (SUT), while some combinations may be forbidden by given constraints (forbidden tuples) [213]. It explores t-way feature interactions inside a given system, by effectively combining all t-tuples of parameter assignments in the smallest possible number of test cases. This allows to budget-constrain the costs of testing while still having a testing process driven by an effective and exhaustive coverage metric [62, 126]. The most commonly applied combinatorial testing technique is pairwise testing, which consists in applying the smallest possible test suite covering all pairs of input values (each pair in at least one test case). In fact, it has been experimentally shown that a test suite covering just all pairs of input values can already detect a significantly large part (typically 50% to 75%) of the faults in a program [68, 192]. Dunietz et al. [74] compared t-wise coverage to random input testing with respect to the percentage of structural (block) coverage achieved, showing that the former achieves better results if compared to random test suites of the same size. Burr and Young [55] reported 93% code coverage from applying pairwise testing of a large commercial software system, and many CIT tools (see [161] for an up to date listing) and techniques have already been developed [69, 101, 126] and are currently applied in practice [50, 124, 188]. CIT is used in a variety of applications for unit, system, and interoperability testing. It has generated both high-level test plans and detailed test cases.

In several applications, it greatly reduced the cost of test plan development. Testers can base their input on detailed development requirements or on a system's high-level functional requirements.

2.2.2 The Oracle Problem

In software testing, while test case generation, execution and fault localization have considerably advanced, the quality of the oracle remains an important bottleneck [111].

The effectiveness of testing depends both on the quality of the test cases and on the quality of the oracle [37, 191, 209]. The Oracle Problem is the problem of defining accurate oracles, capable of detecting all and only faulty behaviors exercised during testing [12], [18], [25], [36], [37], [38]. Without a (good) oracle to determine whether the test output is correct, test inputs that satisfy the strictest adequacy criteria remain useless and testing is ineffective. The quality of the oracle can be evaluated based on two properties: *completeness* (no false positives), and soundness (no false negatives: all faults should be rejected by the oracle).

With respect to our proposed model repair process, we assume the existence of an oracle to assess non-conformancies between the model and the system. In one case we automated the oracle (we used the implicit oracle), in other cases, the oracle could be set by the user or predicted. Oracle evaluation is often the longest phase in software testing, and a bottleneck in testing automation. Indeed, it is where domain knowledge expertise is needed, and often only human intervention is reliable, as the oracle is intimately related to domain knowledge representation, and to requirements engineering.

2.2.3 SAT and SMT solvers

In computer science, the Boolean satisfiability problem (abbreviated with *SAT*) is the problem of determining if there exists an interpretation that satisfies a given Boolean formula [43]. It asks whether the variables of a given Boolean formula can be consistently replaced by the values *true* ($\top$) or *false* ($\bot$) in such a way that the formula evaluates to *true*, i.e., the formula is *satisfiable*. If no such assignment exists, the formula is unsatisfiable.

For example, the formula $a \wedge \neg b$ is satisfiable because one can find the values $a = \top \wedge b = \bot$, which make $(a \wedge \neg b) = \top$. In contrast, $a \wedge \neg a$ is unsatisfiable.

The SAT problem is NP-complete, but heuristic SAT-algorithms are able to solve problem instances involving tens of thousands of variables and formulas consisting of millions of symbols, which is sufficient for many practical SAT problems from, e.g., artificial intelligence, circuit design, and automatic theorem proving [130, 44].

SAT and SMT (Satisfiability Modulo Theories) solvers are employed in various applications in software engineering, especially in model checking, program verification, and static analysis [56, 146]. In the context of model repair, these techniques can also be used to identify redundant constraints, for constraints simplification, and to obtain failure-inducing combinations between two sets of constraints [218].

By definition, the SAT solver needs as input a propositional formula in Conjunctive Normal Form (CNF), normally as DIMACS format, and a variable assignment, and tells whether it is satisfiable or not.

Although the SAT solver is very fast, transforming a propositional formula in an arbitrary format into CNF, although always possible, can be computationally expensive, and may lead to an exponential grow in formula size [181].

This is one of the practical reasons for which for some applications in software engineering, in certain cases, BDDs are preferred over SAT/SMT solvers.

2.2.4 BDDs and MDDs

Binary Decision Diagrams (BDD) and Multi-Valued Decisions Diagrams (MDD) are commonly used structures for representing Boolean functions and multi-valued functions, respectively.

On a more abstract level, BDDs can be considered as a compressed representation of sets or relations. Unlike other compressed representations, operations are performed directly on the compressed representation, i.e. without decompression. Other data structures used to represent Boolean functions include negation normal form (NNF), Zhegalkin polynomials, and propositional directed acyclic graphs (PDAG).

BDDs are extensively used in CAD software to synthesize circuits (logic synthesis) and in formal verification [51]. There are several lesser known applications of BDD, including fault tree analysis, Bayesian reasoning, product configuration, and information retrieval [95].

Although building a BDD is normally computationally more expensive than solving a SAT problem with a state-of-the-art SAT solver, it allows for a deeper analysis over the propositional formula that leads to the BDD, in particular:

- Using the BDD representation, it is easy to count the number of satisfiable configurations. This may be useful in certain contexts such when this number is a part of a fitness function of a process, or is needed for evaluation.

- BDDs do not need the input formula (or set of constraints) to be in a CNF representation, therefore avoiding the potentially exponential complexity of the CNF transformation.

- BDDs can be more convenient, and potentially faster, when multiple assignments are to be evaluated over the same propositional formula, because the tree is built only once (building the BDD is the complex operation), and then just traversed many times to check for satisfiability.

Chapter 3. Automated Model Repair

We devise an iterative process to automatically repair models: the model is modified until all the non-conformances between the model and the system (revealed by the generated tests), are solved. Fig. 2 shows an overview of such test-driven repair approach. Among testing techniques to be employed, black-box testing is an effective technique to detect faults in a system focusing on the inputs, without needing access to the code, but requiring simply an oracle that states if a particular test passes or fails. Combinatorial Interaction Testing (CIT), in particular, has shown to achieve high coverage in software systems for which a parametric, configurable model can be defined. Moreover, we use mutation analysis and search-based methods for *feature models*, a model that has also a graphical tree representation, and these methods tend to apply *small* edits to the model, helping preserve domain knowledge.

Unlike processes that detect faults and repair code, the main hypothesis of this project is that testing techniques can be used not only for fault detection and localization, but also for model repair. This methodology aims at changing the model *with less impact as possible* (to preserve domain knowledge) to *repair* the non-conformance w.r.t. the *solution space* (it can be the current implementation, a new specification, or an available *oracle*). The applied repairs are local and little, based on the *competent modeler* assumption, that the model is *a little* outdated, and minimal changes are enough.

3.1 Research Hypothesis

By making the assumption that the fault resides in the model, and the implementation is correct, the rationale that drives this PhD proposal is the investigation of software testing techniques to automatically *repair* models. Model *repair* can be useful whenever the model is outdated w.r.t. the implementation, or a defined oracle or the specification. Sometimes an engineer also wants to detect which is a model of a current aspect of an implemented system, and can use testing techniques not only to find bugs in a system, but also to *repair* or to entirely *learn* a model from the system implementation. The main hypothesis of this research project is that testing techniques can be used not only for fault detection and localization, but also for model repair.

3.1.1 Assumptions

The applicability of the proposed failure-driven model repair framework is subject to the following assumptions, that we identified:

- prescriptive model: there should be an initial prescriptive model of the system, from which tests can be then generated;
- traceability: the parameters in the model can be traced back to variable in the system. It should be always possible to translate an abstract configuration (i.e., a test), into a concrete test;
- oracle definition: there should be the possibility to execute a test towards the system, and obtain a *pass/fail* result, e.g., depending if the system compiles correctly, doesn't crash, doesn't expose a vulnerability, etc.
- faulty model: we assume that the fault(s), if present, reside in the model, while the system is correct, and can be used as oracle.

3.2 Contribution

The contribution of this PhD project is the proposal and evaluation of testing techniques to repair models of software systems. The methodology aims at changing the model *with less impact as possible* (to preserve domain knowledge) to *repair* the non-conformance w.r.t. the *solution space* (it can be the current implementation, a new specification, or an available *oracle*).

From the initial prescriptive model, (1) we generate and execute tests (test generation), (2) we detect failing interactions (fault localization), and (3) we repair the model accordingly (model repair), obtaining a *descriptive* model of the system, that we assume to *accurately* represent the new *prescriptive* model of the system.

3.3 Research Approach

The research process, designed in accordance with the defined objectives, consists of two stages: the reviewing stage, and the formulation stage. In the reviewing stage, we evaluate existing techniques that may be useful for the objective, and explore possible application scenarios, to determine the kind of models that can be repaired from software testing results. In the formulation stage, for each scenario, we try to formulate the concepts and *decline* the repair problem to that particular kind of model; then we apply or tailor the existing techniques to propose a solution to the model repair problem. Among testing techniques to be employed, black-box testing is an effective technique to detect faults in a system focusing on the inputs, without needing access to the code, but requiring simply an oracle that states if a particular test passes or fails. Combinatorial Interaction Testing (CIT), in particular, has shown to achieve high coverage in software systems for which a parametric, configurable model can be defined. Mutation analysis and search-based methods are also to take into consideration, when possible, as they tend to apply *small* edits to the model, helping preserve domain knowledge. The process is iterative: the model is modified until all the non-conformances between the model and the system (revealed by the generated tests), are solved. Fig. 2 shows an overview of such test-driven repair process. Benchmarks should be selected for evaluation, and tools to implement the repair process should be reused if existing, or customized or built if needed.

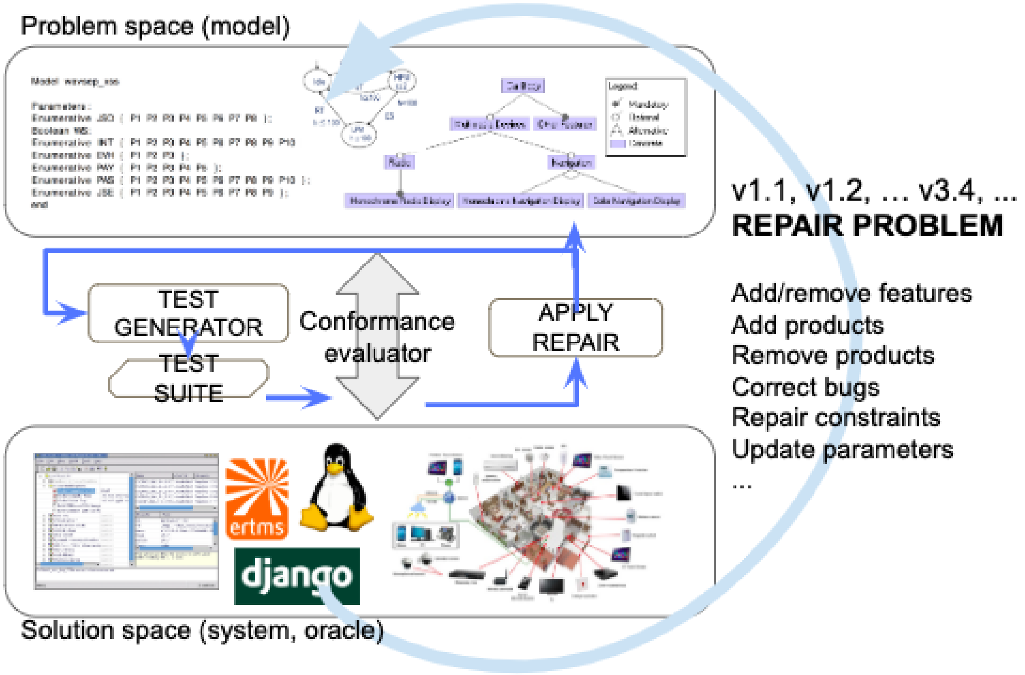


Figure 2: Test-Driven process to repair models

3.4 Results

In the initial stage of the research project we applied the repair process to *configurable* models. Configurable models can model, for example, system features decided at *compile time* such as preprocessor directives, or decided at *operation time* such as configuration files, and also constraints among parameters in a running system.

In the following subsections we briefly describe the results obtained from the application of test-driven repair in three kinds of configurable models: combinatorial models, feature models, and timed automata. Results are then reported in details in Chapter 4 for combinatorial models, Chapter 5 for feature models, and Chapter 6 for timed automata.

3.4.1 Repair of Configuration Constraints

A model for a combinatorial problem consists of parameters which can take various domain values. Combinatorial models may have also constraints among parameter values to, for example, model inconsistencies between certain hardware components, limitations of the possible system configurations, or simply because of design choices. Some methods have been introduced to automate the process of inferring constraints, but they do not aim at *repairing* existing ones [8, 196]. Therefore, in [87] we proposed an iterative approach that uses a fault-localization tool based on combinatorial testing, called BEN, and CIT policies introduced in our previous work [88] to find failure-inducing combinations of parameter values. The model is then repaired *logically*, by translating such failure-inducing combinations as expression in propositional logic. We also developed a tool to make it easier the editing of such combinatorial models, and the test suite generation, called CTWedge [89].
Chapter 4 presents results in detail.

3.4.2 Repair of Feature Models

Feature models are a widely used modeling notation for variability management in software product line (SPL) engineering. In order to keep an SPL and its feature model aligned, feature models must be changed by including/excluding new features and products, either because faults in the model are found or to reflect the normal evolution of the SPL. Such changes can be complex and error-prone due to the size of the feature model. Therefore, we try to *repair* a feature model w.r.t. a given update request in the form of a set of configurations to be accepted or rejected, that may come from *failing* test cases, or from a change in the specification. The method is based on an evolutionary algorithm that iteratively mutates the original feature models and checks if the update request is semantically fulfilled [26, 27]. The approach has been evaluated on real-world feature models: although it does not guarantee to completely update all the possible feature models, on average, around 89% of requested changes are applied, with minimal edits, helping in preserving domain knowledge. Chapter 5 presents results in detail.

3.4.3 Repair of Timed Automata

We apply the repair framework also to timed automata clock guards. In this case, a test is a timed sequence (i.e., actions associated to an absolute time) that can be exercised on the real system. Our approach generates an abstraction of the initial TA in terms of a PTA, generates some tests, and then refines the abstraction by identifying only those TAs contained in the PTA that correctly evaluate all the tests. Chapter 6 presents results in detail.

Chapter 4. Repair of Constraints Among Parameters

A model for a combinatorial problem consists of parameters which can take various domain values. Combinatorial models may have also constraints among parameter values to, for example, model inconsistencies between certain hardware components, limitations of the possible system configurations, or simply because of design choices [87].

In this chapter we try to answer the research question RQ3 from Section 1.1: *How to repair configuration constraints using software testing?*. To answer this question, we consider combinatorial models as notation to express configuration constraints, and we describe an iterative approach to repair combinatorial models that uses a fault-localization tool based on combinatorial testing, called BEN [97], and combinatorial test generation policies to find failure-inducing combinations of parameter values. The model is then repaired *logically*, by translating such failure-inducing combinations into expressions in propositional logic. The repairs are of two types, depending on whether the model is true and the system (i.e., the oracle) false for a given test case (in this case, the model is *under-constrained*) or vice-versa (in this case, the model is *over-constrained*). Tab. 1 reports possible scenarios in which such condition may occur in a system with three boolean parameters A, B, C, that map to directives in a C program in which both B and C can be enabled only if also A is activated.

Table 1: Test suites with faults (in gray)

(a) under-constraining fault

A	B	C	$\mathcal{M}_{f1}$	oracle
T	T	F	T	F
T	F	F	F	F

(b) over-constraining fault

A	B	C	$\mathcal{M}_{f2}$	oracle
F	T	T	T	T
F	F	T	T	T
F	T	F	F	T
F	F	F	F	T

Experiments for five real-world systems (Libssh, Telecom, Aircraft, Concurrency, and Django) show that our approach can repair on average 37% of conformance faults. Moreover, we also notice that it can infer and repair parameter constraints for the configurations that lead to a successful startup of Django, a well-known open source web application framework written in Python.

4.1 Validation of Constraints Among Configuration Parameters Using Search-Based Combinatorial Interaction Testing

The appeal of highly-configurable software systems lies in their adaptability to users' needs. Search-based Combinatorial Interaction Testing (CIT) techniques have been specifically developed to drive the systematic testing of such highly-configurable systems. In order to apply these, it is paramount to devise a model of parameter configurations which conforms to the software implementation. This is a non-trivial task. Therefore, we extend traditional search-based CIT by devising 4 new testing policies able to check if the model correctly identifies constraints among the various software parameters. Our experiments show that one of our new policies is able to detect faults both in the model and the software implementation that are missed by the standard approaches.

Most software systems can be configured in order to improve their capability to address user's needs. Configuration of such systems is generally performed by setting certain parameters. These options, or *features*, can be created at the software design stage (e.g., for *software product lines*, the designer identifies the features unique to individual products and features common to all products in its category), during compilation (e.g., to improve the efficiency of the compiled code) or while the software is running (e.g., to allow the user to switch on/off a particular functionality). A configuration file can also be used to decide which features to load at startup.

Large configurable systems and software product lines can have hundreds of features. It is infeasible in practice to test all the possible configurations. Consider, for example, a system with only 20 Boolean parameters. One would have to check over one million configurations in order to test them all (2^{20} to be exact). Furthermore, the time cost of running one test could range from fraction of a second to hours if not days. In order to address this combinatorial explosion problem, Combinatorial Interaction Testing (CIT) has been proposed for testing configurable systems [63]. It is a very popular black-box testing technique that tests all interactions between any set of t parameters. There have been several studies showing the successful efficacy and efficiency of the approach [125, 126, 168].

Furthermore, certain tests could prove to be infeasible to run, because the system being modelled can prohibit certain interactions between parameters. Designers, developers, and testers can greatly benefit from modelling parameters and constraints among them by significantly reducing modelling and testing effort [168] as well as identifying corner cases of the system under test. Constraints play a very important role, since they identify parameter interactions that need not be tested, hence they can significantly reduce the testing effort. Certain constraints are defined to prohibit generation of test configurations under which the system simply should not be able to run. Other constraints can prohibit system configurations that are valid, but need not be tested for other reasons. For example, there's no point in testing the *find* program on an empty file by supplying all possible strings.

Constructing a CIT model of a large software system is a hard, usually manual task. Therefore, discovering constraints among parameters is highly error prone. One might run into the problem of not only producing an incomplete CIT model, but also one that is over-constrained. Even if the CIT model only allows for valid configurations to be generated, it might miss important system faults if one of the constraints is over-restrictive. Moreover, even if the system is not *supposed* to run under certain configurations, if there's a fault, a test suite generated from a CIT model that correctly mimics only desired system behaviour will not find that error. In such situations tests that exercise those corner cases are desirable.

The objective of this work is to use CIT techniques to validate constraints of the model of the system under test (SUT). We extend traditional CIT by devising a set of six policies for generating tests that can be used to detect faults in the CIT model as well as the SUT.

4.1.1 Combinatorial Models of Configurable Systems

[88] Combinatorial Interaction Testing (CIT), or simply combinatorial testing, aims to test the software or the system with selected combinations of parameter values. There exist several tools and techniques for CIT. Good surveys of ongoing research in CIT can be found in [101, 156], while an introduction to CIT and its efficacy in practice can be found in [129, 168].

A model for a combinatorial problem consists of several parameters which can take several domain values. In most configurable systems, dependencies exist between parameters. Such constraints may be introduced for several reasons, e.g., to model inconsistencies between certain hardware components, limitations of the possible system configurations, or simply design choices [63]. In our approach, tests that do not satisfy the constraints in the CIT model are considered *invalid*.

We assume that the models are specified using CitLab [90, 58]. This is a framework for combinatorial testing which provides a rich abstract language with precise formal semantics for specifying combinatorial problems, and an eclipse-based editor with a rich set of features. CitLab does not have its own test generators, but it can utilize, for example, the search-based combinatorial test generator CASA[1][94]. CIT problems can be formally defined as follows.

Definition 4.1 (Constrained Model)**.** *Let $P = \{p_1, \ldots, p_m\}$ be the set of parameters. Every parameter p_i assumes values in the domain $D_i = \{v^i_1, \ldots, v^i_{o_i}\}$. Every parameter has its name (it can have also a type with its own name) and every enumerative value has an explicit name. We denote with $C = \{c_1, \ldots, c_n\}$ the set of constraints.*

[1] `http://cse.unl.edu/~citportal/`

Definition 4.2 (Strength of a test suite)**.** *The objective of a CIT test suite is to cover all parameter interactions between any set of t parameters. t is called the strength of the CIT test suite. For example, a pairwise test suite covers all combinations of values between any 2 parameters.*

```
Model WashingMachine
Definitions:
Number maxSpinHL = 1400;
end
Parameters:
Boolean HalfLoad;
Enumerative Rinse {Delicate Drain Wool};
Numbers Spin { 800 1200 1800 };
end
Constraints:
# HalfLoad => Spin < maxSpinHL #
# Rinse==Rinse.Delicate =>
( HalfLoad and Spin==800) #
end
```

(a) Washing Machine example

```
Model Greetings
Parameters:
Boolean HELLO;
Boolean BYE;
end
Constraints:
# HELLO != BYE#
end
```

```
#ifdef HELLO
char* msg = "Hello!\n";
#endif
#ifdef BYE
char* msg = "Bye bye!\n";
#endif

void main() {
printf(msg);
}
```

(b) Compile time configurable example, its CIT model (left) and the source code (right)

Figure 3: Combinatorial interaction CitLab models

Constraints c_i are given in general form, using the language of propositional logic with equality and arithmetic. Fig. 3a shows the CitLab model of a simple washing machine consisting of 3 parameters. The user can select if the machine has `HalfLoad`, the desired `Rinse`, and the `Spin` cycle speed. There are two constraints, including, if `HalfLoad` is set then the speed of spin cycle cannot exceed `maxSpinHL`.

Software systems can be configured by setting specific parameter values at different stages of the software testing process.

Compile time Configurations can be set at compile time. An example is shown in Figure 3b. Depending on the value settings of the Boolean variables `HELLO` and `BYE` different messages will be displayed when the program is run.

Design time Configurations can also be set at design time. For example, in case of a SPL, a configurability model is built during the design.

Runtime Another way of setting parameter configurations is at runtime. This can be usually done by means of a graphical user interface (GUI). In a chat client, e.g., you can change your availability status as the program is running.

Launch time We also differentiate the case where parameters are read from a separate configuration file or given as program arguments, *before* the system is run. We say that these parameters are set at launch time of the given application. They decide which features of the system should be activated at startup. Examples of such systems include chat clients, web browsers and others.

4.1.2 Basic Definitions

We assume that the combinatorial model represents the specification of the parameters and their constraints for a real system as it has been implemented. We are interested in checking whether this system specification correctly represents the software implementation. We assume that the parameters and their domains are correctly captured in the specification, while the constraints may contain some faults. Specification S belongs to the problem space while software implementation I belongs to the solution space [153].

Formally, given an assignment $\bar{p}$ that assigns a value to every parameter in P of the model S, we introduce two functions:

Definition 4.3. *Given a model S and its implementation I, val_S is the function that checks if assignment $\bar{p}$ satisfies the constraints in S, while $oracle_I(\bar{p})$ checks if $\bar{p}$ is a valid configuration according to implementation I.*

We assume that the oracle function $oracle_I$ exists. For instance, in case of a compile-time configurable system, we can assume that the compiler plays the role of an oracle: if and only if the parameters $\bar{p}$ allow the compilation of the product then we say that $oracle(\bar{p})$ holds. We may enhance the definition of oracle by considering also other factors, for example, if the execution of the test suite completes successfully. However, executing $oracle_I$ may be very time consuming and it may require, in some cases, human intervention.

On the model side, the evaluation of $val_S(P)$ is straightforward, that is, $val_S(\bar{p}) = c_{1[P \leftarrow \bar{p}]} \wedge \ldots \wedge c_{n[P \leftarrow \bar{p}]}$.

Definition 4.4. *We say that the Constrained CIT (CCIT) model is correct if, for every p, $val_S(p) = oracle_I(p)$. We say that a specification contains a* conformance *fault if there exists a $\bar{p}$ such that $val_S(\bar{p}) \neq oracle_I(\bar{p})$.*

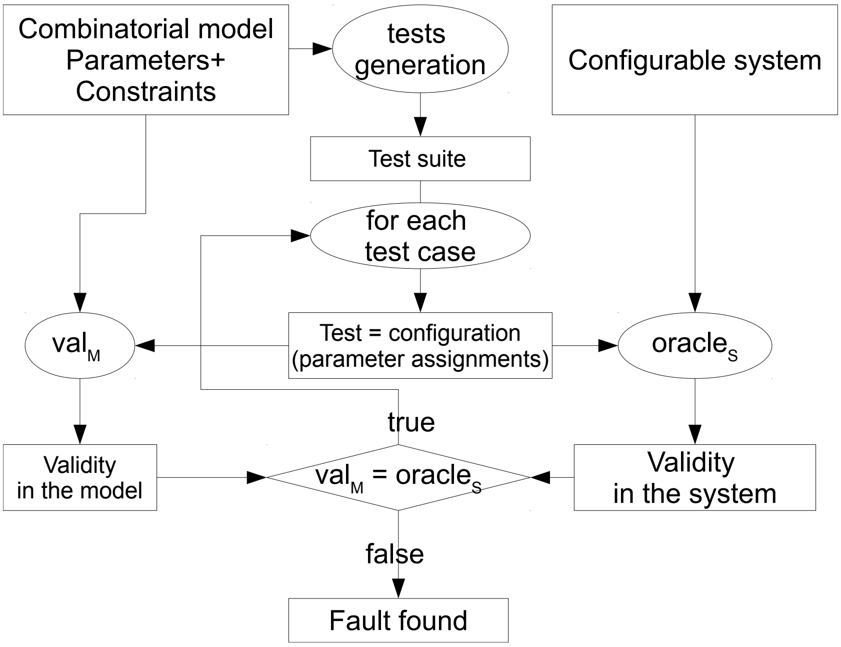


Figure 4: Validating constraints by CIT

4.1.3 Finding Faults by Combinatorial Testing

In order to find possible faults as defined in Definition 4.4, the exhaustive exploration of all the configurations of a large software system is usually impractical. In many cases, the evaluation of $oracle_I$ is time consuming and error prone, so the number of tests one can check on the implementation can be very limited. Instead, we can apply combinatorial testing in order to select the parameters values and check that for every generated CIT test $val_S(p) = oracle_I(p)$ holds. This approach does not guarantee, of course, finding all possible conformance faults, but we can assume that faults are due to the interaction of parameters and we can leverage the success of CIT in finding faults in real configurable systems [126, 168].

We have devised a process that is able to find possible conformance faults. It is depicted in Figure 4 and consists of the following steps:

1. Create a CIT model S that takes constraints into account.
2. Generate a CIT test suite according to one of the policies (see Section 4.1.4).

3. For every test in the test suite,

    a) Compute its validity as specified by the constraints in the CIT model.

    b) Compute $oracle_I$, by executing the software system under each configuration to check if it's acceptable.

    c) Compare the validity, as defined by the model, with the actual result.

    d) If $val_S \neq oracle_I$ a fault (either in the model or in the system) is found.

A discrepancy between the model and the real system means that a configuration is correct according to the model but rejected by the real system (or the other way around) and this means that the constraints in the model do not correctly describe constraints in the system under test.

Invalid Configuration Testing.

In classical combinatorial interaction testing, only valid tests are generated, since the focus is on assessing if the system under test produces valid outputs. However, we believe that invalid tests are also useful. In particular, they address the following issues.

The CIT model should minimize the number of constraints and the invalid configuration set: invalid configurations, according to the model, should only be those that are actually invalid in the real system. This kind of test aims at discovering faults of over-constraining the model. This problem is a variant of the bigger problem of over-specification. Moreover, critical systems should be tested if they safely fail when the configuration is incorrect. This means that the system should check that the parameters are not acceptable (i.e. it must *fail*) and it should fail in a safe way, avoiding crashes and unrecoverable errors (it must fail *safely*). Furthermore, creation of a CIT model for a large real-world software system is usually a tedious, error-prone task. Therefore, invalid configurations generated by the model at hand can help reveal constraints within the system under test and help refine the CIT model. In line with the scientific epistemology, our research focuses on generating not only tests (i.e., valid configurations) that confirm our theory (i.e., the model), but also tests that can *refute* or *falsify* it. Since the number of invalid configurations might be huge, such configurations must be chosen in accordance with some criteria. We choose to use the same t-way interaction paradigm as in standard CIT.

4.1.4 Combinatorial Testing Policies

We propose to use search-based combinatorial interaction testing techniques to verify the validity of CIT models. In particular, given a CIT model, we modify it according to one of the policies introduced in this section. Next, we use CASA to generate the test suite satisfying the modified CIT model. We use the term “valid test" to denote the generated configuration that satisfies all the constraints of the original CIT model. Conversely, the term “invalid test" is used for a configuration that does not satisfy at least one of the constraints of the original CIT model. Words “test" and “configuration" are used interchangeably, though we note in real-world systems one configuration may lead to multiple tests.

UC: Unconstrained CIT.

In unconstrained CIT, constraints are ignored during CIT test generation. They are used only to check the validity of the configuration selected during generation. The main advantage is that test generation is simplified and efficient methods that work without constraints can be used. Moreover, in principle, both valid and invalid configurations can be generated - there is no control over model validity. It may happen that the test generation algorithm generates only valid combinations (i.e., $val_S(t)$ for every t in the test suite). This may reduce the effectiveness of the test suite: if only valid tests are generated, one can miss faults only discoverable by invalid tests, as explained in Section 4.1.3. On the other hand, only invalid tests can be equally useless.

Example 4.1. *In the washing machine example shown in Fig. 3a, UC policy will produce a pairwise test suite with at least 9 test cases, including an invalid test case where* `HalfLoad` *is set to true in combination with* `Spin` *equal to 1800.*

Test generation. UC can be applied by simply removing the constraints c_i from the original CIT model. The validity of each test can be later computed by checking if the generated configuration satisfies all the c_i. There are several CIT tools that do not handle constraints (for example, those that use algebraic methods for CIT test suite generation), hence can be used with this policy.

CC: Constrained CIT.

In this classical approach, constraints are taken into account and only valid combinations among parameters are chosen. Among these parameters a certain level of desired strength is required. The rationale behind this policy is that one wants to test only valid combinations. If a certain interaction among parameters is not possible, then it is not considered even if it would be necessary in order to achieve the desired level of coverage. The main advantage is that no error should be generated by the system. However, this technique can only check one side of equation given in Def. 4.7, namely that $val_S(p) \rightarrow oracle_I(p)$, since val_S is always true. If the specification is too restrictive, no existing fault will be guaranteed to be found, if it refers to configurations that are invalid.

Example 4.2. *In the washing machine example shown in Fig. 3a, the CC policy produces 7 tests for pairwise, all of which satisfy the constraints. Some pairs are not covered: for instance* `HalfLoad`*=true and* `Spin`*=1800 will not be covered.*

Test generation. CC is the classical constrained combinatorial testing (CCIT), and CASA can correctly deal with the constraints and generate only valid configurations. However, CASA requires the constraints in Conjunctive Normal Form (CNF), so CitLab must convert the constraints from general form to CNF.

CV: Constraints Violating CIT.

In case one wants to test the interactions of parameters that produce errors, only tests violating the constraints should be produced. This approach is complementary with respect to the CC in which only valid configurations are produced. In CV, we ask that the maximum possible CIT coverage for a given strength is achieved considering only tuples of parameter values that make at least one constraint false (i.e. each test violates the conjunction $c_1 \wedge \cdots \wedge c_n$).

Example 4.3. *In the example presented in Fig. 3a, the CV policy produces 6 test cases, all of which violate some constraint of the model. For instance, a test has* `Rinse`*=Delicate,* `Spin` *=800, and* `HalfLoad`*=false.*

Test generation. CV can be applied by modifying the model by replacing all the constraints with $\neg(c_1 \wedge \cdots \wedge c_n)$ and then classical CC is applied.

CuCV: Combinatorial Union.

One limitation of the CC technique is that with an over-constrained model, certain faults may not be discovered. On the other hand, by generating test cases violating constraints only, as in CV, certain parameter interactions may not be covered by the generated test suite. In order to overcome these limitations we propose the combination of CC and CV.

Test generation. CuCV is achieved by generating tests using policy CC and policy CV and then by merging the two test suites. Since every test is either valid (in CC) or invalid (in CV), merging the test suites consists of simply making the union of the two test suites.

ValC: CIT of Constraint Validity.

CuCV may produce rather big test suites, since it covers all the desired parameter interactions that produce valid configurations *and* all those that produce invalid ones according to the given CIT model. On the other hand, UC may be too weak since there is no control over the final constraint validity and therefore there is no guarantee that the parameter values will influence the final validity of the configuration. On one extreme, UC might produce a test suite without any test violating the constraints. We propose the ValC policy that tries to balance the validity of the tests without requiring the union of valid and invalid tests. ValC requires the interaction of each parameter with the validity of the whole CIT model. That is, both tests that satisfy all the constraints will be generated as well as those that don't satisfy any of the constraints in the given CIT model. Formally, ValC requires that the validity of each configuration $\bar{p}$ (i.e., $val_S(\bar{p})$) is covered in the same desired interaction strength (see Definition 4.2) among all the parameters.

Example 4.4. *For the WashingMachine, CuCV generates 13 test cases (6+7). ValC requires only 11 test cases.*

Test generation. ValC requires to modify the original CIT model by introducing a new Boolean variable validity and replacing all the constraints with one constraint equal to validity $\leftrightarrow (c_1 \wedge \cdots \wedge c_n)$

CCi: CIT of the Constraints.

Every constraint may represent a condition over the system state. For instance, the constraint HalfLoad => Spin < maxSpinHL identifies the critical states in which the designer wants a lower spin speed. One might consider each constraint as a property of the system and be interested in covering how these conditions interact with each other and with the other parameters. The goal is to make the constraints interact with the other system parameters.

Test generation. CCi requires the introduction of a new Boolean variable $\mathsf{validity}_i$ for every constraint, and replacing every constraint c_i with $\mathsf{validity}_i \Leftrightarrow c_i$.

4.1.5 Experiments

In order to test our proposed approach we conducted the following experiments. We used 4 case studies to evaluate our proposed approach:

1. **Banking1** represents the testing problem for a configurable Banking application presented in [180].

2. **libssh** is a multi-platform library implementing SSHv1 and SSHv2 written in C[2]. The library consists of around 100 KLOC and can be configured by several options and several modules (like an SFTP server and so on) can be activated during compile time. We have analysed the `cmake` files and identified 16 parameters and the relations among them. We have built a feature model for it in [33] and we have derived from that a CitLab model.

3. **HeartbeatChecker** is a small C program, written by us, that performs a Heartbeat test on a given TLS server. The Heartbeat Extension is a standard procedure (RFC 6520) that tests secure communication links by allowing a computer at one end of a connection to send a “Heartbeat Request" message. Such a message consists of a payload, typically a text string, along with the payload’s length as a 16-bit integer. The receiving computer then must send exactly the same payload back to the sender. HeartbeatChecker reads the data to be used in the Heartbeat from a configuration file with the following schema:

```
TLSserver: <IP>
TLS1_REQUEST Length: <n1> PayloadData: <data1>
TLS1_RESPONSE Length: <n2> PayloadData: <data2>
```

Configuration messages with `n1` equal to `n2` and `data1` equal to `data2` represent a successful Heartbeat test (when the TLS-server has correctly responded to the request).

[2] `https://www.libssh.org/`

HeartbeatChecker can be considered as an example of a *runtime* configurable system, since thanks to the parameters one can perform different types of tests (with different lengths and payloads). We have written an abstract version of HeartbeatChecker in the combinatorial model shown in Fig. 5: we ignore the actual content of the PayloadData and we model only the lengths: Length represents the declared lengths and PayloadData_length is the actual length of the PayloadData. The constraints represent successful exchanges of messages in the Heartbeat test. The oracle is true if the Heartbeat test has been successfully performed with the specified parameters.

```
Model Heartbeat
Parameters:
Range REQ_Length [ 0 .. 65535 ] step 4369;
Range REQ_PayloadData_length [ 0 .. 65535 ] step 4369;
Range RES_Length [ 0 .. 65535 ] step 4369;
Range REQ_PayloadData_length [ 0 .. 65535 ] step 4369;
end
Constraints:
// the declared length in the REQUEST is correct
# REQ_Length==REQ_PayloadData_length #
// the declared length in the RESPONSE is correct
# RES_Length==RES_PayloadData_length #
// the RESPONSE has the same length as the REQUEST
# REQ_Length==RES_Length #
end
```

Figure 5: HeartbeatChecker CIT model

4. **Django** is a free and open source web application framework, written in Python, consisting of over 17k lines of code, that supports the creation of complex, database-driven websites, emphasizing reusability of components[3]. Each Django project can have a configuration file, which is loaded every time the web server that executes the project (e.g. Apache) is started. Therefore, the configuration parameters are loaded at *launch time*. In the model we made, among all the possible configuration parameters, we selected and considered one Enumerative and 23 Boolean parameters. We elicited the constraints from the documentation, including several forum articles and from the code when necessary. We have also implemented the oracle, which is completely automated and returns true if and only if the HTTP response code of the project homepage is 200 (HTTP OK).

[3] https://www.djangoproject.com/

name	#var	#constraints	#configurations	vr	#pairs
Banking1	5	112	324	65.43%	102
Libssh	16	2	65536	50%	480
HeartbeatChecker	4	3	65536	0.02%	1536
Django	24	3	33554432	18.75%	1196

Table 2: Benchmark data. vr **is the validity ratio, defined as the percentage of configurations that are valid.**

Table 12 presents various benchmark data: number of variables and constraints, size of the state space (the total number of possible configurations), the percentage of configurations that are valid (i.e. the ratio vr), the number of pairs that represent the pairwise testing requirements (ignoring constraints). Note that a low ratio indicates that there are only few valid configurations (see, for example, the HeartbeatChecker benchmark). We collected models of real-world systems from different domains, with a good level of diversity (in terms of size, constraints, etc.) in order to increase the validity of our findings.

Experiments were executed on a Linux PC with two Intel(R) i7-3930K CPU (3.2 GHz) and 16 GB of RAM. All reported results are the average of 10 runs with a timeout for a single model of 3600 secs. Test suites were produced using the CASA CIT test suite generation tool according to the pairwise testing criterion.

Test generation and coverage

In our first experiment, we are interested in comparing the policies in terms of test *effort* measured by the number of tests and by the test suite generation time. Table 3 presents the following data:

- The *time* required to generate the tests and to evaluate their validity (it does not include the evaluation of the $oracle_I$) in seconds.
- The *size* in terms of the number of tests and how many of those are valid (#Val), i.e. val_S returns true.
- The percentage of parameter interactions (pairs) that are covered. In the count of the pairs to be covered, we ignore constraints as in Table 12.

From Table 3 we can draw the following observations:

	Banking1				Django				libssh				HeartbeatChecker			
Pol.	time	size	#Val	Cov.	time	size	#Val	Cov.	time	size	#Val	Cov.	time	size	#Val	Cov.
UC	0.22	12	11	100%	0.65	10	2	100%	0.25	8	4	100%	447	267	0	100%
CC	0.26	13	13	100%	1.24	10	10	91.8%	0.28	8	8	99.3%	2.74	141	141	6.2%
CV	Out of memory				0.32	11	0	100%	0.25	8	0	99.3%	Out of memory			
CuCV	Out of memory				1.58	21	10	100%	0.52	16	8	100%	Out of memory			
ValC	Out of memory				0.31	11	4	100%	0.29	8	5	100%	Out of memory			
CCi	6.22	12	9	100%	0.58	13	3	100%	0.30	8	2	100%	460	268	0	100%

Table 3: Valid pairwise parameter interactions covered by six test generation policies. (Shaded cells are covered in the prose.) Out of memory errors are due to constraint conversion into the CNF format required by CASA. In particular, as known in the literature, the size in CNF of the negation of a constraint can grow exponentially.

- UC usually produces both valid and invalid tests. However, it may produce all invalid tests (especially if the constraints are strong - see HeartbeatChecker). Having all invalid tests may reduce test effectiveness.

- CC usually does not cover all the parameter interactions, since some of them are infeasible because they violate constraints in the original model. On the other hand, CC generally produces smaller test suites (as in the case of HeartbeatChecker). However, in some cases, CC is able to cover all the required tuples at the expense of larger test suites (as in the case of Banking1).

- CV generally does not cover all the parameter interactions, since it produces only invalid configurations. However, in one case (Django) CV covered all the interactions. This means that 100% coverage of the tuples in some cases can be obtained with no valid configuration generated and this may reduce the effectiveness of testing. Sometimes CV is too expensive to perform.

- CuCV guarantees to cover all the interactions and it produces both valid and invalid configurations. However, it produces the bigger test suites and it may fail because it relies on CV.

- ValC covers all the interactions with both valid and invalid configurations. It produces test suites smaller than CuCV and it is generally faster, but as CuCV may not terminate.

- CCi covers all the interactions, it generally produces both valid and invalid test. However, it may produce all invalid tests (see HeartbeatChecker), and it produces a test suite comparable in size with UC. However, it guarantees an interaction among the constraint validity. It terminates, but it can be slightly more expensive than UC and CC. If the strength of combinatorial testing is greater or equal to the number of constraints, it guarantees also that valid and invalid configurations are generated.

Fault detection capability

We are interested in evaluating the fault detection capability of the tests generated by the policies presented above. We have applied mutation analysis [116] which consists of introducing artificial faults and checking if the proposed technique is able to find them. In our case, we have introduced the faults by hand and then we have applied our technique described in Section 5.1.3 in order to check if the fault is detected (or *killed*). Tables in Fig. 6 present a brief description of each introduced fault and if each policy was able to kill it.

In principle, our technique is able to find conformance faults both in the model and in the implementation. Indeed, when a fault is found, it is the designer's responsibility to decide what is the source of the fault. For libssh we have modified both the model and the code (the `cmake` script) (faults L*x*). For the HeartbeatChecker we have modified the model and the source code (faults H*x*). Table in Fig. 6a presents the details of each injected fault, including if it refers to the specification (S) or to the implementation (I).

libssh	L1	forgot all the constraints	S
	L2	remove a constraint	S
	L3	add a constraint	S
	L5	remove a dependency	I
	L6	add a dependency	I
HeartBeatChecker	H1	remove one constraint	S
	H2	== to <=	S
	H4	&& to ||	I
	H5	== to != (all)	I
	H6	== to != (one)	I
	H7	HeartBleed	I

(a) Seeded Faults (S: in Spec, I: in implementation)

	Is the fault detected?													mut.
Policy	L1	L2	L3	L4	L5	L6	H1	H2	H3	H4	H5	H6	H7	score
UC	✓	✓		✓	✓		✓	✓	✓	✓	✓	✓		10/13
CC	✓	✓				✓	✓	✓	✓		✓			7/13
CV	✓		✓	✓	✓		-	-	-	-	-	-	-	4/13
CuCV	✓	✓	✓	✓	✓	✓	-	-	-	-	-	-	-	6/13
ValC	✓	✓		✓	✓		-	-	-	-	-	-	-	4/13
CCi	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	12/13

(b) Fault detection capability of the policies (- means that the test suite was not generated.)

Figure 6: Fault detection capability

Table in Fig. 6b reports which faults were killed by each policy. We can observe that the unconstrained CIT (UC) policy performs better than some policies that consider constraints (CC and CV) even if normally their test suites have the same dimensions. However, in some cases (L6) CC detected a fault where UC failed. For CV, CuCV, and ValC we can analyze only the results for libssh, since they did not complete the test generation for HeartbeatChecker. However, even if we restrict to libssh, CuCV has a very good fault detection capability (but it produces the biggest test suite) while ValC and CV scored as well as UC, although they are more expensive, so according to our studies there is no particular reason to justify the use of ValC and CV alone. However, in one case (L3) CV detected a fault that UC did not.

Overall CCi was the best in terms of fault detection, even with test suites as big as those for UC. However, it missed one of the injected faults (L6). CCi was the only one to find the fault H7 (*HeartBleed*). The HeartBleed fault simulates the famous Heartbleed security bug of the OpenSSH implementation of the TLS protocol. It results from improper input validation (due to a missing bounds check) in the implementation of the TLS Heartbeat extension. In detail, the implementation built the payload length of message to be returned based on the length field in the requesting message, without regard to the actual size of that message's payload. In our implementation, the faulty HeartbeatChecker missed to check that REQ_Length==RES_Length. This proves that testing how parameters can interact with single constraints increases the fault detection capability of combinatorial testing. Our new policies may thus prove useful in detecting faults missed by standard approaches due to the so-called *masking effects* [215].

4.1.6 Related Work

The problem of modelling and testing the configurability of complex systems is non-trivial. There has been much research done in extracting constraints among parameter configurations from real systems (problem space) and modelling system configurability [183, 107, 215]. For instance, the importance of having a model of variability and having the constraints in the model aligned with the implementation is discussed in [153]. However, in that paper, authors try to identify the sources of configuration constraints and to automatically extract the variability model. Our approach is oriented towards the validation of a variability model that already exists. Moreover, they target C-based systems that realize configurability with their build system and the C preprocessor. A similar approach is presented in [193], where authors extract the compile-time configurability from its various implementation sources and examine for inconsistencies (e.g., dead features and infeasible options). We believe that our approach is more general (not only compile-time and C-code) and can be complementary used to validate and improve automatically extracted models.

Testing configurable systems in the presence of constraints is tackled in [63] and [168]. In these papers, authors argue that CIT is a very efficient technique and that constraints among parameters should be taken into account in order to generate only valid configurations. This allows to reduce the cost of testing. Also in [57], authors have shown how to successfully deal with the constraints by solving them by using a constraint solver such as a Boolean satisfiability solver (SAT). However, the emphasis of that research is more on testing of the final system not its model of configurability. CIT is also widely used to test SPLs [167].

In SPL the validation and extraction of constraints between features is generally given in terms of feature models (FMs). Synthesis of FMs can be performed by identifying patterns among features in products and in invalid configurations and build hierarchies and constraints (in limited form) among them. For instance, Davril et al. apply feature mining and feature associations mining to informal product descriptions [70]. There exist several papers that apply search based techniques, which generally give better results [104, 136, 83, 138]. However, checking and maintaining the consistency between a SPL and its feature model is still an open problem. A preliminary proposal is presented in [33], which however does not use CIT but a more complex logic based approach. We plan to compare our approach with [33] in order to check if CIT can provide benefits in terms of easiness in test generation and shorter generation times.

4.2 Combinatorial Interaction Testing for Automated Constraint Repair

This section presents a process to automatically *repair* constraints of a combinatorial model, using combinatorial interaction testing, and reflects the content of paper [87].

Combinatorial models can represent an aspect of configurable systems, and constraints may be used to encode interactions among those parameters: allowed interactions, or forbidden interactions. It is a fact that most software systems have the possibility to be configured in order to improve their capability to address users' needs. Configuration of such systems is generally performed by setting system parameters. These options, or *features*, can be created during design time. For instance, in the case of a *software product line*, the designer identifies the features unique to individual products and features common to all products in its category.

Such options can also be decided during compilation time, in order to improve some characteristics of the compiled code (scalability, efficiency, etc.). For example, in case of preprocessor directives, the programmer can decide which libraries to use, what code to execute and what to ignore etc. Software configurations can also be modified during operation time, when the system is already running and the user wants to switch on/off a particular feature or functionality. In this case, for example, the parameters can be saved in a configuration file and modified if necessary. Such a configuration file can also be used to decide which features to load at startup. In this scenario, we distinguish between *problem space* to represent the modeling of the system configuration parameters and their dependencies, while the *solution space* is the technical realization of the system [153], i.e., its implementation.

We use combinatorial interaction testing (CIT) because it showed a high fault coverage with very few tests, even for a high amount (dozens, or a few hundreds) of configuration parameters [128]. Large configurable systems and software product lines, in fact, can have hundreds of features. It is infeasible in practice to test all the possible configurations. Consider, for example, a system with only 20 Boolean parameters. One would have to check over one million configurations in order to test them all.

Furthermore, the time cost of running one test could range from fraction of a second to hours if not days.

CIT, as introduced in Sect. 2.2.1, is a very popular black-box testing technique that tests all interactions between any set of *t* parameters. There have been several studies showing the successful efficacy and efficiency of the approach [126, 125, 168, 169].

Moreover, certain tests could prove to be infeasible to run, because the system being modeled can prohibit certain interactions between parameters. Designers, developers, and testers can greatly benefit from modelling parameters and constraints among them by significantly reducing modelling and testing effort [168] as well as identifying corner cases of the system under test. Constraints play a very important role, since they identify parameter interactions that need not be tested, hence they can significantly reduce the testing effort. Certain constraints are defined to prohibit generation of test configurations under which the system simply should not be able to run. Other constraints can prohibit system configurations that are valid, but need not be tested for other reasons. For example, there’s no point in testing the *find* program on an empty file by supplying all possible strings.

Constructing a CIT model of a large software system is a hard, usually manual task. Therefore, discovering constraints among parameters is highly error prone. One might run into the problem of not only producing an incomplete CIT model, but also one that is over-constrained. Even if the CIT model only allows for valid configurations to be generated, it might miss important system faults if one of the constraints is over-restrictive. Moreover, even if the system is not *supposed* to run under certain configurations, if there’s a fault, a test suite generated from a CIT model that correctly mimics only desired system behavior will not find that error. In such situations tests that exercise those corner cases are desirable.

Software evolution is a typical scenario in which automated repair of the model of system's configurations may be useful. The initial model can conform with the system, but when the system or its configuration space undergo changes, the model becomes non-conforming and needs to be repaired.

The problem of finding and fixing conformance faults between a given software system and its combinatorial model is a challenging task. Due to the size and complexity of current software systems, the interactions between different software parameters are hard to find. Combinatorial models are frequently derived manually, by expert software engineers. The Software-artefact Infrastructure Repository[4], for instance, contains models of dozens of software systems, which were derived by hand.

Several methods have been introduced to automate the process of inferring constraints. However, they do not guarantee that the derived model will be correct [196]. Therefore,

we introduce a novel automated approach for finding and fixing conformance faults between the given software system and its combinatorial model.

We use combinatorial interaction testing policies introduced in our previous work [88] to find such faults and fix them by repairing constraints in the original CIT model.

We conduct several experiments aiming to answer the following research questions:

[**RQ1:**] *How effective is our automated constraint repair approach at fixing faults in an existing CIT model?*

In particular, we apply our approach to mutated CIT models and show that our approach can automatically fix on average 37% of conformance faults.

[**RQ2:**] *How effective is our approach in inferring parameter constraints in real-world software?*

We present a case study that shows that our approach can be used to derive constraints for an unconstrained CIT model of a large real-world software system.

[**RQ3:**] *How efficient is our constraint repair approach?*

In order to evaluate the efficiency of our approach, we measure the time taken to repair constraints in CIT models as well as the number of tests needed for the repair. On the five systems evaluated, the mutated models were repaired within seconds.

This section reports the content of the paper [87], and it is structured as follows: Section 4.2.1 gives a brief overview of combinatorial interaction testing activities with the CitLab tool; Section 4.2.2 briefly describes CIT policies used in our approach; Section 4.2.3 presents definitions and notation used throughout the section; Section 4.2.4 presents our approach to fixing conformance faults between

[4] `http://sir.unl.edu/portal/index.php`

a software system and its combinatorial model; Section 4.2.5 contains experimental results and answers to research questions posed; and Section 4.2.6 presents related work. Conclusions and future work are discussed jointly at the end of the chapter, in Section 4.4.

4.2.1 Combinatorial Models and Testing of Configurable Systems

> *[87]* Combinatorial Interaction Testing (CIT), often called combinatorial testing or combinatorial testing design, aims to test configurable software systems under the various combinations of its parameter values. There exist several tools and techniques for CIT. Good surveys of ongoing research in CIT can be found in [101, 156], while an introduction to CIT and its efficacy in practice can be found in [129, 168].

Combinatorial models and CitLab

> A model for a combinatorial problem consists of several parameters (at least 2) which can take various domain values. In most configurable systems, constraints (or dependencies) exist between parameters.
>
> Constraints might be introduced for several reasons, for example, to model inconsistencies between certain hardware components, limitations of the possible system configurations, or simply because of design choices [64]. Constraints were first described as being important to combinatorial testing in [62] and were introduced in the AETG system. In our approach tests that do not satisfy the constraints in the CIT model are considered *invalid*.
>
> In this section we assume that the models are specified using CitLab [90, 58].

For the later work presented in Section 4.3, instead, we use the successor of CitLab: ctwedge (described in Chapter 7).

> CitLab is a framework for combinatorial testing which provides a rich abstract language with precise formal semantics for specifying combinatorial problems, an eclipse-based editor with a rich set of features (syntax highlighting, autocompletion, outline view, and others), and a Java API library which includes utility methods for generating all the test requirements for combinatorial coverage of given strength. CitLab does not have its own test generators, but it relies on other off-the-shelf tools, namely ACTS[5] and CASA[6].

[5] `http://csrc.nist.gov/groups/SNS/acts/`

[6] `http://cse.unl.edu/~citportal/`

Parameters and constraints are given in a unique file that contains the whole model. To formally describe a combinatorial problem, the user has to identify at least 2 parameters and their possible values.

Definition 4.5 (Constrained Model). *Let $P = \{p_1, \ldots, p_m\}$ be the set of parameters. Every parameter p_i can take a value from the domain $D_i = \{v^i_1, \ldots, v^i_{o_i}\}$. Every parameter has a name (it can have also a type with its own name) and every enumerative value has an explicit name. We denote with $Constr = \{c_1, \ldots, c_n\}$ the set of constraints. P and $Constr$ constitute the CIT model.*

CitLab adopts the language of propositional logic with equality and arithmetic to express constraints. To be more precise, it uses propositional calculus, enriched with the arithmetic over integers and enumerative symbols. As operators, it admits the use of equality and inequality for any variable, the usual Boolean operators for Boolean terms, and the relational and arithmetic operators for numeric terms.

Figure 7 shows the CitLab model of a simple washing machine consisting of 3 parameters. Users can select if the machine has HalfLoad, the desired Rinse, and the speed of Spin cycle. In the Constraints section there are two constraints: if HalfLoad is set then the speed of spin cycle cannot be High; if rinse is set to delicate, then HalfLoad must be true.

Note that the CitLab model in Fig. 7 is a possible variation of the CitLab model of a washing machine presented in Fig. 3a, representing the same system but with different value names for the spin, and a difference in the second constraint.

```
Model WashingMachine
Parameters:
Boolean HalfLoad;
Enumerative Rinse { Delicate Drain Wool };
Enumerative Spin { Low Mid High };
end
Constraints:
# HalfLoad => Spin != Spin.High #
# Rinse==Rinse.Delicate => HalfLoad #
end
```

Figure 7: An example CIT model of a washing machine.

4.2.2 Combinatorial Testing Policies

[87] In our previous work [88] we proposed to use combinatorial interaction testing techniques to verify the validity of CIT models, which is the approach we use in this work.

In particular, given a CIT model, we modify it according to one of the policies briefly described below. Next, we use one of the standard CIT tools to generate a test suite satisfying the modified CIT model.

We use the term “valid test" to denote the generated configuration that satisfies all the constraints of the original CIT model. Conversely, the term “invalid test" is used for a configuration that does not satisfy at least one of the constraints of the original CIT model. Words “test" and “configuration" are used interchangeably.

In classical combinatorial interaction testing, only valid tests are generated, since the focus is on assessing if the system under test produces valid outputs. However, we believe that invalid tests are also useful. In particular, a combinatorial model can be overconstrained, that is, it might restrict generation of test cases that are valid in the system, yet not be generated due to the constraints in the given model. Furthermore, critical safety systems should be tested if they failed safely in case an invalid configuration is entered. Moreover, creation of a CIT model for a large real-world software system is usually a tedious, error-prone task. Therefore, invalid configurations generated by the model at hand can help reveal constraints within the system under test and help refine the combinatorial model.

Two basic policies can be employed in combinatorial testing: UC (Unconstrained CIT) consists in generating the tests ignoring the constraints, and CC (Constrained CIT) is the classical testing policy that generates only tests satisfying the constraints. Aside from UC and CC, we consider three other policies, introduced in our previous work [88]:

Constraints Violating CIT - CV

This approach produces only the tests violating the constraints, i.e. the produced test suite contains every tuple of parameter values that makes at least one constraint false. This approach is complementary with respect to the CC in which only valid configurations are produced.

Combinatorial Union - CuCV

CuCV is the union of CC and CV, i.e. it covers all the desired parameter interactions producing valid configurations *and* all those producing invalid ones according to the given CIT model.

CIT of Constraint Validity - ValC

ValC requires the interaction of each parameter with the validity of the whole CIT model. That is, both tests that satisfy all the constraints will be generated as well as those that don't satisfy any of the constraints in the given CIT model. ValC is an approach that tries to balance the validity of the tests without requiring the union of valid and invalid tests.

4.2.3 Definitions

We assume that the combinatorial model specifies the parameters and constraints between them for a given software system. We are interested in checking whether this system specification correctly represents the software implementation. We assume that the parameters and their domains are correctly captured in the specification, while the constraints may contain some faults. Software model M belongs to the problem space while implementation of the software system S belongs to the solution space [153]. In our repair process, we assume that the model M may contain faults (in the constraints), as opposed to classical software testing, in which the implementation S is to be checked against a model that is considered correct.

Formally, given an assignment t that assigns a value to every parameter in P of the model M, we introduce two functions:

Definition 4.6. *Given a model M for a software system S, val_M is the function that checks if assignment t satisfies the constraints in M, while $oracle_S(t)$ checks if t is a valid configuration according to the system S.*

We assume that the oracle function $oracle_S$ exists. For instance, in case of a compile-time configurable system, we can assume that the compiler plays the role of an oracle: if the parameter assignment t allows compilation of the product then we say that $oracle_S(t)$ holds. We might enhance the definition of oracle by considering also other factors, for example, if the execution of the test suite completes successfully. However, executing

$oracle_S$ might be very time consuming and it might require, in some cases, human intervention.

On the model side, the evaluation of $val_M(t)$ is straightforward, that is, $val_M(t) = c_{1[P \leftarrow t]} \wedge \ldots \wedge c_{n[P \leftarrow t]}$.

Definition 4.7 (Conformance fault)**.** *We say that the constrained CIT model is correct if, for every t, $val_M(t) = oracle_S(t)$. We say that the model contains a* conformance *fault if there exists a t such that $val_M(t) \neq oracle_S(t)$.*

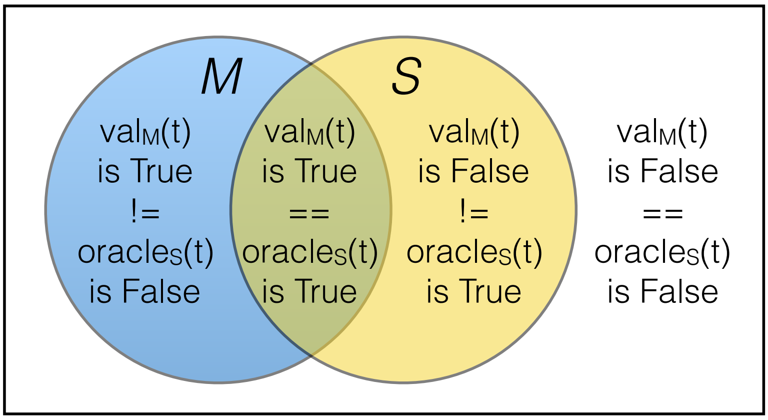


Figure 8: The space of test cases for system S and its model M. Failing test cases appear in the regions M/S and S/M, i.e., where $val_M(t) \neq oracle_S(t)$.

Figure 8 shows two different types of failing tests with respect to all possible parameter assignments (ignoring constraints). A conformance fault might occur if $val_M(t)$ is *True* and $oracle_S(t)$ is *False* (i.e., M/S in Figure 8). Another type of conformance fault occurs, when the model does not pass a test case that is allowed by the system, that is, if $oracle_S(t)$ is *True*, but $val_M(t)$ is *False* (i.e., S/M in Figure 8).

Example 4.5. *As an example, consider a system S of the washing machine modelled in Figure 7, and as model M the same model in Figure 7 without the first constraint: that conformance fault could be revealed by the following test t:*

Parameter Assignments			Function Evaluation	
Rinse	HalfLoad	Spin	$oracle_S$	val_M
Delicate	True	High	False	True

Table 4: An example test case triggering a conformance fault in the washing machine model.

Definition 4.8 (Combination)**.** *Let P be the set of all parameters in a given model. A combination $comb$ is an assignment of values to every parameter in a (non-empty) subset of P .*

Definition 4.9 (Failure-inducing)**.** *A combination $comb$ is failure-inducing if for every test t containing $comb$ (i.e., $comb \subseteq t$), $val_M(t) \neq oracle_S(t)$.*

Example 4.6. *Given the System S of the washing machine modelled in Figure 7, and its Model M presented in Figure 7 without the first constraint: a failure-inducing combination in this case would be HalfLoad and Spin=Spin.High, because for every possible configuration t, $val_M(t) \neq oracle_S(t)$, as shown in the table below:*

Rinse	HalfLoad	Spin	$oracle_S$	val_M
Delicate	True	High	False	True
Drain	True	High	False	True
Wool	True	High	False	True

Table 5: All tests containing the failure-inducing combination HalfLoad and Spin=Spin.High.

Definition 4.10 (Under-constraining and Over-constraining combinations)**.** *We say that a combination $comb$ (i.e., a partial assignment) is over-constraining the CIT model if for every assignment t containing $comb$, $val_M(t)$ = False and $oracle_S(t)$ = True. We say that a combination $comb$ (i.e., a partial assignment) is under-constraining the CIT model if for every assignment t containing $comb$, $val_M(t)$ = True and $oracle_S(t)$ = False.*

When a failure-inducing combination is found, one needs to modify the model M, i.e. *repair* the model, so that it faithfully represents the system S that it models.

The technique we present tries to repair the constraints of a combinatorial model whenever a failure-inducing combination is found.

There are two types of repairs, depending on the type of the failure-inducing combination:

- If a combination $comb$ is over-constraining, the model must be modified in order to include also the configurations identified by $comb$. This can be done by *relaxing* the constraints by adding $comb$ to each constraint that currently prohibits $comb$ by means of a Boolean *OR* (i.e., $Constr \longleftarrow Constr \vee comb$).

- If a combination $comb$ is under-constraining, the model must be modified in order to exclude $comb$ and this can be done by *strengthening* the constraints *Constr* by adding a further constraint $\neg comb$.

In order to fix the CIT model, we modify its constraints using the failure-inducing combinations found. If $comb$ is over-constraining the CIT model, we build a new constraint by allowing $comb$ to be added: $Constr \leftarrow Constr \vee comb$. If $comb$ is under-constraining, we build a new constraint by adding the negation of $comb$ to the old constraint: $Constr \leftarrow Constr \wedge \neg comb$.

Definition 4.11 (Relaxing and Constraining Sets)**.** *We call a* relaxing set *the set of all combinations* comb *that need to be added to repair the model M. We call a* constraining set *the set of all combinations* comb *that need to be removed to repair the model M.*

The aim of our approach is to remove all failure-inducing combinations and thus fix the constraints in the given CIT model. Visually, we want the two circles in Figure 8 to completely overlap, so that there are no more conformance faults between the model and the system.

Example 4.7 (Repair with a constraining set)**.** *Consider the washing machine system S, modelled in Figure 7. Let M be the model presented in Figure 7 without the first constraint: a possible failure-inducing combination is HalfLoad=True, Spin=Spin.High, which is True for val_M and False for $oracle_S$, as shown in Example 4.6. The constraint "! (HalfLoad & Spin==Spin.High)" would then be added to the constraining set and subsequently to the model M, thus repairing the faulty model.*

Example 4.8 (Repair with a relaxing set)**.** *Consider the washing machine system S, modelled in Figure 7. Let M be the model presented in Figure 7 with the first constraint changed to: "HalfLoad => Spin == Spin.Low": a possible failure-inducing combination is "HalfLoad=True, Spin=Mid". The model M would then be corrected by appending to all the constraints, the following combination: " | (HalfLoad & Spin==Spin.Mid)". In this case, the constraint causing the conformance failure is the first one of the model, which after the repair becomes "HalfLoad => Spin==Spin.Low | (HalfLoad & Spin==Spin.Mid)", which is equivalent to "HalfLoad => Spin!=Spin.High". We note here that since we do not use an exhaustive test suite in our approach (details of which are presented in Section 4.2.4), the failure-inducing combination derivation might be incorrect. Since we append* *the same constraint comb to all the constraints, we might potentially need to further repair the model.*

In order to measure the faithfulness of the given model M with respect to the system it models S, we introduce the following measure that we call the *failure index*:

Definition 4.12 (Failure Index)**.** *Given a model M of a system S, we define the* failure index *of M to be the number of (valid and invalid) configurations of M that fail. Formally:*

$$\mathsf{fi} = \frac{|\{t \in T | val_M(t) \neq oracle_S(t)\}|}{|T|}$$

where T is the given test suite.

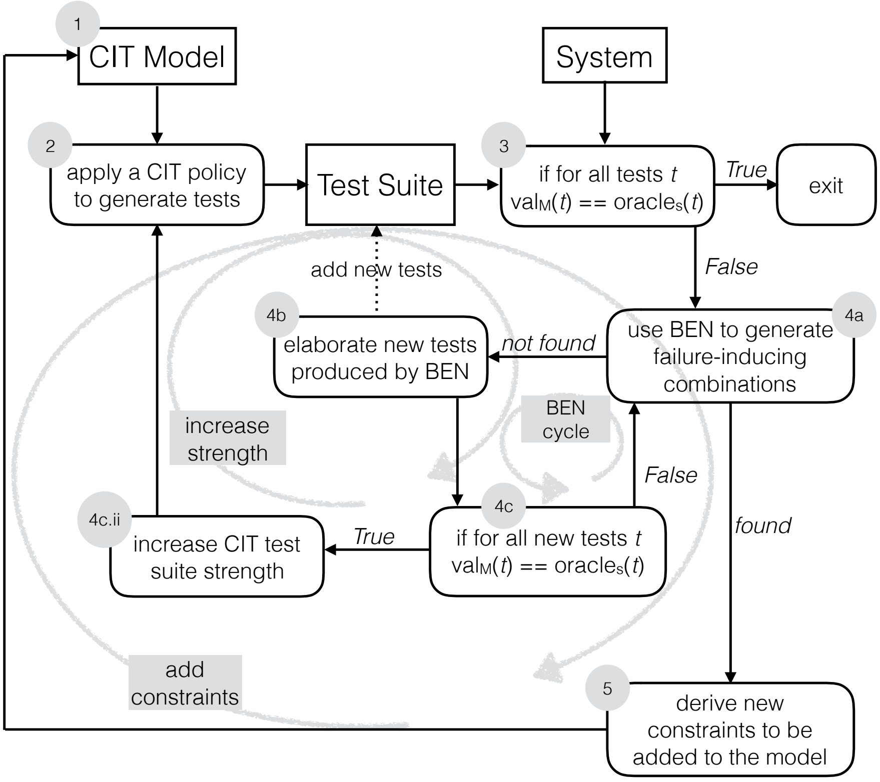

1
CIT Model
System
2
apply a CIT policy to generate tests
Test Suite
3
if for all tests t valM(t) == oracles(t)
True
exit
False
add new tests
4b
elaborate new tests produced by BEN
not found
use BEN to generate failure-inducing combinations
4a
increase strength
BEN cycle
False
found
4c.ii
increase CIT test suite strength
True
4c
if for all new tests t valM(t) == oracles(t)
add constraints
5
derive new constraints to be added to the model

Figure 9: The constraint repair process.

Taking the exhaustive test suite as T, we can compute an absolute failure index (afi) as measure of the quality of a model. However, computing the absolute failure index afi is infeasible in general. In the experiments, we will be able to compute the models afi by assuming that we know the exact model of the system. We use multi-valued decision diagrams (MDDs) [91] in order to calculate the number of tests that are permitted by the constraints in the model.

4.2.4 The constraint repair process

In this section we present the proposed process for constraint repair in CIT models. In our previous work [88] we devised a way of finding conformance faults. We use these techniques to find faults and extend them by proposing an automated way of fixing the faults found. Figure 9 presents an overview of the constraint repair process.

We first generate a set of test cases of given strength k based on one of the CIT policies described in Section 4.2.2. We evaluate each of the tests t using the function $val_M(t) = oracle_S(t)$. We mark each test for which $val_M(t) \neq oracle_S(t)$ as a failing one (since it reveals a conformance fault). We say that the test t passes, otherwise.

We input the generated test suite and the result of $val_M(t) = oracle_S(t)$ for each test t into a combinatorial testing-based fault localization tool called BEN [96]. BEN uses a heuristic approach to identify a set of failure-inducing combinations of given size (i.e., combinatorial strength) or larger. Given an initial test suite and the result for each test, it identifies a set of suspicious parameter-value combinations. Next, it proceeds in an iterative manner: it generates a set of new tests and queries the user for the result of these tests; after the user supplies the data, BEN identifies a new set of suspicious combinations and a new set of tests; the process is repeated until a set of failure-inducing combinations is found or all tests pass (in which case we can increase the strength of CIT testing)[7]. Further details about the heuristic approach implemented in BEN can be found in the following paper: [96]. We note that the problem of finding minimum failure-inducing combinations is a challenging task, since generation of an exhaustive test suite is often infeasible.

Once failure-inducing combinations are found, we modify the constraints of the original CIT model, as explained in Examples 4.7 and 4.8. We repeat the whole process until all test cases (generated by the CIT policy of choice and BEN) pass.

Our proposed constraint repair process proceeds as follows:

1. Start with a constrained CIT model containing a set of constraints C.
2. Derive a t-way CIT test suite according to one of the CIT policies described in Section 4.2.2.

[7] In our experiments we also found there are other cases when BEN terminates. These are, however, usually error states. We treat these cases the same way in which we deal with the ‘all test pass’ case, that is, we report that BEN did not find any failure-inducing combinations of given strength.

3. If for all test cases, the test result is the same according to the model and according to the system, then exit.
4. For all tests t such that $val_M(t) \neq oracle_S(t)$, mark t as a failing test. t is passing otherwise. Use BEN to derive failure-inducing combinations:
   a) Produce an initial set of suspicious combinations with new test cases using BEN.
   b) Add tests produced by BEN to the test suite.
   c) Mark the new test cases as failing or passing according to $val_M = oracle_S$.
      i. If all new tests pass and BEN has not detected any failure-inducing combinations, increase the test suite strength.
      ii. Otherwise, input the new set of tests to BEN.
   d) If BEN terminates and produces a set of failure-inducing combinations, exit BEN.
5. Modify the constraint set C based on the result produced by BEN:
   Given the set of failure-inducing combinations $comb$s, for each $comb$:
   a) If a failure-inducing configuration $comb$ occurs in test cases for which $val_M(t) = \mathit{True}$ and $oracle_S(t) = \mathit{False}$ (i.e., belongs to the constraining set), then $\mathit{Constr} \leftarrow \mathit{Constr} \cup \neg comb$.
   b) If a failure-inducing configuration $comb$ occurs in test cases for which $val_M(t) = \mathit{False}$ and $oracle_S(t) = \mathit{True}$ (i.e., belongs to the relaxing set), then for each $c_i \in \mathit{Constr}$, add $comb$ to c_i, that is, $c_i \leftarrow c_i \vee comb$.
6. Go back to point 1.

Note that this approach can be used to infer constraints by supplying an unconstrained CIT model to our tool.

The final constraints produced by our process can be further simplified. Since constraints in our model can be represented in Boolean form, we can use propositional logic rules. We leave this step as future work.

Another enhancement would be identification of constraints to which $comb$s from the relaxing set need to be added. However, identification of the set of constraints that violate a $comb$ is a non-trivial task. It can be reduced to the problem of finding minimum unsatisfiable sets in Boolean satisfiability solving (SAT). We leave this step as future work.

4.2.5 Experiments

In order to test our approach we conducted the following two experiments[8]. In the first experiment, we applied mutation to a set of models taken from the literature. In the second experiment, we used a configurable software system, namely Django, in order to test our framework on a real case study.

Mutation Analysis

We gathered a set of combinatorial models taken from several papers and applied our process to mutated versions of these models. In particular, for every model m in the benchmarks we derived a set M of mutants.

We have applied simple mutation operators to the constraints in the model under test in order to find mutants to be repaired. Our mutation operators are of 4 types: add a constraint that excludes a value of a parameter, remove an entire constraint, negate a constraint, and change a logical operator to another one (*AND* to *OR* etc.). Then, for each mutant m' in M, representing a possible faulty model, we applied our repair process by using the original model m as the oracle (to compute the $oracle_S$ function).

We considered all CIT policies presented Section 4.2.2. We applied our repair starting with combinatorial strength set to 1 and increase it up to 3 (i.e., we are only concerned with finding failure-inducing combinations of size 1,2 and 3). We included combinatorial tests of strength 1 since some faults can be already detected by a single parameter assignment.

Benchmarks

We used 5 case studies to evaluate our proposed approach:

1. The **WashingMachine** system, model of which is presented in Figure 7. It represents an abstract view of the human interaction with an *embedded system*.
2. **Concurrency** is a testing problem for real-life concurrent system presented in [180].
3. **Telecom** is a real-life telecommunication system presented in [180].

[8] All the experiments have been executed on a Linux PC with two Intel(R) i7-3930K CPU (3.2 GHz) and 16 GB of RAM.

4. **Aircraft** is a small Software Product Line (SPL) found in SPLOT repository and presented in [204].

5. **Libssh** is the combinatorial model for cmake of SSH library taken from [33].

		constraints		State space		validity	mut-
name	#var	#	CNF size	exp	#conf	ratio	ants
WashingM.	3	2	2^3	$2^1 3^2$	18	66%	18
Concurrency	5	7	$2^4 3^1 5^2$	2^5	32	25%	38
Aircraft	8	2	$3^1 4^1$	$2^7 3^1$	384	82%	25
Libssh	16	2	2^2	2^{16}	65536	50%	42
Telecom	10	21	$2^{11} 3^1 4^9$	$2^5 3^1 4^2 5^1 6^1$	46080	39%	116

Table 6: The benchmark data (for CNF size a^b means b clauses with a literals each.)

Table 6 presents various benchmark data: number of variables, number of constraints (including the number of clauses when converted to conjunctive normal form (CNF)), the size of the state space (the total number of possible configurations), the percentage of configurations that are valid (i.e. the ratio of valid configurations), and the number of mutants we generated for each model. Note that a low ratio indicates that there are only few valid configurations. We tried to collect models from different domains, with a good level of diversity (in terms of size, constraints, and so on) in order to increase the validity of our findings.

We ran each constraint repair process 10 times and report averages. We used the ACTS tool for CIT test suite generation.

Evaluation of Effort and Repair Capability by mutation

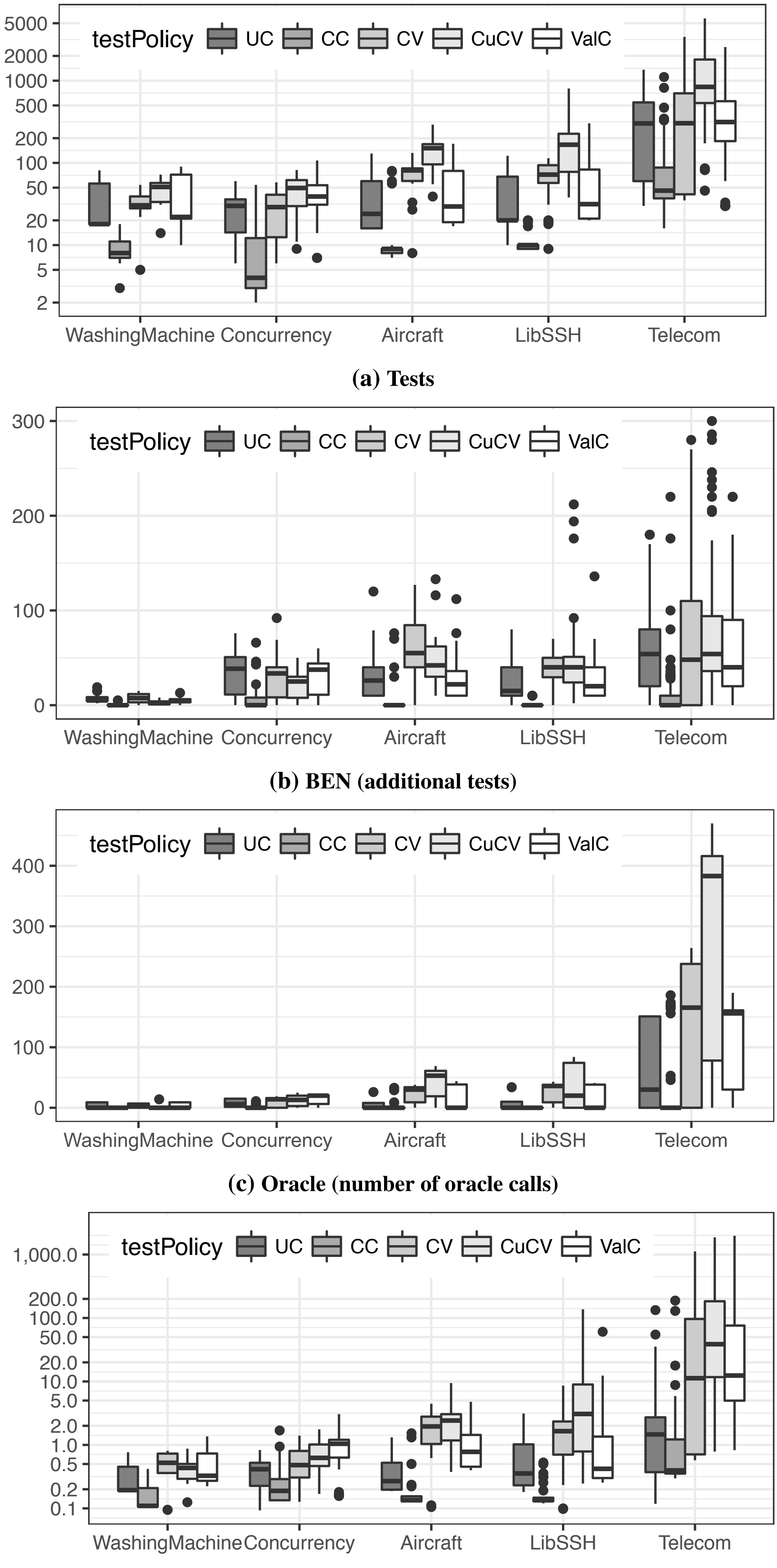
In order to evaluate our technique, we wanted first to measure the required *effort* and how it varies by changing the CIT policy. To measure the effort we used, for each mutant:

Tests The number of test cases are generated according to the given testing policy. Note that for a mutant, test case generation can be invoked several times. This can happen if, e.g., not all the tests pass and BEN is not able to find the failure-inducing combination and we have to increase the combinatorial testing strength.

BEN The number of additional tests BEN requires in order to isolate the failure-inducing combinations.

Oracle The number of times the oracle was called.

Time The total time required to fix a mutated model, including test generation, BEN invocation, and oracle evaluation.



(a) Tests

(b) BEN (additional tests)

(c) Oracle (number of oracle calls)

(d) Time (seconds)

Figure 10: Effort of the repair process

In this experiment, the time required by the oracle to evaluate a test is negligible, since we use the original model as an oracle. However, for real software systems a single invocation of the oracle may take several minutes. Thus the number of oracle calls is the critical factor as the evaluation of the function $oracle_S$ can be expensive, especially if human input is required.

We are also interested in assessing the *repair capability* of our approach. We use the following two measures:

TRM – Totally Repaired Models. The ratio of completely repaired models. In this experiment, we can check if a repaired model is completely fault free by querying whether it is equivalent to the original model. To check equivalence among combinatorial models we use an SMT solver [31].

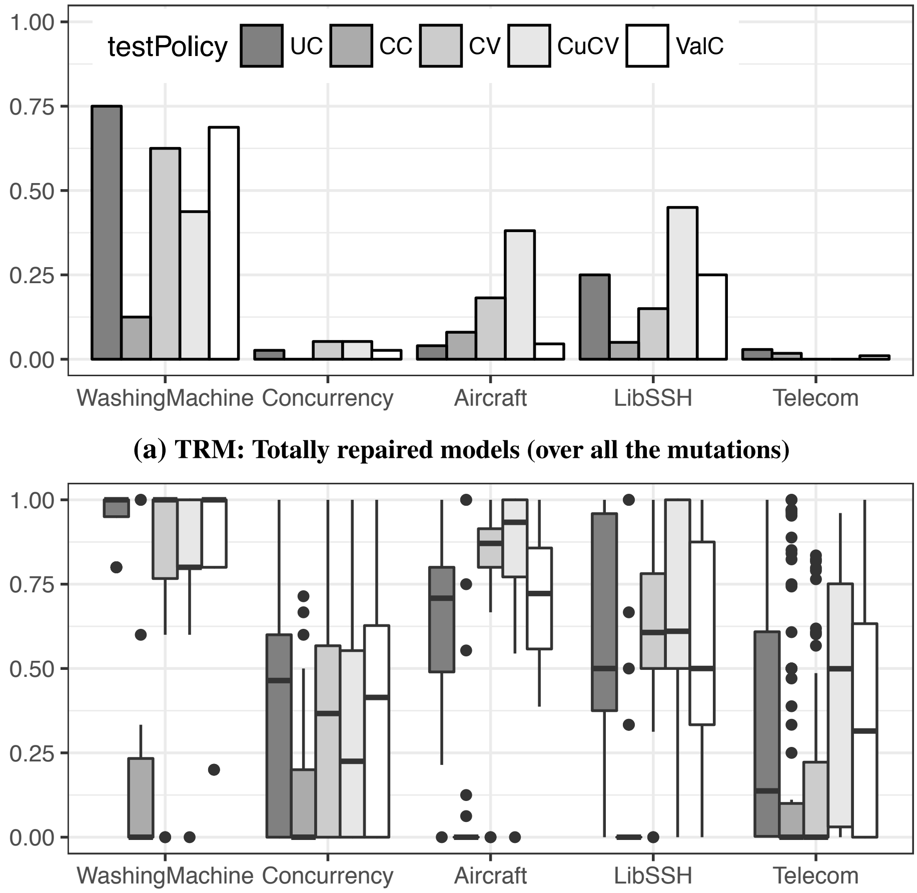
FID – Failure Index Delta. Even if a model is not completely repaired, we are interested in measuring how many conformance faults were repaired. We define FID $= (\mathsf{afi}_{init} - \mathsf{afi}_{final})/\mathsf{afi}_{init}$ in terms of the absolute failure index presented in Def. 4.12. FID represents the percentage of conformance faults that are repaired.

	Time (seconds)					Tests (Initial)					BEN (additional tests)					FID (%)				
name	UC	CC	CV	CuCV	ValC	UC	CC	CV	CuCV	ValC	UC	CC	CV	CuCV	ValC	UC	CC	CV	CuCV	ValC
WashingM.	0.3	0.2	0.5	0.4	0.5	31.5	9.6	29.7	45.0	36.3	6.6	0.6	6.5	2.5	4.6	95	21	81	82	86
Concurrency	0.4	0.3	0.6	0.8	1.1	27.8	11.3	29.1	47.0	41.9	33.0	8.0	29.9	20.7	30.8	36	13	33	29	36
Aircraft	0.4	0.3	2.0	2.9	1.3	41.1	18.1	74.0	144.5	51.6	33.3	8.6	59.5	50.4	31.5	62	14	79	85	60
Libssh	0.7	0.2	1.8	77.9	2.6	44.0	10.8	66.5	187.5	54.6	23.4	1.0	35.3	46.3	26.0	57	10	63	67	50
Telecom	4.5	4.0	83.0	180.3	107.7	352.1	111.8	532.1	1287.0	435.9	56.1	11.5	70.0	77.6	60.1	25	16	12	35	31

Table 7: Means of the quantities over all the mutations

Figures 10 and 11 show respectively the effort and the repair capability of each policy for every model in the benchmark set. The experimental results are also summarized in Table 7.

We can observe that the number of tests generated at the beginning of each repair process (Fig. 10a) and the number of tests BEN requires in order to find failure-inducing combinations (Fig. 10b) are correlated. If one starts with small test suites, BEN will require fewer tests (and it likely identifies fewer failure-inducing combos). In terms of time (Fig. 10d), our process is rather fast for small models, but also for the biggest models, like Telecom, it terminates on average in less than 10 seconds for weak policies (like UC and CC), and in any case in less than 10 minutes when using strong policies like CuCV and ValC. Regarding the repair capability, the TRM is rather small for big specifications, and using



(a) TRM: Totally repaired models (over all the mutations)

(b) FID: Failure Index Delta

Figure 11: Repair capability

more powerful testing policy (like CuCV) does not help much (Fig. 11a). Using our process to obtain a completely bug-free model seems unrealistic for big models: CIT likely provides an insufficient coverage for this. However, if we consider the FID in Fig. 11b, we can say that almost always our technique and CIT improve the model except when using CC. The overall minimum average for FID when using strong policies like CuCV and ValC is 29%. For big models, there are cases in which we are unable to improve the constraints, but on average we can still remove around 35% of the faults with CuCV and ValC. As already observed in [88], the CC policy (that generates only valid tests) detects fewer conformance faults, and for this reason has the lowest FID. The average FID over all the mutants is around 37%. The average FID when using CuCV is 49%: half of faults can be repaired on average if the user chooses this policy.

Repairing the model of Django parameters

Django is a free and open source web application framework, written in Python, that supports the creation of complex, database-driven websites, emphasizing reusability of components[9]. Each Django project has a configuration file.

It is loaded at *launch time*, i.e. every time the web server that executes the project (e.g. Apache) is started. Among all the possible configuration parameters, we selected and modelled one Enumerative and 23 Boolean parameters. We implemented an automated oracle which returns true if and only if the HTTP response code of the Django project homepage is 200 (HTTP OK). This oracle invocation is costly in terms of time, since editing the settings of the Django project, starting the Apache server, and waiting for its response, requires around 6 seconds.

We applied the constraint repair process described in Section 5.2.2 on a default Django start project running on Apache server (without SSL configuration) called at address *localhost*. We started from a model (Django0) with an empty set of constraints because we thought that Django was accepting (and amending if necessary) every possible configuration, and secondly, to test the effectiveness of this technique to *infer* (and not only *repair*) configuration constraints: we call this first experiment "Inference from Django0". At the end of the process, we obtained different set of constraints depending on the policy. For all the policies, we executed the repair process with test suite strength from 1 to 3.

We noticed that the application of the testing policy CV produces an empty test suite at the beginning, and an empty set of constraints at the end of the process, because the initial Django model contains no constraints, and in particular there are no test cases violating constraints. So we ignored this policy in the experiments.

We manually derived the constraints, based on the Django documentation about configuration files, and by looking at the results of a few test cases. As a second experiment, we applied the constraint repair process to this manual model (Manual), in order to improve its conformance with the real Django system: we call this second experiment "Repairing Manual".

[9] https://www.djangoproject.com/

	Policy	Tests	BEN	Oracle	Time	C[a]	P[b]	fi
Django0 (empty model)						0	0	0.789
inference from Django0	UC	36	20	32	201s	5	6	0.325
	CC	173	60	154	1671s	8	9	0.254
	CuCV	320	54	335	3570s	9	12	0.081
	ValC	288	80	124	1453s	17	19	0.204
Manual						3	4	0.033
repairing Manual	UC	63	30	51	435s	3	4	0.033
	CC	63	30	70	436s	3	4	0.033
	CuCV	202	30	167	1354s	4	4	0
	ValC	118	30	92	564s	4	4	0

[a] C: Total number of constraints in the model

[b] P: Total number of parameters involved in the constraints (without considering duplicates)

Table 8: Django inference and repair results

Results of this experiment are summarized in Table 8, which reports the number of tests generated by the testing policy, the number of additional tests required by BEN, the number of oracle calls, the time taken for the whole process, the number of constraints of the final model, the number of parameters involved in the constraints, and the failure index. To compare the different policies in terms of fault detection capability, in this case, we cannot rely on the absolute failure index, since the actual model of the Django system remains unknown. We can, however, compute the failure index (see Definition 4.12) where the test suite T is the union of all the test suites we generated for all the policies.

With regards to *inferring* constraints, the CuCV policy generates the biggest test suite and requires also the largest amount of time: around 1h for constraint inference from Django0 (including oracle invocation). It has also the lowest fi (8%), which however is not zero: no inferred model is completely fault-free. The UC policy is the worst in terms of failure index data, but it is the fastest in our experiment.

ValC produces a model with the largest number of constraints and the failure index (20%) is the second lowest. However, if we compare the failure index of Diango0 (79%), we can say that all the testing policies can substantially improve the initial empty model. In conclusion, all the models contain a reasonable number of constraints and although our best policy (CuCV) does not beat the manual model we have developed (by using our best efforts), it can produce a rather correct model.

Regarding the problem of *repairing* the initial manual model, two out of four testing policies (CuCV and ValC) are able to completely repair the manual model (at least when considering the failure index based on the set of generated tests), while the UC and CC policies did not improve the initial model. For constraint repair, the number of tests and the time required is generally lower (except for UC) than for the process of inferring constraints. We conclude that the availability of an initial model, even if it is not completely correct, makes possible to obtain a fault-free model for a real system when strong testing policies are applied.

Figure 12 reports the constraints for Django0 (the initial model with empty set of constraints), Manual (the model *inferred* manually), and for the repaired models produced using CuCV and ValC (starting from Manual). The constraints obtained in the final models are still readable, making a human inspection possible for further modifications.

```
Django0
# true #
```

```
Manual (constraints "inferred " manually)
# !PREPEND_WWW #
# !SECURE_SSL_REDIRECT #
// if not DEBUG, "localhost" must be allowed
# !DEBUG =>
(ALLOWED_HOSTS==LOCALHOSTIP or ALLOWED_HOSTS==LOCALHOST) #
```

```
CuCV (repairing Manual)
# !PREPEND_WWW #
# !SECURE_SSL_REDIRECT #
# !DEBUG =>
ALLOWED_HOSTS==LOCALHOSTIP or ALLOWED_HOSTS==LOCALHOST #
# ALLOWED_HOSTS==IP =>
!DEBUG or PREPEND_WWW or SECURE_SSL_REDIRECT #
```

```
ValC (repairing Manual)
# !PREPEND_WWW #
# !SECURE_SSL_REDIRECT #
# !DEBUG =>
ALLOWED_HOSTS==LOCALHOSTIP or ALLOWED_HOSTS==LOCALHOST #
# ALLOWED_HOSTS==IP => !DEBUG or SECURE_SSL_REDIRECT #
```

Figure 12: Django models obtained by repairing the manual model using different testing policies

4.2.6 Related Work

The problem of finding and fixing constraints in combinatorial models has been addressed in the field of software product lines. Henard et al. [107] assumes existence of valid configurations of the given software system and use a SAT solver to randomly generate

> configurations for an existing feature model of the system. Once a conformance fault is found, a constraint is added, removed or altered with certain probability. The mutated feature model is compared against the original using a pre-defined fitness function and the best of the two is kept for future evaluation.
>
> In contrast to Henard et al.'s work, we derive test cases in a systematic fashion using combinatorial interaction testing. Moreover, by applying policies presented in [88], we are able to generate invalid configurations. Therefore, in contrast to the work by Henard et al., we are able to find conformance faults caused by an over-constrained model. Finally, we use a systematic approach to fix the model by utilizing BEN to find the minimum fault-inducing configurations.
>
> Some of our policies could be improved by integrating the *testing with negative values* technique described in [67], which disallows two *invalid values* to coexist in one test case (negative test cases), assuming that every configuration containing at least one invalid parameter is invalid. We did not consider that technique because it does not apply to the more general case (which is quite common) of invalid configurations originating from *t-way* parameter interactions. A possible future work is to modify our CV and ValC policies to include only configurations that make *exactly* one constraint false.
>
> Arcaini et al. [33] used mutation testing to detect and fix conformance faults by distinguishing a given feature model from its mutants.

The problem of modelling and testing the configurations of complex software systems is non-trivial. There has been much research done in extracting constraints among parameter configurations from real systems. For instance, the importance of having a model of variability and having the constraints in the model aligned with the implementation is discussed in [153]. However, in that paper, authors try to identify the source of configuration constraints and to automatically extract the variability model. Our approach is oriented towards the validation of a variability model that already exists. Moreover, they target C-based systems that realize configurability with their build system and the C preprocessor. A similar approach is presented in [193], where the authors extract the compile-time configurability from its various implementation sources and examine for inconsistencies (for example, dead features and infeasible options). We believe that our approach is more general (not only compile-time and C-code) and can be complementary in validating and improving automatically extracted models.

Testing configurable systems in the presence of constraints is tackled in [63] and [168]. In these papers, authors argue that CIT is a very efficient technique and that constraints among parameters should be taken into account in order to generate only valid configurations. This allows to reduce the cost of testing. Also in [57], authors showed how to successfully deal with constraints by solving them using a constraint solver such as a Boolean satisfiability solver (SAT). However, the emphasis of that research is more on testing of the final system not its combinatorial model. CIT is also widely used to test SPLs [167].

In SPL the validation and extraction of constraints between features is generally given in terms of feature models (FMs). Synthesis of FMs can be performed by identifying patterns among features in products and in invalid configurations and build hierarchies and constraints (in limited form) among them. For instance, Davril et al. applied feature mining and feature associations mining to informal product descriptions [70]. There exist several papers that apply search based techniques, which generally give better results [104, 136, 83, 138]. However, checking and maintaining the consistency between a SPL and its feature model is still an open problem.

4.3 Using Iterative Constraint Repair to Detect XSS Vulnerabilities

In this Section, we apply model repair to a software security scenario. We consider the case where a knowledge base consists of interactions among parameter values in an input parameter model for web application security testing.

The input model gives rise to attack strings to be used for exploiting XSS vulnerabilities, a critical threat towards the security of web applications. Testing results are then annotated with a vulnerability triggering or non-triggering classification, and such security knowledge findings are added back to the knowledge base, making the resulting attack capabilities superior for newly requested input models. Our approach is an iterative process that *evolves* (i.e., *repairs*) an input model for security testing. Empirical evaluation on six real-world web application shows that the process effectively evolves a knowledge base for XSS vulnerability detection, achieving on average 78.8% accuracy.

One major threat to web applications is posed by Cross-Site Scripting (XSS), which continues to be included in the OWASP Top 10 most critical web application security risks [86]. Security testing is a vital and expensive part of the software development lifecycle. An effective testing technique applied to security testing is *Combinatorial Testing* (CT), for the capability of detecting failures and fail conditions with a small amount of tests that need to be executed compared to the whole input space [186]. Given a discrete finite model of the *system under test* (SUT), made of parameters with a finite list of possible values, called *input parameter model* (IPM), and given an interaction strength t, CT creates a test suite guaranteeing the appearance of all t-way interactions of parameter values, for any selection of t parameters [128]. In the case for testing for exploiting XSS vulnerabilities, the

IPM specifies an attack grammar and is also called *(abstract) attack model*. The aim of this work is to present a way to evolve knowledge bases for security testing. In our approach, the evolution of a knowledge base consists in the integration of learned constraints, using BEN [97], into the IPM. In particular, the contribution of this paper consists in an automated technique to detect all the conditions under which vulnerabilities are triggered, by using combinatorial testing. Evaluation shows that the process is able to evolve the knowledge base to achieve, on average over all the benchmarks, 78.8% accuracy, in 14 minutes computation time.

The rest of the section is structured as follows. In Sect. 4.3.1 we give basic definitions, in Sect. 4.3.2 we discuss our proposed process, and Sect. 4.3.3 presents the results of our case study experiments. We provide a brief overview over related work in Sect. 4.3.4, and the conclusions are discussed at the end of the chapter, in Sect 4.4.

4.3.1 Preliminaries

> *[93]* In the course of *combinatorial security testing* (cf. [186]), attack models have appeared in the form of a BNF grammar, e.g. [48], [187]. In this paper, we will follow this established terminology for designing XSS attack models to be used in conjunction with combinatorial methods. We denote with K_i a knowledge base at time i, encoded as an abstract attack model (IPM, see Fig. 13). Given an IPM, an abstract test case f is a particular assignment of values for its parameters. An abstract test suite is used to derive a concrete test suite, where abstract test cases are being translated into concrete XSS attack strings via a translation function τ. For example, given the following abstract test case:
>
> $$(\mathsf{JSO} = 2, \mathsf{WS} = 1, \mathsf{INT} = 3, \mathsf{EVH} = 2, \mathsf{PAY} = 2, \mathsf{PAS} = 5, \mathsf{JSE} = 7)$$
>
> for each parameter, the respective integer value corresponds to a concrete parameter value (i.e., a string), and the translated concrete test case is obtained by concatenating all these strings together in the order given by the IPM:

```
<script> onError= alert(1)')'\>
```

```
Model wavsep_xss

Parameters:
JSO: { P1 P2 P3 P4 P5 P6 P7 P8} PAY: { P1 P2 P3 P4 P5 }
INT: { P1 P2 P3 P4 P5 P6 P7 P8 P9 P10} EVH: { P1 P2 P3 }
PAS: { P1 P2 P3 P4 P5 P6 P7 P8 P9 P10} WS: Boolean
JSE: { P1 P2 P3 P4 P5 P6 P7 P8 P9}

Constraints:
# ! JSE==JSE.P2 # # ! JSE==JSE.P3 # # ! JSE==JSE.P4 #
# ! INT==INT.P9 # # ! (JSO==JSO.P2 && WS==false && PAS==PAS.P7) #
...
```

Figure 13: Knowledge base K_3 for NavigateCMS: abstract attack model (initially it had no constraints), with detected XSS vulnerability constraints, in CTWedge

The resulting string can be submitted against the SUT, and a boolean function `orac` decides if the outcome of the execution of the translated test case $\tau(f)$ against the SUT triggers an XSS vulnerability or not. We denote also the function $eval(f) := oracle(\tau(f))$, that is $true$ if the test case f triggered an XSS vulnerability (i.e., f is a *vulnerability triggering* test case). The generated test vectors aim at producing valid JavaScript code when these are executed against SUTs. A description of parameters that appear in the attack model is mentioned in [48, 92, 187]. At any time point i, the knowledge base K_i may be used to create test cases, and to classify an abstract test case as either vulnerability-triggering or not, depending on whether the *constraints* are satisfied. This capability of the knowledge base is denoted as a $model$ function, which takes as input an abstract test case, and gives as output a “best guess” that may or may not be correct w.r.t. the actual result of the function $eval$. The attack model initially contains only combinatorial parameters and no constraints. During the process, the knowledge base is enriched by the conditions used to identify the vulnerabilities.

To put this problem into a formal setting, a *knowledge base fault* occurs when a test f is classified as non-vulnerable in the model ($\neg model(f)$), despite it actually triggers a vulnerability in the SUT ($eval(f)$) (**False Negative**: it entails a loss of potentially valuable information for fixing the vulnerability); or when a test is being marked as vulnerability-triggering in the model ($model(f)$), despite it does not trigger a vulnerability ($\neg eval(f)$) (**False Positive**: it triggers a false alarm, and the programmers may consequently waste effort in fixing pieces of code that did not trigger any vulnerability). When such a discrepancy is fixed by updating the model function, we say that the knowledge base evolves.

Since in our experiments it yielded better results, we decided to consider the convention for which the initial model function considers any test to be non-vulnerability triggering; and during the process constraints are added in order to identify and subsequently exclude all the tests that do not trigger any vulnerability. We call this convention *pessimistic* approach, in contrast with the *optimistic* one in which initially any test is vulnerability-triggering, and constraints are added to isolate the tests that actually trigger some vulnerability.

Let us complete some notation for combinatorial analysis. A combination c is an assignment (i.e. configuration) on a subset $Dom(c)$ of all the possible parameters P in the attack model, such that $Dom(c) \subseteq P$. We call *size* of the combination the cardinality of $Dom(c)$. A combination c identifies a set of tests: c *represents* a test f if all the parameters in c are also present in f, associated to the same values. Formally, $c \subseteq f : \forall p \in Dom(c), f(p) = c(p)$. A combination c is *suspicious* in a test set $F \subseteq \Gamma$ if c represents *only failed tests in* F. Formally, $\forall f \in F : c \subseteq f \rightarrow model(f) \neq eval(f)$. For the purposes of this work, we assume that the constraints are in conjunctive relation among each other.

4.3.2 Process for Model Evolution

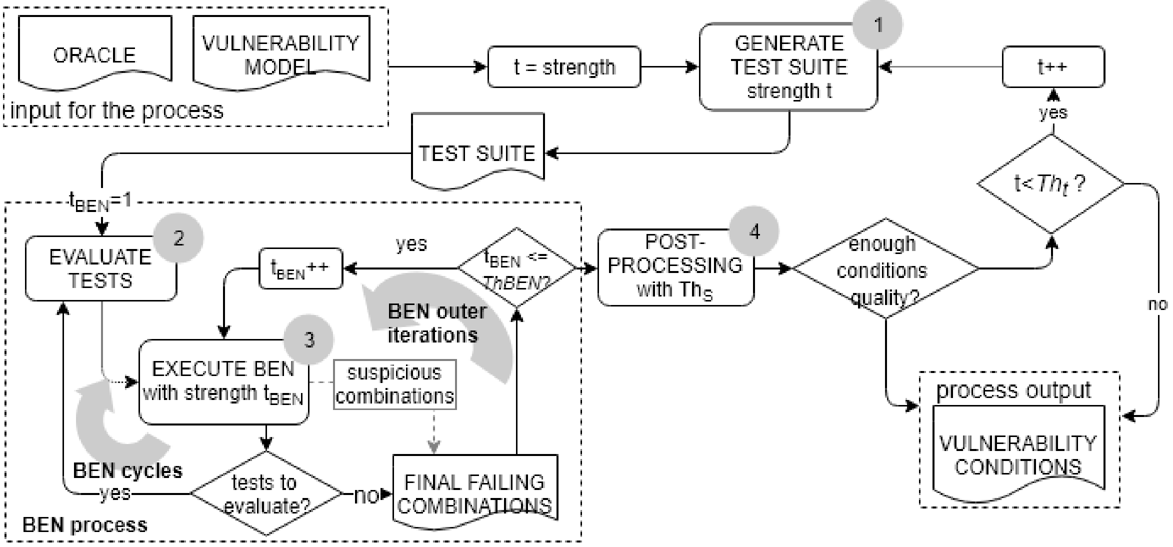


Figure 14: Condition detection meta-process

Fig. 14 shows an overview of our process to automatically evolve an abstract attack model initialized without constraints, to detect conditions that trigger XSS vulnerabilities. The process proceeds according to the following steps:

1. From a defined initial interaction strength t, derive a t-way test suite.

2. Mark the test cases as failing or passing according to the current model and the evaluator. If all the tests pass and Th_t is not yet reached, increment the strength $t \leftarrow t + 1$, and go to point 1. Otherwise, evaluate the tests. Internally, the evaluator executes the translation function τ to obtain the concrete test string to insert in the *reflection* URL to query the SUT. PhantomJS[10] then analyzes the HTML code returned by the SUT for a specific *target function* (the `alert()` function in our case[11]) that was included as payload. If any of the target functions were executed, the injection was successful; if the page loads normally and does not produce any errors, the injection is deemed unsuccessful. Lastly, if JavaScript errors are observed on the page, it is likely that some content was injected, but not in a form that constitutes a usable injection (thus resulting in incorrect syntax). Our current approach regards these test cases as failing, but future evolutionary approaches might take advantage of this particular classification.

3. Pass the evaluated test suite to BEN [97] to derive suspicious combinations, together with their *suspiciousness* level. We call BEN multiple times specifying the size t_{BEN} of the suspicious combinations to detect[12], from 1 to Th_{BEN}. BEN may also ask for a few additional tests (up to 10 tests at a time: *inner BEN cycles*) to reduce the amount of suspicious combinations and improve the accuracy of the computed *suspiciousness* levels.

4. The suspicious combinations from BEN are then translated into a set of constraints for the current knowledge base K_i, by negating the corresponding boolean expression (obtained by putting the assignments in conjunction) of every combination whose *suspiciousness* value is above the threshold Th_S.

Due to low accuracy in the detected suspicious combinations, we noticed that K_i often results to be a contradiction, which is normally not the case in real-world systems. Therefore, we *post-process* the constraints by computing the *unsat-cores* of K_i and removing all such clauses starting from the *least-suspicious* constraints (according to BEN), until K_i is not a contradiction any longer. The process eventually quits if either the user is *satisfied* with the quality of K_i[13], or the threshold Th_t is reached. Otherwise, increase t and go to point 1.

[10] PhantomJS (http://phantomjs.org/) is a headless browser environment enabling introspection of events such as network requests, document edits and JavaScript errors.

[11] in theory, any valid JavaScript functions that will not be called during the normal operation of the SUT can be chosen instead.

[12] note that t_{BEN} is different from the strength t for generating the initial test suite

[13] in this case, we believe that the suspiciousness average and standard deviation could be useful indicators of the F_1 score that the currently inferred model may have

4.3.3 Experiments

The process has been implemented in Java using `CTWedge` [89] to represent and update attack models, `ACTS` [216] to generate combinatorial test suites of a defined strength, and `BEN` [97] as a tool to detect suspicious combinations and compute suspiciousness. Experiments were executed on a PC with Intel i7 3.40GHz processor and 16 GB RAM. We run the process on six real web-applications: four are part of the WAVSEP[14] project, and two are open source content management systems: MiniCMS and NavigateCMS. Each SUT receives over HTTP one GET parameter which is rejected on the page in different contexts, and might optionally be altered by a specific sanitization function. Tab. 9 shows, for each SUT, the respective *vulnerability ratio*, i.e., the ratio of tests that triggered an XSS vulnerability ($eval(f)$) out of the total number of tests executed, that, given the practical infeasibility of the exhaustive test suite, we compute on all the tests generated up to strength t=5 (42830 tests).[15]

SUT ID	SUT name	Reflection site	vulnerability ratio (t=5)
1	Tag2HtmlPageScope	<body>$input</body>	17.08 %
2	Tag2TagStructure	<input type="text" value="$input">	4.06 %
3	Event2TagScope	<img src="$input">	4.63 %
4	Event2DoubleQuotePropertyScope	<img src="$input">	3.45 %
5	MiniCMS	/mc-admin/page.php?date=$input	80.2 %
6	NavigateCMS	/navigate.php?fid=$input	60.0 %

Table 9: XSS reflection sites on WAVSEP benchmarks

To assess the quality of the evolved knowledge base from our method, we use the typical metrics of information retrieval: in particular, *precision* ($\frac{TP}{TP+FP}$), and *recall* ($\frac{TP}{TP+FN}$) give a measure of how the process isolates true positives, $accuracy$ gives an overall ratio of correctly classified tests, and the F_1 score is considered to be a good candidate synthesis index of the inferred model's quality. For the experiments, we set the parameters of the process as follows:

[14] WAVSEP: Web Application Vulnerability Scanner Evaluation Project, `https://github.com/sectooladdict/wavsep`.

[15] The tests suites were generated using the IpoF algorithm, implemented in ACTS

- $Th_{BEN} = 3$. We limited the size of detected suspicious combinations as the BEN process becomes too slow when computing suspiciousness of the too many combinations of size 4 (or larger) on these attack models.
- t, the initial strength of test suite, is set to 2.
- $Th_t = 4$. We limit the maximum strength of the initial test suite since even t=5 (about 36000 tests) would make the BEN process too slow.
- $Th_S = 0$, as we want all the suspicious combinations to be considered.

	sut	time (s)	constraints	suspiciousness avg. ± s.d.	accuracy	precision	recall (TPR)	specificity (TNR)	F_1 score
Th_t=4	1	246	146	0.254 ± 0.0288	72.6	32.5	55.7	76.1	41.0
	2	141	38	0.362 ± 0.00753	93.3	32.6	59.5	94.8	42.1
	3	1437	794	0.137 ± 0.0456	87.4	15.7	39.7	89.7	22.5
	4	1088	551	0.136 ± 0.0483	89.0	13.7	42.3	90.6	20.7
	5	1445	623	0.342 ± 0.0287	78.3	87.0	85.8	48.3	86.4
	6	679	80	0.344 ± 0.0147	52.0	66.6	40.0	70.0	50.0
	avg	839	372	0.263 ± 0.0289	78.8	41.4	53.8	78.3	43.8
Th_t=3	1	9.3	644	0.196 ± 0.0423	41.4	19.8	79.8	33.5	31.8
	2	4.2	164	0.224 ± 0.0874	78.9	14.2	83	78.7	24.2
	3	28.5	764	0.207 ± 0.0628	75.4	12.9	75.7	75.3	22
	4	9.3	123	0.125 ± 0.0379	72.3	8.53	73.7	72.2	15.3
	5	58.5	1340	0.318 ± 0.0268	80.1	80.2	99.9	0.306	89
	6	37.6	2460	0.296 ± 0.0245	62.2	61.8	97.1	9.91	75.5
	avg	24.6	915	0.228 ± 0.0470	68.4	32.9	84.9	45.0	43.0

Table 10: Quality metrics for the inferred models ($Th_{BEN} = 3$**, and** $Th_S = 0$)

Test suites for interaction strengths $t \in \{3, 4\}$ had 900 and 7200 test cases, respectively. For each SUT, Tab. 10 reports the number of constraints included in the inferred model, the average suspiciousness with its standard deviation, and the accuracy, precision, recall, specificity, and F_1 score of the final model, computed over all tests up to strength 5, as for the *vulnerability ratio* in Tab. 9.

RQ1: *What is the quality of the model obtained by the approach?*

We observe that the inferred model achieves an average accuracy of 78.8%, with a maximum of 93.3%. Precision has an average of 41.4%, ranging from 13.7% to 87%, and recall (54% on average) is higher than precision. F_1 score is on average 43.8%, with a maximum of 86.4% on SUT5. With relatively few tests (t=4 out of 7 parameters), the final model is of good quality, but not completely accurate. We can also observe that F_1 is proportional to the vulnerability ratio of the SUT.

RQ2: *How does the quality of the inferred model vary depending on Th_t?*

By increasing Th_t from 3 to 4, the time taken, as expected, increases, while the number of constraints, in most cases, decreases, meaning that with more tests our process is able to describe the vulnerability conditions with fewer constraints. On average, although recall decreases, both precision, accuracy and specificity increase, and F_1 slightly increases. This means that the classification improves.

RQ3: *Which is the computational effort of the proposed process?*

Tab. 10 also reports the total execution time, excluding the actual test execution, as test results are cached, except for the few tests (30 at most) that BEN may ask during the process. For the first two SUTs, the process takes less than 250 seconds to complete, but up to 24 minutes are needed for SUT5 with $Th_t = 4$. Most of the computation time is used internally by BEN; by limiting to 3 the strength of the initial test suite, the total time is always below 1 minute for every SUT.

RQ4: *How does variations of Th_S affect model quality?*

The highest F_1 score is achieved with low values of Th_S (see Fig. 15), except for SUT3 and SUT4, for which a $\overline{Th_S} \simeq 0.13$ achieves the maximum F_1 score. However, we can notice that at least around 25% of the constraints (starting from the least suspicious ones) can be removed with negligible impact on the final F_1 score.

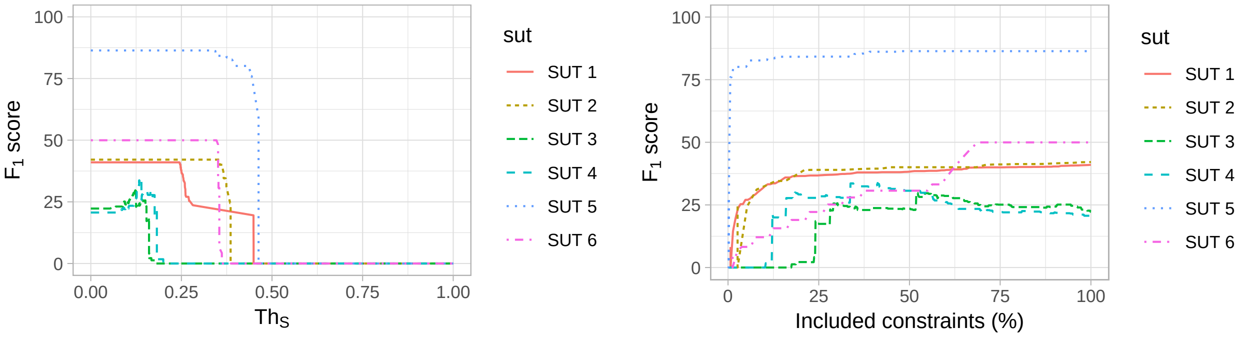


Figure 15: Achieved F_1 score of final model by varying Th_S, when $Th_t = 4$

4.3.4 Related Work

XSS vulnerability detection is not a novel topic in computer science research. Duchene et al. [73] used model based testing and fuzzing to discover XSS vulnerabilities; Melicher et al. [150] proposed improvements on using the DOM model to generate and detect XSS attacks; Simos et al. [187] proposed a combinatorial approach to find attack vectors that trigger XSS vulnerabilities; Jia et al. [115] used machine learning and hyper-heuristic search to improve combinatorial tests; Temple et al. [196] proposed a machine-learning approach to infer constraints among parameters that, although not sound, achieves high precision (about 90%) and recall (80%). Although these works use model-based testing, the usage of combinatorial testing for XSS vulnerability detection to *classify* vulnerabilities based on the input and describe *completely* the vulnerability space of a part of a web application, evolving a *knowledge base*, is the main novelty of our approach. The first phase of BEN [97] as a failure-inducing combination detection and ranking tool has been used by Gargantini et al. [87] to repair constraints in combinatorial models, evaluating different test generation policies.

4.4 Conclusion and Future Work

In this chapter, we presented an approach that extends CIT and aims to (1) automatically check the validity of the configurability model of the system under test, (2) automatically finding and fixing faults in models of parameter configurations of software systems, and (3) evolve an attack model to include conditions among input parameters that trigger XSS vulnerabilities in web applications.

For checking the validity of the combinatorial model, we devised four original policies that can help software testers discover faults in the model of system configurations as well as faults in the software implementation that the model describes [88] Several experiments conducted show the efficacy of our approach. We confirm that constraints play an important role in configurability testing, but the experiments show that also invalid configurations should be considered in order to avoid some problems (like over-specification) and to detect a wider range of faults. Our experiments suggest that techniques including both valid and invalid tests (as CuCV) have a better fault detection capability than techniques including only valid (as CC) or invalid tests (as CV). However, producing invalid tests may be not feasible. In these cases we would suggest the tester to use CCi instead of UC and CC. The experiments suggest that CCi is not very expensive and it offers a superior fault detection capability.

The techniques presented should significantly help software developers in the modeling and testing process of software systems configurations.

We then used these novel CIT policies in an iterative process for finding and fixing conformance faults [87]. This process can help software testers discover faults in the model of system configurations as well as faults in the software implementation that the model describes. We call these conformance faults. We then conduct several experiments on five software systems to validate our approach. We show that we can successfully repair existing combinatorial models. Furthermore, we also show how our approach can be utilized to derive constraints between parameters of large complex software systems. The technique presented should help software developers derive and fix combinatorial testing models by automating this often purely manual and thus time-consuming and highly error-prone task.

As third step, in [93], we introduce the notion of *suspicious combination*, that allows to apply the iterative process devised in [87] to the detection of XSS vulnerability-triggering conditions among input parameters. Suspicious combinations are combinations whose appearance in a test vector would trigger a discrepancy between the *best-guess* of the current model, and the actual outcome when executed against the SUT. Identification of constraints among XSS attack parameters helps to better understand the root cause of an XSS vulnerability and provides insights about how to fix a flawed sanitization function. As future work, we plan to improve the process by reducing the required tests, using information from previous step, and evaluating alternatives to BEN, such as MixTgTe [28]. We believe that this approach can be extended to other security vulnerabilities related to sanitization functions, and to detect discrepancies between a functional system specification and its implementation. Another direction is to further simplify the detected constraints, to reduce them in number and present them to the user in a more readable way, applying methods such as the ones in [29].

Chapter 5. Repair of Feature Models

Feature models are a widely used modeling notation for variability management in software product line (SPL) engineering. The goal of this chapter is to propose an answer to the research question RQ4 introduced in Sect. 1.1: "How to repair feature models?". We present two different techniques to repair feature models. The first approach allows to repair feature models represented in the hierarchical tree-like structure, as well as *simple* constraints (i.e., only in the form of *requires* and *excludes* between couple of features) in propositional logic. The second approach, instead, applies to feature models represented as arbitrarily *complex* constraints, expressed in propositional logic. The two approaches can be combined, as to fully represent variability, unless complicating the model with a large number of abstract feature, it is normally more convenient to use *general* (i.e., *complex*) constraints, besides the hierarchical tree-like structure of feature dependencies. In the rest of the chapter, Sect. 5.1 covers the first approach reporting from paper [27], which includes and extends the previous paper [26], and Sect. 5.2 covers the second approach, i.e., the repair of relations among features in the form of constraints in propositional logic, reflecting the content of the paper [29].

Conclusions on these works are reported jointly in Sect. 5.3, together with lines for future research on this topic.

5.1 Achieving change requirements of feature models by an evolutionary approach

We developed an approach that *repairs* a feature model w.r.t. a given update request in the form of combinations representing a set of configurations to be accepted or rejected, that may be detected by *failing* test cases, or directly by engineer domain knowledge. The method is based on an evolutionary algorithm that iteratively mutates the original feature models and checks if the update request is semantically fulfilled. We employ mutations such as switching an optional feature to mandatory, or changing an *or* group to an *and* group, based on [26]. We generate faults between two real versions of feature models of the `MobileMedia`, `HelpSystem`, `SmartHome`, and `ERP_SPL` systems in the SPLOT repository[1], and we notice that although our approach does not guarantee to completely update all the possible feature models, on average, around 89% of requested changes are applied, with minimal edits.

[1] `http://52.32.1.180:8080/SPLOT/feature_model_repository.html`

5.1.1 Basic definitions

[27] In software product line engineering, feature models are a special type of information model representing all possible products of an SPL in terms of features and relations among them. Specifically, a feature model *fm* is a hierarchically arranged set of features F, where each parent-child relation between them is a constraint of one of the following types[2]:

- *Or*: at least one of the sub-features must be selected if the parent is selected.
- *Alternative* (xor): exactly one of the children must be selected whenever the parent feature is selected.
- *And*: if the relation between a feature and its children is neither an *Or* nor an *Alternative*, it is called *and*. Each child of an *and* must be either:
  - *Mandatory*: the child feature is selected whenever its respective parent feature is selected.
  - *Optional*: the child feature may or may not be selected if its parent feature is selected.

Only one feature in F has no parent and it is the *root* of *fm*.

In addition to the parental relations, it is possible to add *cross-tree constraints*, i.e., relations that cross-cut hierarchy dependencies. *Simple* cross-tree constraints are:

- A *requires* B: the selection of feature A in a product implies the selection of feature B. We also indicate it as $A \rightarrow B$.
- A *excludes* B: A and B cannot be part of the same product. We also indicate it as $A \rightarrow \neg B$.

Some frameworks for feature models also support *complex* cross-tree constraints [119] through *general* propositional formulas. In our approach, we allow feature models to contain complex cross-tree constraints.

Feature models can be visually represented by means of feature diagrams, and their semantics can be expressed by using propositional logic [38, 39]: features are represented by propositional variables, and relations among features by propositional formulae. We identify with BOF(*fm*) the BOolean Formula representing a feature model *fm*.

Definition 1 (Configuration). *A* configuration *c of a feature model fm is a subset of the features F of fm (i.e., $c \subseteq F$).*

[2] As done by FeatureIDE [149], we assume that each feature can be the father of only one group (either *Or*, *Alternative*, or *And*). This is not a limitation, as having different groups as children can be obtained by using abstract features [200].

If *fm* has n features, there are 2^n possible configurations.

Definition 2 (Validity). *Given a feature model fm, a configuration c is* valid *if it contains the* root *and respects the constraints of fm. A valid configuration is called* product.

For our purposes, we exploit the propositional representation of feature models for giving an alternative definition of configuration as a set of n literals $c = \{l_1, \ldots, l_n\}$ (with $n = |F|$), where a positive literal $l_i = f_i$ means that feature f_i belongs to the configuration, while a negative literal $l_i = \neg f_i$ means that f_i does not belong to the configuration. We will also represent a configuration as a BOolean Formula as follows: $\textsc{bof}(c) = \bigwedge_{i=1}^{n} l_i$.

Furthermore, since in the proposed approach we need to evaluate a feature model over a possibly wider set of features U, we introduce $\textsc{bof}(\mathit{fm}, U) = \textsc{bof}(\mathit{fm}) \wedge \bigwedge_{f \in U \setminus F} \neg f$, where *fm* explicitly refuses all the configurations containing a feature not belonging to its set of features F; such technique has been already employed by different approaches that need to compare feature models defined over different sets of features [199, 32].

Example 1. Let's consider the feature model shown in Fig. 16.

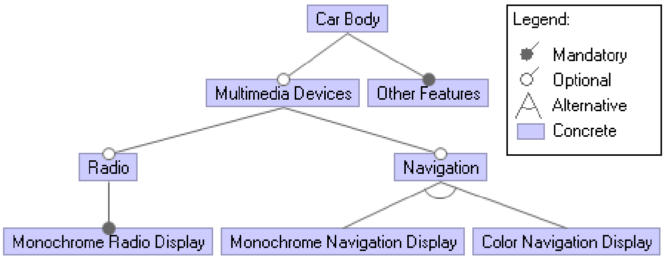


Figure 16: Example of feature model (taken from [170])

It is the third version of the CAR SPL described in [170]. It is composed of eight features F = {`CarBody`, `MultimediaDevices`, `OtherFeatures`, `Radio`, `Navigation`, `MonochromeRadioDisplay`, `MonochromeNavigationDisplay`, `ColorNavigationDisplay`}. `CarBody` is the root feature; its children `MultimediaDevices` and `OtherFeatures` are respectively optional and mandatory. `MultimediaDevices` has two optional children: `Radio` that is the father of the mandatory feature `MonochromeRadioDisplay`, and `Navigation` that is the father of the alternative group between `MonochromeNavigationDisplay` and `ColorNavigationDisplay`.

5.1.2 Specifying an update request

We suppose that the product line engineer wants to update an existing feature model fm_i (*initial feature model*); although (s)he knows which are the desired updates in terms of products and features to add or remove, (s)he does not know how to write a feature model fm' that satisfies all these updates.

In this section, we describe how the user can specify her/his *change requirements*. By analyzing existing evolutions of feature models described in literature [170, 54, 179, 206, 84], we identified the following five types of change requirements:

- a feature must be identified with a new name. Although this change requirement is straightforward to achieve, it is widely used (e.g., for the SmartHome SPL [179] considered in the Ample Project, we have observed 12 feature renames) and it is important to keep track of it, otherwise someone reading the updated feature model may have the impression that one feature has been added and one removed (in particular, if also other changes have been done on the feature model and, therefore, spotting the renamed feature is not easy).
- a feature must be added to the feature model. This change requirement occurs when a new feature must now be supported by the SPL. For example, in the CAR SPL [170], the support for *DVD entertainment* has been introduced in 2012 (documented in the fourth version of the SPL feature model).
- a feature must be removed from the feature model. This change requirement occurs when a feature is no more supported by the SPL.[3] For example, in the CAR SPL [170], feature *Color Radio Display* has been removed in 2011 (third version of the SPL feature model).
- some products must now be accepted by the SPL. This change requirement occurs when some constraints existing on the SPL (due to technical reasons or regulations) do not hold anymore and so some configurations that were forbidden in the past can now be accepted. For example, in the second version of the Pick-&-Place (PPU) Unit SPL [54], the system is able to process either *plastic* or *metal* workpieces; in the third version of the SPL, instead, a technical improvement has allowed to handle plastic and metal workpieces in combination.

[3] Note that this change requirement could be achieved by removing all the products containing the feature (see last change requirement), but not removing the feature from the feature model. However, in this way, the feature would become *dead* [39], making the feature model less readable.

- some products cannot be accepted anymore by the SPL. This change requirement happens when some constraints over the SPL emerge (e.g., due to new regulations). For example, in the HelpSystem SPL [206], the second version of the SPL does not allow anymore that the sensor only detects either *pressure* or *not pressure*, i.e., the products with only *pressure* or *not pressure* are not allowed and have been removed.

In the following, we provide the formal definition of update request. We assume that the initial feature model does not contain any dead feature or redundant constraint [39, 75]; in case it has any, we can remove them using standard techniques, as, for example, that provided by FeatureIDE [149].

Definition 3 (**Update request**). *Given an initial feature model fm_i defined over a set of features F, we call* update request *UR the modifications a user wants to apply to fm_i in order to obtain the desired updated model fm'. An update request is composed of five change requirements, three regarding the features of the feature model, and two the configurations/products. The first* feature-based *change requirement regards the features names:*

- **Rename features***: $F_{TBR} = \{f_1, \dots, f_m\}$ is a subset of features of F that must be renamed and ren a renaming function; F_R is the set obtained by replacing every feature $f_i \in F_{TBR}$ with $ren(f_i)$, i.e., $F_R = (F \setminus F_{TBR}) \cup \bigcup_{f_i \in F_{TBR}} \{ren(f_i)\}$. We identify with fm_{ren} the feature model obtained by renaming the features according to F_{TBR} and ren.*[4]

The other two feature-based *change requirements over the features of fm_{ren} are:*

- **Add features***: F_{add} is a set of features the user wants to add to fm_{ren}. For each e in F_{add}, the user has to define in $F_R \cup F_{add}$ the parent of e, denoted by $parent(e)$. By adding a feature e with parent p, we assume that the user wants to duplicate all the products that contain p by adding also e.*[5]
- **Remove features***: F_{rem} is a subset of the features of F_R to remove; by removing a feature f, we assume that the user wants to remove f from all the existing products and prohibit its selection in new products.*

[4] Note that feature renaming is already supported by feature model editors, such as FeatureIDE. We keep it among our change requirements for completeness with respect to the feature model evolutions observed in literature.

[5] Note that this corresponds to have e as optional feature of p. However, this could not be always achievable, as explained in Sect. 5.1.3 (unless abstract features [200] are used); therefore, we define the change requirement in this more general form, in order to keep the semantics of change requirement and the way to achieve it clearly distinguished.

We identify with F' the new set of features that must be used in the updated feature model fm′, namely $(F_R \cup F_{add}) \setminus F_{rem}$.

Two product-based *change requirements, instead, are related to the products/configurations of the feature model (over the new set of features F'):*

- **Add products***: C_{relax} is a set of predicates $\{\rho_1, \ldots, \rho_n\}$ over F' that denote conditions that must be allowed in fm′. The logical models satisfying ρ_i are products that must be added to the feature model, provided that they do not violate feature model constraints defined over features not included in ρ_i.*
- **Remove products***: C_{rem} is a set of predicates $\{\gamma_1 \ldots, \gamma_m\}$ over F' that denote a set of products to be removed. Each product (i.e., logical model) satisfying a γ_i must be removed from the valid product set.*

The meaning of C_{relax} and C_{rem} is to modify the set of valid products, as depicted in Fig. 17.

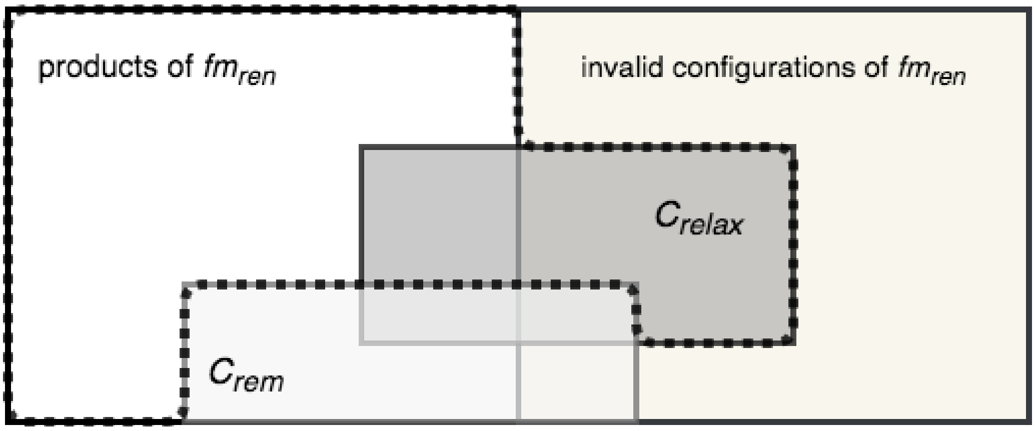


Figure 17: Added and removed products

The new product set (in dotted line) is enriched by configurations identified by C_{relax} but deprived of the products identified by C_{rem}; note that C_{rem} could also remove some products added by C_{relax}.

If the user wants to include/exclude a specific configuration c, then (s)he simply adds BOF(c) to C_{relax}/C_{rem}.[6]

[6] This is what we actually do in the experiments in Sect. 5.1.4 where update requests are computed as differences between different versions of feature models, and C_{relax} and C_{rem} can only be identified in terms of sets of configurations.

Well-formedness of an update request

We impose some constraints on the update request to be sure that the different change requirements are not contradictory and useful (i.e., they actually affect the product set):

1. Each added feature in F_{add} must have an ancestor in $F_R \setminus F_{rem}$. This implies that it is not possible to remove by F_{rem} the parents p of features added in F_{add}, i.e., $F_{rem} \cap (\bigcup_{f \in F_{add}} \{parent(f)\}) = \emptyset$.
2. It is not possible to remove features that have been renamed, i.e., $F_{rem} \cap (\bigcup_{f_i \in F_{TBR}} \{ren(f_i)\}) = \emptyset$.
3. Predicates in C_{relax} cannot predicate over features that have been removed in F_{rem}.
4. C_{relax} actually increases the set of products: for each ρ_i in C_{relax}, there exists at least one non-valid configuration that satisfies ρ_i.
5. C_{rem} actually deletes some products: for each γ_i in C_{rem}, there exists at least one product that satisfies γ_i.
6. C_{rem} does not remove all the configurations that must be added by C_{relax}, i.e., for each ρ_i and γ_j there exists a configuration that satisfies ρ_i (it must be added) and it does not satisfy γ_j.
7. The update request should be achievable by a consistent feature model (i.e., accepting at least one product) without dead features (i.e., each feature is selected in at least one model).

Constraints 1, 2, and 3 can be easily checked directly at the syntactical level on the update request. If one of these constraints is not satisfied, the update request cannot be applied and we ask the user to fix it.

Checking constraints 4, 5, and 6, instead, requires to reason over the propositional representation of fm_{ren} (i.e., $\textsc{bof}(fm_{ren})$) and the predicates in C_{relax} and C_{rem}; for example, checking constraint 4 consists in verifying that, for each $\rho_i \in C_{relax}$, $\neg\textsc{bof}(fm_{ren}) \wedge \rho_i$ is satisfiable. If one of these constraints is not satisfied, the update request is still consistent, although the corresponding change requirement is useless; in case of constraint violation, we warn the user about the useless change requirement and, if (s)he confirms that the change requirement is indeed not necessary, we continue the updating process without that requirement.

Checking constraint 7, instead, requires to reason about the interaction of the different change requirements and can only be done after we define the target, as explained in the next section.

Example 2. Given the CAR SPL model shown in Fig. 16 and described in Ex. 1, an update request could be the following[7]:

- **Rename features** F_{TBR} = {`MonochromeRadioDisplay`} and *ren* (`MonochromeRadioDisplay`) = `RadioDisplay`.
- **Add features** F_{add} = {`DVDEntertainment`} and *parent*(`DVDEntertainment`) = `MultimediaDevices`.
- **Remove features** F_{rem} = {`OtherFeatures`}.
- **Add products** C_{relax} = {`MultimediaDevices` ∧ `Navigation` ∧ (`MonochromeNavigationDisplay` ↔ `ColorNavigationDisplay`)}. We also want to allow products with support for navigation, and having both displays or having no display at all.
- **Remove products** C_{rem} = {`Navigation` ∧ ¬`Radio`}. We want to exclude that navigation is present when the radio is not.

Update request semantics

We here precisely define the semantics of an update request *UR*. In order to do this, we define a formula that accepts and rejects configurations as the updated feature model should do.

In order to define the semantics of F_{rem} and C_{relax}, we need to be able to represent the feature model without some features. We exploit the approach used in [200] to remove features from feature models. To eliminate a feature f from a propositional formula, we substitute f by its possible values (true or false).

Definition 4 (Features removal)**.** *Given a feature model fm and a set of features* $K = \{f_1, \ldots, f_k\}$ *to remove, we recursively define filter*(fm, K) *as follows:*

[7] Note that the change requirement F_{add} is the same reported in [170] for the evolution from the third version (shown in Fig. 16) to the fourth version of the CAR SPL; C_{rem}, instead, is taken from the evolution from the first to the second version of the same SPL. In order to have a complete example, the other three change requirements have been identified by us; F_{TBR} and F_{rem} resemble similar change requirements observed in literature, respectively in the SmartHome SPL [179] and in the CAR SPL [170].

$$filter(fm, K) = \begin{cases} \textsc{bof}(fm) & \text{if } K = \emptyset \\ filter(fm, K')[f \leftarrow \top] \vee filter(fm, K')[f \leftarrow \bot] & \text{if } K = K' \cup \{f\} \end{cases}$$

The formula *filter*(fm, K) has as logical models the same models as BOF(fm) except that all the features in K have been removed.

Exploiting Def. 4, the semantics of F_{rem} is captured by the formula

$$\phi_{rem} = filter(fm_{ren}, F_{rem})$$

whose logical models (i.e., products) are those of fm_{ren} without the removed features.

In order to capture the semantics of a $\rho_i \in C_{relax}$, we need to characterize the set of configurations to add. These are those that satisfy ρ_i but still respect the constraints of ϕ_{rem}, except for those involving the features of ρ_i. This is captured by

$$\phi^i_{relax} = filter(\phi_{rem}, features(\rho_i)) \wedge \rho_i$$

being *features* a function returning the features (i.e., propositional variables) contained in a formula. The *filter* on ϕ_{rem} has the effect of making the features in ρ_i unconstrained, i.e., it keeps only the constraints of ϕ_{rem} that do not *interfere* with ρ_i.

We can now build the formula that defines the semantics of the whole update request. We name the formula as *target* as it will be used as oracle to guide the proposed updating process (see Sect. 5.1.3).

Definition 5 (Target). *The target t is a propositional formula whose models exactly correspond to the products of the desired updated feature model. Assume an update request* $UR = \{ren, parent, F_{rem}, C_{relax}, C_{rem}\}$ *defined over a feature model fm, with the functions ren defined over* F_{TBR} *and parent defined over* F_{add}*. Let* fm_{ren} *be the feature model renamed according to ren; the target is defined as:*

$$t = (\overbrace{\phi_{rem}}^{\textit{remove } F_{rem} \textit{ from products}} \vee \overbrace{\bigvee_{\rho_i \in C_{relax}} \phi^i_{relax}}^{\textit{add products}}) \wedge \overbrace{\bigwedge_{f \in F_{add}} f \rightarrow parent(f)}^{\textit{add } F_{add} \textit{ features only when possible}} \wedge \overbrace{\bigwedge_{f \in F_{rem}} \neg f}^{\textit{disable } F_{rem} \textit{ features}} \wedge \overbrace{\bigwedge_{\gamma \in C_{rem}} \neg \gamma}^{\textit{remove products}}$$

The target correctly rejects all the configurations of C_{rem} and those containing a removed feature in F_{rem}; the accepted configurations are those in C_{relax}, plus those of ϕ_{rem} (i.e., the original feature model without the removed features) possibly extended with each added feature f of F_{add} only when $parent(f)$ is present.

Note that the target correctly predicates over all the features $F_U = F' \cup F_{rem}$: those of the original feature model (after renaming), those added, and those removed.

Checking the target

On the target, we can finally check constraint 7 described in Sect. 5.1.2, requiring that the update request does not produce anomalies.

First of all, we check that the target accepts at least one product (i.e., it is satisfiable); if not, we warn the user that the update request cannot be applied.

Moreover, we also check that each feature $f \in F'$ can be actually selected in at least one product; it this is not possible, it means that f is required to be dead in the final model. If this is the case, we warn the user about this and, if this is the intended behavior, we move f in F_{rem}, so that we can directly try to remove it during the repairing process.

5.1.3 Evolutionary updating process

Modifying the initial feature model fm_i such that it satisfies the update request as specified by the target is a challenging task. Note that, in general, there could be no fm' that exactly adds and removes the specified configurations and features, unless *complex* cross-tree constraints are used. However, we claim that the usage of these constraints should be discouraged. Urli et al. [202] observe that “they make FMs complex to understand and maintain”, Reinhartz-Berger et al. [175] experimentally assessed that they are more difficult to understand than parent-child relationships (at least, by modelers who are unfamiliar with feature modeling), and Berger et al. [41] report that “they are known to critically influence reasoners”. Also the authors of FeatureIDE noted that cross-tree constraints “are harder to comprehend than simple tree constraints" and that “relations among features should be rather expressed using the tree structure if possible” [149].

In this work, we therefore avoid the addition of complex cross-tree constraints, as we not only aim at correctness (i.e., full achievement of the update request), but also at readability of the final model.

Some approaches that aim at simplifying complex constraints exist [119, 205], but they may diminish readability and other qualities, such as compactness, traceability, and maintainability.[8]

In this section, we propose a heuristic approach that tries to achieve the update request *as much as possible*, by using the target as *oracle* to compute the *fault ratio*. The *fault ratio* tells how close we are to the correct solution. In Sect. 5.1.3, we give the definition of fault ratio and then, in Sect. 5.1.3, we describe the updating process we propose.

Correctness

We can use the target to evaluate whether a feature model fm' captures the desired change requirements, i.e., fm' is equivalent to the target ($\models t \leftrightarrow \text{BOF}(fm', F_U)$). In the following, we will only compare feature models fm' whose features $F_{fm'}$ are, at most, those in F_U, i.e., $F_{fm'} \subseteq F_U$.

Although a feature model could not fulfill all the change requirements, it could satisfy them partially. We give a measure of the *difference* between a feature model fm (either the initial one fm_i or a modified one fm') and the target as follows.

Definition 6 (**Fault ratio**). *Given a feature model fm and a target t, the* fault ratio *of fm w.r.t. t is defined as follows:*

$$FR(fm, t) = \frac{\texttt{nsat}(\textit{BOF}(fm, F_U) \neq t, F_U)}{2^{|F_U|}}$$

where $\texttt{nsat}(\varphi, V)$ *returns the number of logical models of formula* φ*, i.e., all the truth assignments* m *to propositional variables* V*, such that* $m \models \varphi$*.*[9]

If the fault ratio is equal to 0, it means that fm accepts as products the same configurations that are logical models of t; otherwise, there are some configurations that are wrongly evaluated by fm, as shown in Fig. 18: fm could wrongly accept some configurations and/or wrongly refuse some others.

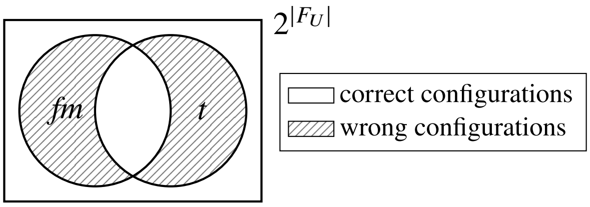


Figure 18: Faults

[8] In Sect. 5.3, we discuss how complex cross-tree constraints could be used to achieve all the change requirements and the issues that could be related to that approach.

[9] In our approach, we represent formulas as Binary Decision Diagrams (BDDs) in JavaBDD that implements `nsat` by means of the JavaBDD method `satCount` that directly computes the cardinality of the set without enumerating all the models.

Example 3. Fig. 19 shows a possible updated model that exactly satisfies all the change requirements reported in Ex. 2 for the model in Fig. 16.

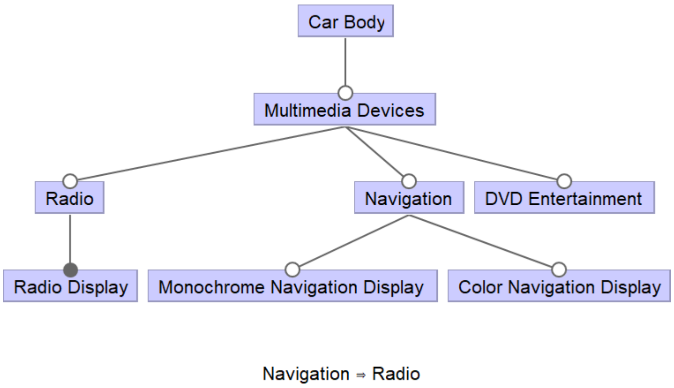


Figure 19: Updated feature model

Note that the fault ratio of the initial renamed feature model fm_i w.r.t. to the target t (that corresponds to the final feature model) is $\frac{20}{2^9}$, as fm_i wrongly accepts all its 7 products and wrongly rejects all the 13 products of the target.

Updating process

The process we propose to update an initial feature model fm_i, given an update request UR, is depicted in Fig. 20.

As initial step, we generate the target t as a Binary Decision Diagram (BDD), as described in Def. 5. Then, we start the updating process, that is composed of two consecutive macro-phases. We first try to deal with the feature-based change requirements (see Sect. 5.1.3) and then with the product-based ones (see Sect. 5.1.3).

Dealing with feature-based change requirements

First of all, we apply F_{TBR} to rename features, obtaining the feature model fm_{ren}.

Then, we modify fm_{ren} in order to try to achieve the change requirements of F_{add} and F_{rem}. For each $f \in F_{add}$, we add f as child of $parent(f)$ as follows: if $parent(f)$ is the father of an *Or* or an *Alternative* group, f is added to the group; in all the other cases, it is added as *Optional* child of $parent(f)$. We name as fm_A the feature model obtained after this step. Then, for each feature $f \in F_{rem}$, f is removed from fm_A and replaced by its children Ch_f (if any). The relation of the moved children Ch_f of f with their new parent p is set according to the way FeatureIDE removes features [149]:

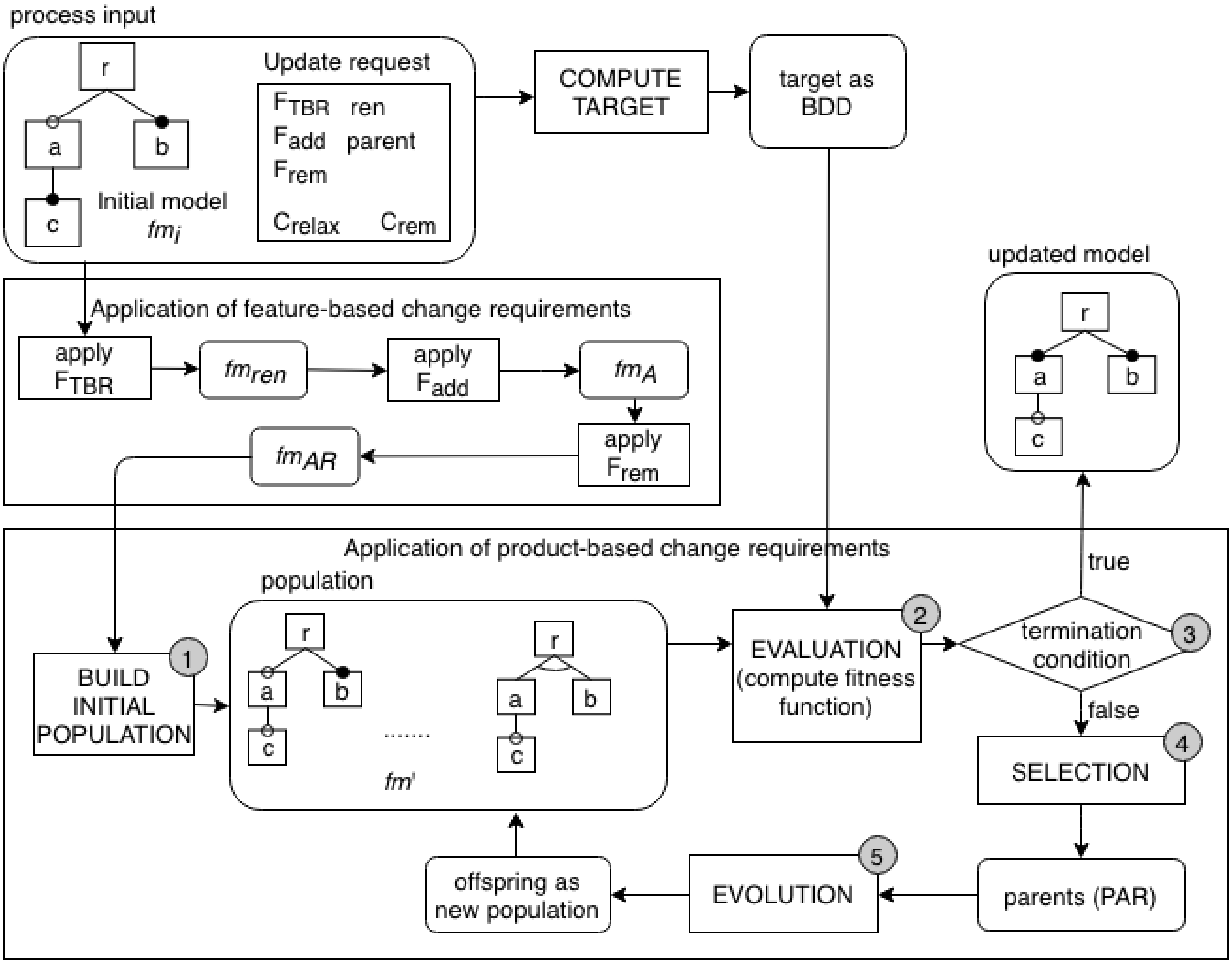


Figure 20: Proposed evolutionary approach

1. If f was the only child of p, p takes the group type of f.
2. If p has group type *And*, a) if the children Ch_f are in *And* relationship, they keep their type (either *Mandatory* or *Optional*) b) otherwise (they are in *Alternative* or *Or*), they are set to *Optional*.
3. Otherwise, if p has group type *Alternative* or *Or*, features in Ch_f are simply added to the group (regardless of their type).

We name as fm_{AR} the feature model obtained after this step.

Note that the model fm_{AR} could still be not equivalent to the target, i.e., $\not\models t = \textsc{bof}(fm_{AR}, F_U)$. This could be due to two reasons. First of all, the update request could also require to add as products configurations described by constraints in C_{relax} and/or remove configurations described by constraints in C_{rem}. Moreover, the two previous transformations do not guarantee to precisely implement the required change requirements F_{add} and F_{rem}, and they could introduce some wrong configurations (either wrongly accepted or rejected). For example, in order to implement the addition of a feature f with $parent(f) = p$ and p father of an alternative group, we add f to the group; however, this is not the exact semantics of F_{add} that requires to duplicate the products containing p and adding f to them.

Dealing with product-based change requirements

Starting from fm_{AR}, we apply an evolutionary updating approach to try to obtain an updated feature model equivalent to the target. The process is an instance of classical evolutionary algorithms [81]; an evolutionary algorithm can be understood (in a metaphor-free language [189]) as an optimization problem in which different solutions are modified by random changes and their quality is checked by an objective function. Precisely, the steps are as follows:

1. **Initial population**: at the beginning, a population P is created. P is a set of candidate solutions.
2. **Iteration**: the following steps are repeatedly executed:
    a) **Evaluation**: each member of the population P is evaluated using a given *fitness function*, representing the objective function.
    b) **Termination**: a termination condition is checked in order to decide whether to start the next iteration. If the termination condition holds, the candidate with the best *fitness value* is returned as final model.
    c) **Selection** (*Survival of the Fittest*): some members of P having the best values of the fitness function are selected as parents of the next generation and collected in the set PAR.
    d) **Evolution**: parents PAR are mutated to obtain the *offspring* to be used as population in the next iteration. The mutation performs random changes suitable to improve the existing solutions.

In our approach, we assume that the population P is a multiset (i.e., possibly containing duplicated elements) with fixed size M equal to $H \cdot |F'|$, where H is a parameter of the process. In the following, we describe each step in details.

Initial population

As initial population, we generate the set P by cloning fm_{AR} M times (step 1 in Fig. 20). In this way, if fm_{AR} is already correct, it will be returned as final model in the *termination condition* phase.

Evaluation

As first step of each iteration (step 2 in Fig. 20), each candidate member fm' of the population P is evaluated using a *fitness function* that tells how *good* the member is in achieving the overall goal. We define the fitness function both in terms of fault ratio (see Def. 6) and of *complexity* of the model structure. Indeed, we would like to avoid that, during the updating process, the feature model becomes unreadable, unnecessarily complex, and difficult to maintain [41, 202, 175, 149]. We have decided to consider, at least initially, the number of cross-tree constraints as indicator of complexity, since the constraints among features should be expressed by structural relationships and cross-tree constraints should be used only when really necessary. We introduce the following fitness function:

$$fitness_t(fm') = 1 - FR(fm', t) - k \times ctc(fm') \tag{5.1}$$

where ctc is a function returning the number of cross-tree constraints of a feature model and k a constant. In our approach, the quality of a candidate must be mainly given by the percentage of configurations that it evaluates correctly, i.e., $1 - FR(fm', t)$; if a candidate c_1 evaluates correctly more configurations than a candidate c_2, its fitness should be guaranteed to be greater than that of c_2. However, for models having the same fault ratio, the fitness should penalize those that are structurally more complex. In order to obtain this effect, we use as k:

$$k = \frac{1}{2^{|F_U|} \times 2|F_U|^2} \tag{5.2}$$

Note that $2|F_U|^2$ is a safe strict upper bound on the number of cross-tree constraints among the features of the feature model. Indeed, among the possible $2|F_U|^2$ cross-tree constraints, some of them are not introduced because redundant (e.g., two *excludes* constraints between (a, b) and (b, a) are redundant and only one is necessary). Therefore, the term $\frac{ctc(fm')}{2^{|F_U|} \times 2|F_U|^2}$ is guaranteed to be less than $\frac{1}{2^{|F_U|}}$ that is the minimal possible variation of the fault ratio due to the change of evaluation of a single configuration. This means that the term can only affect the ranking of feature models having the same fault ratio.

Termination condition

In this step (step 3 in Fig. 20), the process checks whether at least one of the following conditions is met:

- a defined level of fitness Th_f is reached, i.e., there exists an fm' in P with $fitness_t(fm') \geq Th_f$. For example, $Th_f = 1$ means that we want to obtain a completely correct model without any cross-tree constraint; with $Th_f = 1 - \frac{1}{2^{|F_U|+1}}$, instead, we still want to have a correct model, but we allow to have any number of cross-tree constraints;
- in the previous Th_{NI} iterations there has been no improvement of the fitness value of the best candidate;
- a maximum number Th_i of iterations have been executed;
- a total time threshold Th_t has been reached.

If at least one of the previous conditions holds, the fm' in P with the highest fitness value is returned as final model.[10]

Selection

In the *selection* step (step 4 in Fig. 20), starting from population P, a multiset of *parents* *PAR* of size p is built, being p a parameter of the evolutionary process. Different selection strategies have been proposed in literature:

- *Truncation*: it selects the first $n = \lceil K \cdot |P| \rceil$ members of the population with the highest fitness value, where K is a parameter specifying a percentage of the population ($0 < K \leq 1$). Then, the first n elements are added to *PAR* as many times as necessary to reach the size p. Such strategy could result in premature convergence, as candidates with lower fitness values are not given the opportunity to improve their fitness.
- *Roulette wheel*: p members of the population are selected randomly; each member can be selected with a probability proportional to its fitness value. Note that one or more individuals could be selected multiple times.
- *Rank*: it is similar to roulette wheel, except that the selection probability is proportional to the relative fitness rather than the absolute fitness, i.e., the probability of selecting a member is inversely proportional to its ranking number (where the

[10] If there is more than one model with the highest fitness value, we randomly select one of these models.

member with highest fitness has ranking number 1). This strategy tends to avoid premature convergence by mitigating the selection pressure that comes from large differences in fitness values (as it happens in truncation selection).

Evolution

In the *evolution* step (step 5 in Fig. 20), the parents *PAR* are used to generate the *offspring* that constitutes the population of the next generation.

The idea we assume here is that the feature model should be updated applying a limited number of mutations. Making updates through the use of mutation operators has the benefit of reducing the risk of loss of domain knowledge, by changing the feature model as less as possible. Note that this assumption is similar to the competent programmer hypothesis [116] that assumes that the user has defined the artifact close to the correct one. If our approach is used for removing faults, we can directly rely on the competent programmer hypothesis. On the other hand, if the approach is used to evolve the feature model to align it with the SPL, we can still assume that the mutation operators are sufficient to obtain the updated model; indeed, it is unlikely that the updated version of the feature model should be too different from the initial one.

In order to build the next population *P*, we mutate all the feature models in *PAR* using the operators presented in Table 11.

We set an upper bound *M* to the size of the new population. If the mutation operators generate a maximum of *M* mutants, we take all of them as the new population, otherwise we randomly select *M* of them. In our approach, the offspring replaces the entire population.

Description of mutation operators In [32], we have proposed some mutation operators for feature models, divided in *feature-based* and *constraint-based* operators that are a subset of the edit operations identified in [54]. We use eight of the feature-based mutation operators proposed in [32], and introduce two new ones (`OrToAndOpt` and `AlToAndOpt`) that provide slightly different versions of `OrToAnd` and `AlToAnd`. Moreover, we also introduce two operators that permit to move a feature as sibling of the parent (`PullUp`) or as child of one its siblings (`PushDown`); we also introduce two versions of these operators that move all the children of a feature (`PullUpCh`) and all the siblings of a feature (`PushDownSibl`). Note that we do not allow the movement of a feature in any

Name	Description
`OptToMan`	an *optional* feature is changed to *mandatory*
`ManToOpt`	a *mandatory* feature is changed to *optional*
`OrToAl`	an *or* group is changed to *alternative*
`OrToAnd`	an *or* group is changed to *and* with all children *mandatory*
`OrToAndOpt`	an *or* group is changed to *and* with all children *optional*
`AlToOr`	an *alternative* group is changed to *or*
`AlToAnd`	an *alternative* group is changed to *and* with all children *mandatory*
`AlToAndOpt`	an *alternative* group is changed to *and* with all children *optional*
`AndToAl`	an *and* group is changed to *alternative*
`AndToOr`	an *and* group is changed to *or*
`PullUp`	a feature is moved as sibling of its parent
`PushDown`	a feature is moved as child of one of its siblings
`PullUpCh`	all children of a feature are moved as siblings of their parent
`PushDownSibl`	all siblings of a feature are moved as children of that feature
`DelConstr`	a cross-tree constraint (*requires* or *excludes*) is deleted
`ReqToExcl`	a *requires* constraint is changed to an *excludes* constraint
`ExclToReq`	an *excludes* constraint is changed to a *requires* constraint
`AddReq`	a *requires* constraint is created
`AddExc`	an *excludes* constraint is created

Table 11: Mutation operators

part of the feature model as this would produce too many mutants and could change too much the structure of the feature model. However, if in order to obtain the correct feature model a feature should be moved *far* from its current position, this is still obtainable by a suitable sequence of `PullUp` and `PushDown` mutations.

Finally, we use all the *constraint-based* operators[11] proposed in [32], and introduce two new ones (`AddReq`, `AddExc`) to create *requires* and *excludes* constraints; in order to limit the number of generated mutants, we only create constraints among features that belong to different sub-trees of the feature model, i.e., neither feature of the constraint is ancestor of the other. Moreover, we avoid to create constraints that would be redundant or that would introduce dead features: for instance, $A \rightarrow \neg B$ is not added if $A \rightarrow B$ is already present in the model, because A would become a dead feature.

[11] Note that the mutation operator `DelConstr` corresponds to operator `MC` described in [32].

Although we allow feature models with constraints also in general form, we decided to not modify them, nor introduce new ones, in order to avoid the introduction of too many mutants and to achieve better readability of the final model.

In general, we cannot evaluate a priori if a mutant introduces an anomaly; therefore, for each mutant, we check if it is infeasible, it contains redundant constraints, or it has dead features. If it has any of these anomalies, we do not select it.

5.1.4 Experiments

The process[12] has been implemented in Java by using Watchmaker[13] as evolutionary framework, FeatureIDE [149] to represent and mutate feature models, and JavaBDD for BDDs manipulation.

We conducted a set of experiments to evaluate the proposed evolutionary approach; they have been executed on a Windows 10 system with an Intel i7-3770 3.40GHz processor, and 16 GB RAM.

Benchmarks

For the experiments, we used two sets of benchmarks, both shown in Table 12.

		model size		*UR* size				
	SPL	input	target	$\lvert F_{TBR}\rvert$	$\lvert F_{add}\rvert$	$\lvert F_{rem}\rvert$	$\lvert C_{relax}\rvert$	$\lvert C_{rem}\rvert$
BENCH$_{\text{REAL}}$	`MobileMedia d1` (V5..8)	18.7 (15-23)	22.3 (18-26)	11.3 (1-17)	4 (3-5)	0.33 (0-1)	26.7 (0-48)	258 (24-560)
	`MobileMedia d2` (V5..8)	16.5 (15-18)	24.5 (23-26)	12 (11-13)	8 (8-8)	0 (0-0)	64 (48-80)	536 (528-544)
	`MobileMedia d3` (V5,V8)	15	26	9	11	0	80	1648
	`HelpSystem` (V1, V2)	25	26	0	1	0	672	2016
	`SmartHome` (V2.0, V2.2)	39	60	12	23	2	1.92×10^{9}	2.31×10^{10}
	`ERP_SPL` (V1, V2)	43	58	0	15	0	0	1.51×10^{7}
	`PPU d1` (V1..9)	13.9 (9-17)	14.9 (11-17)	0 (0-0)	1.13 (0-4)	0.125 (0-1)	3.75 (0-27)	27.8 (0-183)
	`PPU d2` (V1..9)	13.4 (9-17)	15.4 (11-17)	0 (0-0)	2.14 (0-6)	0.142 (0-1)	9 (0-27)	54.4 (0-243)
	`PPU d3` (V1..9)	12.8 (9-17)	16.2 (13-17)	0 (0-0)	3.5 (1-6)	0.167 (0-1)	16 (0-27)	77.5 (9-156)
	`CAR d1` (V2009..2012)	7 (6-8)	9.33 (7-13)	0 (0-0)	2.67 (1-5)	0.333 (0-1)	5 (0-13)	16 (1-40)
	`CAR d2` (V2009..2012)	6.5 (6-7)	10.5 (8-13)	0 (0-0)	4.5 (3-6)	0.5 (0-1)	6 (0-12)	34 (9-59)
	`CAR d3` (V2009,V2012)	6	13	0	7	0	0	87
BENCH$_{\text{MUT}}$	`Register`	11	11	0	0	0	11.85 (0-40)	62.28 (0-210)
	`Graph`	6	6	0	0	0	12.71 (0-28)	0 (0-0)
	`Aircraft`	13	13	0	0	0	196.86 (0-315)	53.53 (0-365)
	`Connector`	20	20	0	0	0	8.23 (0-18)	26.02 (0-336)

Table 12: Benchmark properties

[12] The code is available at `https://github.com/fmselab/eafmupdate`

[13] `https://watchmaker.uncommons.org/`

Real models

The first benchmark set $\texttt{BENCH}_{\texttt{REAL}}$ is constituted by SPLs described in literature for which different versions of their feature model have been developed. First, we have identified in the SPLOT repository[14] four SPLs that evolved over time:

- MobileMedia: a program to manipulate multimedia on mobile devices (four versions) [84];
- HelpSystem: a cyber-physical system with multiple sensors (two versions) [206];
- SmartHome: a set of smart house components (two versions) [179];
- ERP_SPL: an Enterprise Resource Planner (two versions).

Then, we have also considered the industrial case of a Pick-and-Place Unit (PPU) [54]; for this system, the feature model has been changed eight times to adapt to new requirements (therefore, there are nine feature models available [54]). Finally, we have also considered the product line model of a CAR [170], for which four different feature models have been produced.

For each SPL, we identified couples (fm_i, fm_t) of their feature models: the latest version was considered as target model[15] fm_t, and the oldest one as the initial model fm_i we want to update. For SPLs with more than two feature models, in addition to couples of feature models of consecutive versions, we also considered models with version distance 2 and 3 (in the table, d1, d2, and d3 indicate the distance). In this way, we attempt to reproduce update requests of different complexity. In total, $\texttt{BENCH}_{\texttt{REAL}}$ contains 36 couples of models.

Generated models

The second benchmark set $\texttt{BENCH}_{\texttt{MUT}}$ has been built with the aim of evaluating our approach under the assumption we did in Sect. 5.1.3 that mutation operators are sufficient to update the feature model. We selected four feature models developed for four SPLs (for which only one model is available):

- Register: a register of supermarkets, adapted from [185];
- Graph: a graph library;

[14] http://52.32.1.180:8080/SPLOT/feature_model_repository.html

[15] Note that, in the real usage of our approach, we do not have a target feature model, but an update request *UR* from which we generate the target as propositional formula.

- `Aircraft`: the configurations of the wing, the engine, and the materials of airplane models;
- `Connector`: IP connection configurations.

From these models (used as target models fm_t), we automatically generated other versions to be used as input models fm_i; we randomly mutated the target models (using 1 to 10 mutations), applying the operators described in Table 11. For each target model, we generated 100 input models. Therefore, $\texttt{BENCH}_{\texttt{MUT}}$ contains 400 couples.

Deriving the update request

In the devised usage of the approach, the user should specify the update request that must be provided as input to the evolutionary process; however, for our experiments, we do not have any update request available. Therefore, we automatically generated an update request UR from the initial feature model fm_i and the target feature model fm_t of each couple (fm_i, fm_t) of the benchmarks.

In order to detect renamed features in F_{TBR}, we manually inspected the two feature models and produced the renamed model fm_{ren}. Then, from fm_{ren} and fm_t, we automatically identified the differences of their features for building F_{add} and F_{rem}; moreover, using their BDD representation, we identified the configurations that are differently evaluated in order to build C_{relax} and C_{rem} (we built a predicate for each wrongly evaluated configuration[16]). Note that configurations that are added and removed by F_{add} and F_{rem} are not also specified in C_{relax} and C_{rem}.

Table 12 reports, for all the benchmarks, the size of the input and target models in terms of number of features, and the number of requirements of the update request. For the SPLs in $\texttt{BENCH}_{\texttt{REAL}}$ having more than one couple of feature models, the reported values are aggregated by distance; for each SPL in $\texttt{BENCH}_{\texttt{MUT}}$, the values are aggregated among its 100 input models. For these aggregated models, we report the average, minimum, and maximum number of elements in the update request. In $\texttt{BENCH}_{\texttt{MUT}}$, since we did not add or remove features for producing the input models, their size is the same of that of the target model and so F_{add} and F_{rem} are empty; moreover, we do not even rename features and so also F_{TBR} is empty.

[16] Note that, in this way, the sets C_{relax} and C_{rem} are very large because they contain a predicate for each added and removed configuration. However, in a real setting, the user should specify predicates capturing sets of configurations and so the sets should be much smaller.

Analysis

We now evaluate the proposed approach by a series of research questions. In these experiments, we set the parameters of the termination conditions as follows: Th_f to $1 - \frac{ctc(fm_i)}{2^{|F_U|} \times 2|F_U|^2}$, Th_{NI} to 15, Th_i to 25, and Th_t to 30 minutes. Note that the value chosen for Th_f requires to have a totally correct model, with at most the same number of cross-tree constraints of the starting feature model. The parameter H used to determine the maximum population size M (as defined in Sect. 5.1.3) has been set to 5, and the parameter p of the selection phase has been set to $M/2$. All the reported data are the averages of 30 runs.

In [26], we experimented the effect of the different selection policies of the evolutionary approach and we found that truncation with K=5% is the best policy in terms of fault ratio reduction, and the second best as execution time. Therefore, we here select it as selection policy and evaluate the approach using other research questions.

RQ1: *Is the proposed approach able to achieve the change requirements specified by the update request?*

For each benchmark SPL, Table 13 reports (among other things) the initial and final values of *FR*, and the *FR* reduction.

	SPL	time (s)	initial FR (%)	final FR (%)	FR reduction (%)	# iterations	sem. eq. (%)	synt. eq. (%)	ctc	ED
$\text{BENCH}_{\text{REAL}}$	MobileMedia d1	71.11	4.17e-03	1.10e-04	96.03	8.42	64.44	17.78	0.23	14.86
	MobileMedia d2	157.98	3.90e-03	4.41e-04	86.54	16.00	0.00	0.00	0.63	34.32
	MobileMedia d3	178.46	2.57e-03	3.57e-04	86.13	16.00	0.00	0.00	0.80	36.97
	HelpSystem	112.41	4.01e-03	1.24e-04	96.92	8.80	86.67	0.00	5.57	25.53
	SmartHome	1990.40	7.24e-07	8.39e-08	88.42	6.10	0.00	0.00	0.43	57.60
	ERP_SPL	2014.30	5.24e-09	8.86e-10	83.08	6.00	0.00	0.00	0.43	18.80
	PPU d1	5.98	0.09	2.27e-03	97.44	3.88	86.67	48.33	1.23	7.40
	PPU d2	10.23	0.10	0.54e-03	96.48	6.84	68.57	36.67	1.39	11.63
	PPU d3	15.90	0.09	0.01	87.43	9.69	48.89	28.89	1.94	17.04
	CAR d1	3.85	1.13	0.25	69.81	11.18	57.78	6.67	0.77	8.02
	CAR d2	4.21	1.31	0.17	80.31	10.52	50.00	0.00	1.83	10.15
	CAR d3	6.59	1.06	0.37	65.52	16.00	0.00	0.00	2.37	20.13
$\text{BENCH}_{\text{MUT}}$	Register	4.35	3.62	0.17	95.84	7.15	80.27	65.20	0.22	2.32
	Graph	0.12	19.86	0.00	100.00	1.76	100.00	99.03	2.67e-03	0.02
	Aircraft	8.09	3.06	0.07	97.49	6.58	84.53	68.63	0.29	2.89
	Connector	28.68	3.27e-03	6.87e-05	98.11	5.80	94.00	68.47	0.20	2.27

Table 13: Performance of the updating process

For the models in $\texttt{BENCH}_{\texttt{REAL}}$, the table reports the averages among couples at the same distance; for the models in $\texttt{BENCH}_{\texttt{MUT}}$, instead, it reports the averages among the 100 input models of each SPL.[17] We observe that for all models we can reduce the fault ratio of at least 65%, with an average of 89.1%. Comparing the two benchmark sets, we notice that the reduction is higher in $\texttt{BENCH}_{\texttt{MUT}}$ (on average, 97.86%) than in $\texttt{BENCH}_{\texttt{REAL}}$ (on average, 86.15%); this means that, as expected, if the assumption that the models can be updated using the proposed mutation operators holds, the approach behaves very well. However, also for general models as those in $\texttt{BENCH}_{\texttt{REAL}}$, the performance of the approach is quite good.

RQ2: *Which is the computational effort of the proposed approach?*

We are here interested in the effort required by the proposed approach in terms of computation time and iterations of the evolutionary process. Table 13 also reports the total execution time of our process and the number of iterations of the evolutionary approach. For all but two models, the process takes at most 179 seconds; as expected, smaller models (as `CAR`, `Graph`, and `Register`) are updated faster than larger models (as `MobileMedia`, `HelpSystem`, `SmartHome`, and `ERP_SPL`). We observe that the process terminates when one of these three terminating conditions occurs: (i) Th_f, i.e., the model is completely updated (as in `HelpSystem` and `Graph`), (ii) Th_t, i.e., the timeout has occurred (as in `SmartHome` and `ERP_SPL`[18]), or (iii) Th_{NI}, i.e., no fitness improvement has been observed in the previous 15 iterations (as, for example, all the models of `MobileMedia d2` and `d3`, and those of `CAR d3`). Note that the process never terminates because of the maximum number of iterations Th_i.

RQ3: *Is there a relation between the initial fault ratio and its* updatability*?*

We are here interested in investigating whether there is a relation between the initial fault ratio and its reduction. Fig. 21 shows, for each benchmark model (a point in the plot), its initial fault ratio and the fault ratio reduction.

It seems that there is no proper correlation: we reduce (or do not reduce) the fault ratio in the same proportion among models having different initial fault ratios. We checked

[17] Non-aggregated results are reported online at `http://foselab.unibg.it/eafmupdate/`

[18] Note that the time for these two models is around 3.5 minutes above the threshold Th_t of 30 minutes; indeed, the terminating condition is checked at the end of an evolution step, but the threshold could be overcame during the step.

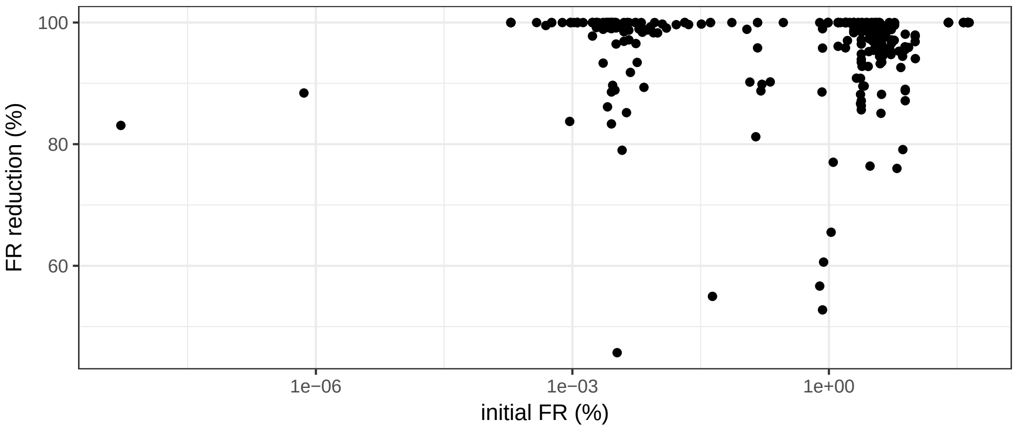


Figure 21: Relation between the initial fault ratio and the fault ratio reduction

the correlation with the Spearman rank-order correlation coefficient [210], and indeed we found a value of 0.23 indicating almost no correlation [109].

RQ4: *Are the final models similar to those produced by SPL designers?*

The main aim of the proposed approach is to obtain a feature model that satisfies all the change requirements; to this purpose, we use the fault ratio as measure of model correctness. However, the final model we obtain, although correct, may be not readable by and not useful for an SPL designer. Although we could not ask real SPL designers to validate our models in terms of usefulness and readability, we have access to models developed by SPL designers to achieve the same update requests we tackled in the experiments (models fm_t used as target in the experiments). We can assume that SPL designers are likely to find our models useful if models produced by our process are similar to their models. We therefore measure the *readability* of the final model of our process in terms of *distance* from the model fm_t developed by a designer for the same update request. We compute the *edit distance* [165, 166] of the final model fm_f from fm_t, defined as the number of *edits* (insertion, deletion, and rename of tree nodes) that we have to apply to fm_f in order to obtain fm_t.[19] Table 13 reports also the edit distance (ED) to the target model (average among the models), and the percentage of models that are syntactically equal and semantically equivalent to the target model. Of course, models not completely updated are not syntactically equal to the target and have edit distance greater than 0. Completely correct models (semantically equivalent) are often also syntactically equal (for example, 86.67% of the `PPU d1` models are semantically equivalent and 48.33% are

[19] Note that these edit operations are more fine-grained than our mutation operators and we are always able to compute the distance between two feature models.

also syntactically equal); however, there are some correct final models that are different from the target model fm_t (for example, HelpSystem is completely updated 86.67% of the times, but always in a different way than fm_t).

RQ5: *Does considering the number of cross-tree constraints in the fitness impact the final results?*

As explained in Sect. 5.1.3, our fitness function (see Eq. 5.1) can also take into consideration the number of cross-tree constraints (*ctc*); the value we selected for k (see Eq. 5.2) has the aim of penalizing models with higher *ctc* at the same fault ratio (in order to limit the insertion of such constraints and instead give precedence to changes of the parental relations). In order to assess the impact of this choice, we have executed the same experiments presented before with $k = 0$ in the fitness function (that becomes $fitness_t(fm') = 1 - FR(fm', t))$ and $Th_t = 1$. Table 14 reports the data in terms of average fault ratio reduction, percentage of semantically equivalent, percentage of syntactically equal models, and the average number of cross-tree constraints.

	SPL	Fitness without constr. ($k = 0$)				Fitness with constr. (k as in Eq. 5.2)			
		FR reduction (%)	sem. eq. (%)	synt. eq. (%)	ctc	FR reduction (%)	sem. eq. (%)	synt. eq. (%)	ctc
$\text{BENCH}_{\text{REAL}}$	MobileMedia d1	94.10	61.11	17.78	0.69	96.03	64.44	17.78	0.23
	MobileMedia d2	84.83	0.00	0.00	1.85	86.54	0.00	0.00	0.63
	MobileMedia d3	85.16	0.00	0.00	2.43	86.13	0.00	0.00	0.80
	HelpSystem	97.22	86.67	0.00	5.57	96.92	86.67	0.00	5.57
	SmartHome	88.42	0.00	0.00	0.53	88.42	0.00	0.00	0.43
	ERP_SPL	82.96	0.00	0.00	0.40	83.08	0.00	0.00	0.43
	PPU d1	98.10	87.50	45.83	1.62	97.44	86.67	48.33	1.23
	PPU d2	95.89	68.57	37.14	2.11	96.48	68.57	36.67	1.39
	PPU d3	87.25	50.00	27.22	2.95	87.43	48.89	28.89	1.94
	CAR d1	73.61	55.56	3.33	1.89	69.81	57.78	6.67	0.77
	CAR d2	78.06	45.00	0.00	3.22	80.31	50.00	0.00	1.83
	CAR d3	62.80	0.00	0.00	4.63	65.52	0.00	0.00	2.37
$\text{BENCH}_{\text{MUT}}$	Register	96.11	82.70	64.03	0.60	95.84	80.27	65.20	0.22
	Graph	100.00	100.00	99.37	0.00	100.00	100.00	99.03	0.00
	Aircraft	97.43	85.97	68.73	0.42	97.49	84.53	68.63	0.29
	Connector	98.05	93.70	63.00	0.48	98.11	94.00	68.47	0.20

Table 14: Performance of the updating process with the two versions of the fitness

For easing the comparison, we also report the same data of the results obtained with the previous experiment (already reported in Table 13). In order to check whether the consideration of cross-tree constraints has some effect on the final results, we have applied to the data the classical hypothesis testing by performing the Wilcoxon signed-rank test[20] [210] between the results with the two versions of fitness. The null hypothesis that considering *ctc* has no impact of the final model cannot be rejected for the percentage of totally updated models (i.e., semantically equivalent), but it is rejected for the syntactical equivalence with p-value equal to 0.0038. This confirms that penalizing the usage of cross-tree constraints in the fitness improves the quality (readability) of the final model without compromising the ability of the approach in achieving the update request.

5.1.5 Threats to validity

We discuss the threats to the validity of our results along two dimensions, *external* and *internal* validity [210].

External validity

Regarding external validity, a threat is that the obtained results could be not generalizable to real-world (industrial) feature models having specific update requests. However, as first benchmark, we have selected 9 couples of models showing the evolution of real SPLs [84, 206, 179] taken from the SPLOT repository, and other 27 couples from the evolution of other two real SPLs described in literature [54, 170] (see Sect. 5.1.4); moreover, in order to enlarge the set of evaluated models, we generated 400 input models by randomly mutating other 4 feature models (acting as target). We believe that this way of selecting the benchmarks reduces the bias w.r.t. other real models this process may be applied to in the future.

Another threat to the external validity could be that the proposed process does not scale well to models larger than those considered in the experiments. In particular, the time for computing the fitness function grows exponentially with the number of features, as it is based on BDD construction. In order to address this problem, as future work, we plan to devise a technique that, given a single change requirement c, identifies the sub-tree *st* of the feature model *affected* by c, i.e., c can be achieved by only modifying *st*; only

[20] We performed a non-parametric test as we found, with the Shapiro-Wilk test, that the distributions are not normal.

considering *st* would allow to improve the process performance, as the mutations would be more targeted and the fitness computation would be performed on a smaller BDD.

Another threat to the external validity is that the update requests that we use in the experiments could be different by those written by SPL designers. However, update requests have been obtained by computing the difference of consecutive versions of feature models written by SPL designers. While change requirements F_{TBR}, F_{add}, and F_{rem} are guaranteed to be the same as those specified by SPL designers, C_{relax} and C_{rem} are only semantically equivalent. However, this is not a threat, as the evolutionary approach only considers the semantics of the C_{relax} and C_{rem}, not their structure.

Internal validity

Regarding internal validity, a threat is that the obtained results could depend on the values chosen for the parameters of the evolutionary process (parameters of termination conditions, and parameters of the selection and evolution phases) and that, with some other values, the results would have been different (e.g., a given selection strategy could perform better); although we kept all the parameters fixed, we believe that the overall result that our approach is able to actually update the feature model is not affected. However, as future work, we plan to perform a wider set of experiments in which the effect of each single parameter is evaluated. For finding the best parameter setting, we could use a parameter-tuning framework as irace [139].

5.1.6 Related work

Different approaches have been proposed for updating and/or repairing feature models.

In a previous work, we proposed a technique to generate *fault-detecting configurations* (tests) able to show *conformance faults* (i.e., configurations wrongly accepted or wrongly rejected) in feature models [32]; in [33], we then presented an iterative process based on mutation that first shows these fault-detecting configurations to the user who must assess their correct evaluation, and then modifies the feature model to remove the faults (if any). The approach proposed here is different, since it is based on an evolutionary approach, it is completely automatic, and does not require the interaction with the user who must only provide the initial update request. Moreover, in the current approach we consider update requests not only coming from failing tests but also from the normal evolution of the SPL.

Another approach trying to remove faults from feature models is presented in [107]: it starts from a feature model and, through a cycle of *test-and-fix*, improves it by removing its wrong constraints; the approach uses configurations derived both from the model and from the real system and checks whether these are correctly evaluated by the feature model. The approach is similar to ours in considering wrong configurations, but does not allow to add and remove features. The main differences with our approach are that we have a precise definition of target we need to reach, we rely on an evolutionary approach, and we assume that the model evolution can be obtained through mutation.

In [34], we proposed an approach to repair variability models by modifying the constraints of the model using some *repairs*; that approach differs from the one presented in this work in different aspects. First of all, the oracle (similar to our target) in [34] is given by the implementation constraints, while here the target comes from update requests. Then, the aim of [34] is only to remove faults from the model, while here we also support the evolution of the model. Finally, the approach in [34] always improves the conformity index (similar to our fitness function) during the process, with the risk of obtaining local optima; in the current approach, instead, we maintain a set of candidate solutions in which some of them may decrease the fitness function in some iteration, but that could obtain a better result at the end.

Regarding the use of evolutionary algorithms for feature models, the work in [137] proposes a process to reverse engineer feature models starting from a set of products: the process starts from a population of randomly generated models and evolves it using as fitness function the number of correctly evaluated products. The approach is similar to ours in using an evolutionary approach based on mutation (some used mutation operators are similar to ours), but differs in the aim and in the starting point: we start from an existing feature model that we want to update to achieve some change requirements (removing faults or business requirements), while the approach in [137] wants to build a new feature model starting from some known products.

Evolutionary approaches have been widely used also for testing and repairing programs. For example, GenProg [132] is a repair tool based on genetic programming. It uses mutation and crossover operators to search for a program variant that passes all tests.

Model $\mathcal{M}$	Implementation (System $\mathcal{S}$):

$A \to B$
$A \to C$
	`#ifdef C //Hello`
	`char* msg = "Hello !";`
	`#endif`
	`#ifdef B // Bye`
	`char* bye = "Bye";`
	`#endif`
	`#ifdef A // lowercase`
	`msg [0] = 'h';`
	`bye [0] = 'b';`
	`#endif`

Figure 22: Example of problem and solution spaces

5.2 A Process for Fault-Driven Repair of Constraints Among Features

Constraints exist among system features. They can prohibit system configurations that are dangerous or undesired, or can describe conditions leading to certain properties or errors in code, such as *preprocessor errors*, *parser errors*, *type errors*, and *feature effect* [153]. Designers, developers, and testers can greatly benefit from modelling features and constraints among them, as it allows to reduce development effort [168] and to identify corner cases of the system under test.

Constraints among features can be modeled using *variability models*, and imposed on the implementation by means of preprocessor directives, makefiles, etc. These two ways of modeling variability are usually known as *problem space* and *solution space* [153]. Fig. 22 presents an example of a variability model containing the constraints among three system features (A, B, and C) that are implemented as preprocessor directives in the C program.

This separation between problem and solution space allows users to model configuration without knowledge about low-level implementation details. On the other hand, these two spaces need to be consistent; code and models, however, are often not kept synchronized, and *repairs* are needed. In the evolution of product lines, two common types of repair are performed: *debugging and program repair*, when the variability model is correct but the implementation has to be fixed; and *model repair*, when the program is correct, but the variability model is outdated. This latter case occurs when the description of variability is evolved in the implementation, but not in the variability model. This work tackles this problem and proposes a technique to automatically repair variability models. Fig. 23 shows an overview of this context.

In order to detect discrepancies between the problem space and the solution space, classical techniques for testing of propositional formulas (as the constraints of a variability model) can be used: the classical decision and condition coverage, the MCDC [61], fault based criteria [30], and also combinatorial

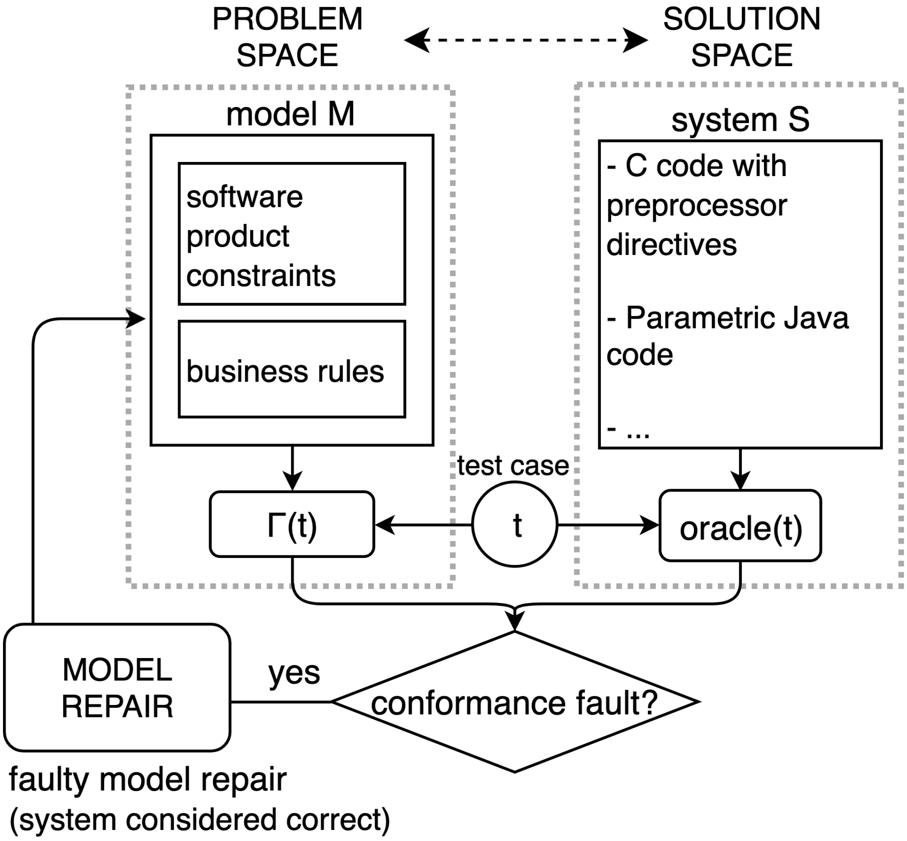


Figure 23: Fault-driven repair of variability models

testing [32]. In this work, we assume that some tests have been generated according to some coverage criterion and some *faults* (i.e., non-conformances of the model w.r.t. the solution space acting as oracle) have been detected; our aim is to *repair* the constraints of the variability model in order to remove the faults. We propose an automated process that, upon some failing tests, is able to *automatically* correct the constraints in such a way that they maintain their original validity for all the configurations, except for those found failing.

The rest of the section reflects the content of the paper [29], and it is organized as follows. Sect. 5.2.1 presents some basic definitions in addition to the ones introduced previously in the chapter. Sect. 5.2.2 presents the basic repair process and some possible optimizations. Sect. 5.1.4 shows the empirical results. Threats to validity are tackled in Sect. 5.2.4, whereas an overview of the related work is given in Sect. 5.2.5. The conclusions, together with lines for future research, are given jointly at the end of the chapter, in Sect. 5.3.

5.2.1 Basic Definitions

[29]

Definition 7 (Variability Model)**.** *A variability model $\mathcal{M}$ is made of a set of features $F = \{f_1, \dots, f_n\}$ and a set of constraints $\Gamma = \{\gamma_1, \dots, \gamma_m\}$ over the features.*

The features F represent the system parameters. The expressions in Γ identify the features configurations for which the actual system is expected to work.

Definition 8 (Configuration)**.** *A* configuration *(or* test*) t is a particular assignment of values for all the features F. We identify with $t(f_i)$ the value of feature f_i in test t. A configuration is valid if it respects the constraints, i.e., $t \models \Gamma$. We also use Γ as predicate to check the constraint satisfaction: $\Gamma(t) = true$ iff $t \models \Gamma$.*

Definition 9 (Test suite)**.** *A* test suite *T is a set of tests. We identify with T_e the* exhaustive test suite*, i.e., the set of all the possible tests.*

Definition 10 (Oracle)**.** *The oracle function $oracle(t)$ tells whether the configuration t is functionally correct for the system S.*

We assume that an oracle exists, that tells whether a configuration is valid or not in the real system.

We assume that the set of features F is known and correctly modeled, while the constraints could be faulty.

Definition 11 (Model correctness)**.** *We say that the model $\mathcal{M}$ is* correct *if it conforms with the oracle for every possible configuration t, i.e., $\forall t \in T_e \colon \Gamma(t) = oracle(t)$.*

Definition 12 (Conformance fault)**.** *We say that the model contains a* conformance fault *if there exists a configuration t such that $\Gamma(t) \neq oracle(t)$.*

Definition 13 (Combination)**.** *A* combination *(or* partial configuration*) c is an assignment to a subset $features(c)$ of all the possible features F, i.e., $features(c) \subseteq F$. A configuration (or test) is thus a particular combination in which $features(c) = F$. The value assigned by the combination c to the feature f is denoted as $c(f)$.*

Definition 14 (Propositional representation of combinations)**.** *A combination c can be expressed in propositional logic by making the conjunction of the truth value assignments of its features:*

$$c = \left(\bigwedge_{\{f \in features(c) | c(f)\}} f \right) \wedge \left(\bigwedge_{\{f \in features(c) | \neg c(f)\}} \neg f \right)$$

Definition 15 (Combination containment)**.** *A test (or configuration) t* contains *a combination c if all features values in c are the same in t. Formally, $\forall f_i \in features(c): c(f_i) = t(f_i)$.*

Given a test suite T, we identify all the tests containing a combination c as $T(c)$. Formally, $T(c) = \{t \in T | c \subseteq t\}$.

Definition 16 (Combination completeness)**.** *Given a test suite T and a combination c, we say that c is* complete *w.r.t. T iff $T(c)$ contains all possible tests containing c.*

Lemma 5.1. *If c is complete w.r.t. a test suite T, then it holds $\forall t \in T_e \setminus T(c): c = false$.*

Definition 17 (Failure-containing combination)**.** *A combination c is a* failure-containing combination *(fcc) if:*

1. *c is contained in at least a failing test of T, i.e., $\exists t \in T(c): \Gamma(t) \neq oracle(t)$.*
2. *every configuration containing c has the same value in the oracle, i.e., $(\forall t \in T(c): oracle(t) = false) \vee (\forall t \in T(c): oracle(t) = true)$. In the former case, we call c an* under-constraining *fcc; in the latter case, an* over-constraining *fcc.*

We further classify a conformance fault as under-constraining fault *if it exposed by an under-constraining fcc, or as* over-constraining fault *if it is exposed by an over-constraining fcc.*

In the following, we only consider complete *fccs*.

Example 4 (Under-constraining *fcc*). Let's assume that the oracle is the system (C program) of Fig. 22 with features $F = \{A, B, C\}$ and that we have generated the test suite shown in Table 15a. Given a faulty model $\mathcal{M}_{f1}$, with only one constraint $\Gamma = \{A \rightarrow B\}$, we observe only one fault which is represented by the *fcc* $c = A \wedge \neg C$.

(a) under-constraining fault

A	B	C	$\mathcal{M}_{f1}$	oracle
T	T	F	T	F
T	F	F	F	F

(b) over-constraining fault

A	B	C	$\mathcal{M}_{f2}$	oracle
F	T	T	T	T
F	F	T	T	T
F	T	F	F	T
F	F	F	F	T

Table 15: Test suites with faults (in gray)

Note that c is a complete *fcc* that identifies an under-constraining fault, i.e., it proves that the model is under-constrained.

Example 5 (Over-constraining *fcc*). Consider now a faulty version $\mathcal{M}_{f2}$ of the model in Fig. 22, characterized by $\Gamma = \{A \rightarrow B, C\}$. Given the test suite shown Table 15b, we detect two over-constraining faults identified by the complete *fcc* $c = \neg A$.

5.2.2 Fault-driven Repair

We here propose a process to *repair* the constraints of a variability model, based on the detection of conformance faults between the model and the system, represented as failure-containing combinations. Fig. 24 shows the context in which our process is applied.

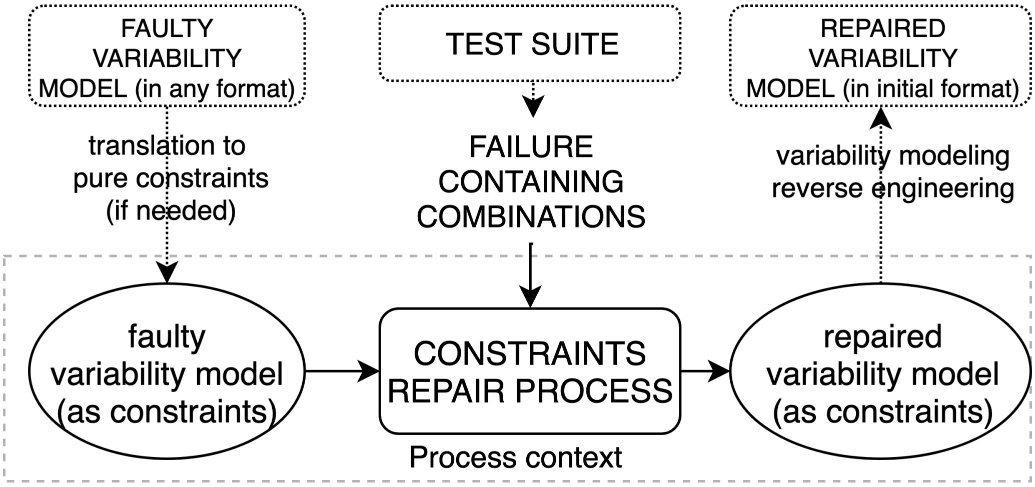

FAULTY VARIABILITY MODEL (in any format)
TEST SUITE
REPAIRED VARIABILITY MODEL (in initial format)
translation to pure constraints (if needed)
FAILURE CONTAINING COMBINATIONS
variability modeling reverse engineering
faulty variability model (as constraints)
CONSTRAINTS REPAIR PROCESS
repaired variability model (as constraints)
Process context

Figure 24: Context of the process to repair constraints among features in variability models

We assume that a possibly faulty variability model is translated to a set of boolean formulas representing the constraints. For example, if the model is a feature model, semantic transformations presented in [38] can be used. From a sufficiently large test suite, complete *fccs* have been identified. To this aim, one can use well-known fault localization techniques like [96, 28]. Our process takes as input the *fccs* and the constraints and repair them. If the user wants to go back to the initial format of the variability model, (s)he must apply some reverse engineering (which is out of the scope of this work).

Naïve repair approach

In Def. 17, we distinguish between two types of failure-containing combinations (i.e., under-constraining and over-constraining *fcc*), depending on how the model fails with respect to the oracle.

We can devise a naïve repair approach that applies a specific type of repair on the base of the fault type:

1. `Strengthening repair`: in case of under-constraining *fcc* c, $\neg c$ is added as a new constraint to Γ, i.e., the constraints set Γ' of the repaired model becomes $\Gamma' := \Gamma \cup \{\neg c\}$.
2. `Weakening repair`: in case of over-constraining *fcc* c, c is disjuncted with every constraint in Γ, i.e., the constraints set Γ' of the repaired model becomes $\Gamma' = \cup_{\gamma_i \in \Gamma}\{\gamma_i \vee c\}$.

Example 6 (Strengthening repair). The *under-constraining* fault in Ex. 4 is repaired by adding $\neg c = \neg(A \wedge \neg C) \equiv A \rightarrow C$ as a new constraint in Γ. The repaired constraints become $\Gamma' = \{A \rightarrow B, A \rightarrow C\}$.

Example 7 (Weakening repair). The *over-constraining fault* in Ex. 4 is repaired by adding $c = \neg A$ in disjunction with all the existing constraints, so that the repaired constraints become $\Gamma' = \{(A \rightarrow B) \vee \neg A, C \vee \neg A\}$. Note that the first constraint is *redundant*, as it is equivalent to the original constraint $A \rightarrow B$. The only necessary application of the repair is the one in the second constraint, as it correctly allows to have both features A and C assigned to *false* (as in the oracle).

Theorem 5.2 (Correctness of the naïve approach)**.** *If a combination c is complete w.r.t. its test suite $T(c)$, the repairs applied by the naïve approach to* Γ *(obtaining the modified constraints set* Γ'*) are correct, i.e., they remove all existing faults in $T(c)$ and do not introduce new ones, i.e.,*

1. $\forall t \in T(c)\colon \Gamma'(t) = oracle(t)$;

2. $\forall t \in T_e \setminus T(c)\colon \Gamma'(t) = \Gamma(t)$.

Proof. Let's consider the two kinds of repairs separately:

- `Strengthening repair`: the repaired constraints are $\Gamma' = \{\gamma_1, \ldots, \gamma_m, \gamma_{m+1}\}$, where $\gamma_{m+1} = \neg c$.
    1. From the definition of under-constraining *fcc*, we know that it holds $\forall t \in T(c)\colon oracle(t) = false$. Furthermore, we also know that $\forall t \in T(c)\colon \Gamma'(t) = false$, because the new constraint $\gamma_{m+1} = \neg c$ falsifies all the tests containing the *fcc* c. Therefore, it holds $\forall t \in T(c)\colon \Gamma'(t) = oracle(t)$.

2. By Lemma 5.1, we know that $\forall t \in T_e \setminus T(c)\colon c = \textit{false}$. Therefore, the added constraint $\gamma_{m+1} = \neg c$ is always true in tests $T_e \setminus T(c)$. Since γ_{m+1} has no influence on the evaluation of these tests, it holds $\forall t \in T_e \setminus T(c)\colon \Gamma'(t) = \Gamma(t)$.

- `Weakening repair`: the repaired constraints are $\Gamma' = \{\gamma_1 \vee c, \ldots, \gamma_m \vee c\}$.
  1. From the definition of over-constraining *fcc*, we know that it holds $\forall t \in T(c)\colon \textit{oracle}(t) = \textit{true}$. Furthermore, we also know that $\forall t \in T(c)\colon \Gamma'(t) = \textit{true}$, because all the constraints $\gamma_i' = \gamma_i \vee c$ admit all the tests containing the *fcc* c. Therefore, $\forall t \in T(c)\colon \Gamma'(t) = \textit{oracle}(t)$.
  2. By Lemma 5.1, we know that $\forall t \in T_e \setminus T(c)\colon c = \textit{false}$. Since c is added as a disjunction to the existing constraints, it leaves the constraints equivalent to the original ones, i.e., $\forall t \in T_e \setminus T(c)\colon \Gamma'(t) = \Gamma(t)$.

□

Optimized repair approach

The naïve repair approach described in Sect. 5.2.2 could generate some redundancy, as shown in Ex. 7. Therefore, we introduce techniques for constraint selection and simplification to reduce the potential redundancy generated by the naïve approach. The goal is to make fewer edits as possible to the model, since we assume that a model with fewer edits better preserves domain knowledge.

Fig. 25 shows the optimized repair approach.

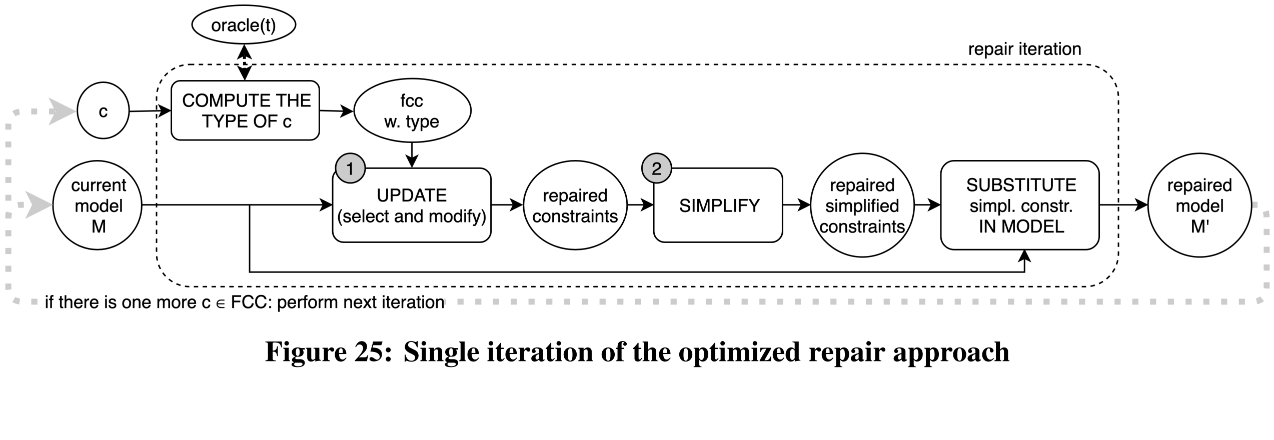

oracle(t)
repair iteration
c
COMPUTE THE TYPE OF c
fcc w. type
current model M
1
UPDATE (select and modify)
repaired constraints
2
SIMPLIFY
repaired simplified constraints
SUBSTITUTE simpl. constr. IN MODEL
repaired model M'
if there is one more c ∈ FCC: perform next iteration

Figure 25: Single iteration of the optimized repair approach

It consists of three phases: (1) selection of some constraints to modify, (2) modification of the selected constraints, and (3) simplification of the modified constraints. Note that the process can be iterative if we identified more than one *fcc* (as in the experiments in Sect. 5.2.3). In the following, we consider one iteration of the process.

Update phase: selection and modification

To preserve the domain knowledge embedded in the constraints, we want the process to make as few changes as possible to them. Namely, we would like that the constraints Γ are updated with the following qualities:

1. possibly no more constraints are added;
2. as many constraints as possible are preserved identical;
3. some constraints can be removed.

Therefore, the process performs a pre-processing phase, which selects only the constraints $\Gamma_S \subseteq \Gamma$ containing some configurations in common with the *fcc* c, and then modifies them. This phase is specific to the type of repair:

Strengthening repair Only the constraints γ_i sharing at least one feature with the *fcc* c, are selected. Formally, $\Gamma_S = \{\gamma_i \in \Gamma | (\mathit{features}(\gamma_i) \cap \mathit{features}(c)) \neq \emptyset\}$, where *features* collects the features contained in a formula. Then, the modification phase updates only one constraint γ_s selected randomly from Γ_S, by conjuncting $\neg c$. The repaired constraints set becomes $\Gamma' = (\Gamma \setminus \{\gamma_s\}) \cup \{\gamma_s \wedge \neg c\}$.

Weakening repair Only the constraints γ_i that exclude at least one configuration contained in the *fcc* c are selected. Formally, $\Gamma_S = \{\gamma_i \in \Gamma | \mathit{isSAT}(\neg\gamma_i \wedge c) = \mathit{true}\}$, where *isSAT* tells whether a formula is satisfiable or not. Then, the modification phase updates all the constraints in Γ_S by disjuncting them with c. The repaired constraints set becomes $\Gamma' = (\Gamma \setminus \Gamma_s) \cup \bigcup_{\gamma_i \in \Gamma_s} \{\gamma_i \vee c\}$.

Simplification phase

The constraint simplification procedure aims at reducing redundancy in the repaired model, especially when failure-containing combinations involve many features. A straightforward way to simplify a formula (or make it more readable) is to find the smallest, but equivalent expression. This problem is known as the *minimum-equivalent-expression* problem [53, 106].

We compared the three existing formula minimization techniques, and one minimization method based on mutations we have implemented (ATGT):

1. JBool[21]: a tool that recursively applies logic rules and preprocessing techniques, preserving equivalence [42]: *literal removal*, *negation simplification*, *and/or reduplica-*

[21] JBool: https://github.com/bpodgursky/jbool_expressions

tion and flattening, *child expression simplification*, and *propagation*, *De Morgan's law*.

2. Quine-McCluskey (QM) [148], a generalization of the Karnaugh Maps method. It requires the constraints to be in Disjunctive Normal Form (DNF) and its exponential complexity in the number of features makes it suitable only for small models (up to 15 features).
3. Espresso[22], a faster version of the QM method that relies on some heuristics [178].
4. ATGT[23] [56]: a hill-climbing process we implemented that iteratively mutates a formula randomly and checks it for equivalence.

We propose different methods, as we have seen that, in practice, these techniques often produce different outputs and none of them is guaranteed to generate an output which is always minimal compared to the others.

Correctness

Does the optimized approach produce correct repairs? A repair r is correct if it is equivalent to the one obtained by the naïve approach.

Theorem 5.3 (Correctness of the optimized approach)**.** *The optimized approach is correct.*

Proof. The techniques applied in the *simplification phase* preserve equivalence. Therefore, we only show that the repairs computed in the *update phase* are equivalent to those computed by the naïve approach, because the two following properties hold:

- The strengthening repair is correct because of distributivity and commutativity of boolean conjunction: $\gamma_1 \wedge \ldots \wedge \gamma_s \wedge \ldots \wedge \gamma_m \wedge \neg c \equiv \gamma_1 \wedge \ldots \wedge (\gamma_s \wedge \neg c) \wedge \ldots \wedge \gamma_m$.
- The weakening repair is by definition equivalent to the naïve approach for all the constraints γ_i that are selected in Γ_S to be modified in the optimized approach (i.e., it performs the same operation of the naïve approach). It is correct also for each non-selected constraints γ_j, since for these constraints it holds $\gamma_j \vee c = \gamma_j$ (as c is *false* in the non-selected constraints): thus, γ_j can be left as it is. In fact, by translating this expression to a satisfiability problem, we obtain the condition under

[22] Espresso logic minimizer, sources available at `https://ptolemy.berkeley.edu/projects/embedded/pubs/downloads/espresso/index.htm`

[23] ATGT: ASM Test Generation Tool. `http://fmse.di.unimi.it/atgtBoolean.html`

which the process does not select the constraint:

$$\begin{aligned}((\gamma_j \vee c) = \gamma_j) &\Leftrightarrow \neg isSAT((\gamma_j \vee c) \neq \gamma_j)\\ &\Leftrightarrow \neg isSAT((\gamma_j \vee c) \oplus \gamma_j)\\ &\Leftrightarrow \neg isSAT((\gamma_j \wedge \neg\gamma_j) \vee (c \wedge \neg\gamma_j) \vee (\neg\gamma_j \wedge \neg c \wedge \gamma_j))\\ &\Leftrightarrow \neg isSAT(\neg\gamma_j \wedge c)\end{aligned}$$

□

5.2.3 Evaluation

In order to apply our process, we need a faulty variability model $\mathcal{M}$, a set of failure-containing combinations *FCC*, and an *oracle*. For the sake of experiments, we take as oracle another variability model $\mathcal{M}_o$, instead of the real oracle; in this way, we can also extract the set *FCC* by comparing $\mathcal{M}$ and $\mathcal{M}_o$.

Benchmarks

We have built two sets of benchmarks: $\texttt{BENCH}_{\texttt{MUT}}$ with seeded faults, and $\texttt{BENCH}_{\texttt{REAL}}$ with versioned models.

$\texttt{BENCH}_{\texttt{MUT}}$ (seeded faults)

In order to build this benchmark set, we first selected some models to be used as $\mathcal{M}_o$, from previous papers and feature model repositories:

- example, from Example 4.
- register, a VSpec model for a register typically found in supermarkets, inspired by [185].
- django, an open source web application framework written in Python. Each Django project has a configuration file loaded at launch time. We considered 12 Boolean parameters (features), with constraints devised in our previous work [87].
- tight_vnc from FeatureIDE repository [149].

In order to obtain the initial faulty model $\mathcal{M}$, we seeded random faults in $\mathcal{M}_o$ using the following mutation operators:

- **RC**: removal of a constraint. There are studies showing that this is the most common case in practice [141].
- **RL**: removal of a literal in a constraint.
- **SL**: substitution of a literal in a constraint.

We generated 30 faulty versions $\mathcal{M}$ of each model $\mathcal{M}_o$ (10 with each mutation operator).

$\mathtt{BENCH}_{\mathtt{REAL}}$ (versioned models)

For this benchmark set, we have considered two versions of variability models of the same system. We use the second version as oracle $\mathcal{M}_o$, and the first one as the faulty model $\mathcal{M}$. We picked three models of industrial applications from the SPLOT repository[24] [151]:

- the process model rhiscom, between versions 2.0 and 3.0;
- an enterprise resource planner (ERP-SPL);
- a windows accessibility module, between versions 7.0 and 8.0.

Table 16 reports the size of all the faulty models $\mathcal{M}$ to be repaired in the two benchmarks, in terms of number of features, number of constraints, and total number of literals in the constraints.

	Name	# features	# constraints
			avg(min - max)
				# literals

$\mathtt{BENCH}_{\mathtt{MUT}}$	example	3	1.67 (1-2)	3.0 (2-4)
	register	3	1.67 (1-2)	3.87 (2-5)
	django	12	4.6 (4-5)	10.87 (9-12)
	tight_vnc	24	11.67 (11-12)	53.2 (45-55)
$\mathtt{BENCH}_{\mathtt{REAL}}$	rhiscom	36	70	140
	ERP-SPL	43	75	151
	windows	335	943	2031

Table 16: Benchmarks size

For the constraints and literals of $\mathtt{BENCH}_{\mathtt{MUT}}$, it reports the average number across the 30 mutants and the minimum and maximum number between parentheses (the number of features is the same across the mutants).

[24] http://52.32.1.180:8080/SPLOT/feature_model_repository.html

Failure-containing combinations

For the sake of experiments, we obtain the set *FCC* from the faulty model $\mathcal{M}$ (having constraints Γ) and the model we use as oracle $\mathcal{M}_o$ (having constraints Γ_o), using the following process:

1. first, we generate a test showing the difference (i.e., conformance fault) between the two models. The test is built as $t = \mathit{getModel}(\Gamma \neq \Gamma_o)$, where *getModel* returns a model of the propositional expression, if it exists, or *null* (in this case, the models are equivalent).
2. then, we start from $c \leftarrow t$, and,
   - if $\Gamma_o(t)$, for each feature $f \in F$, if $\neg \mathit{isSAT}(\neg\Gamma_o \wedge \mathit{rem}(f, c))$ holds, then we do $c \leftarrow \mathit{rem}(f, c)$, where *rem* removes the assignment of f in c and returns the modified c.
   - if $\neg\Gamma_o(t)$, for each feature $f \in F$, if $\neg \mathit{isSAT}(\Gamma_o \wedge \mathit{rem}(f, c))$ holds, then we do $c \leftarrow \mathit{rem}(f, c)$.

This way we can obtain *fccs* that are as minimal as possible, and complete (i.e., the oracle is always true in case of over-constraining *fcc*, and always false in case of an under-constraining *fcc*).

Repair quality metrics

We want to assess the quality of a repair w.r.t. two goals: (i) simplification of the constraints, and (ii) minimization of the impact of edits. To this aim, we introduce two quality metrics that are used to compare Boolean expressions. We apply them to compare the conjunction of the constraints Γ of the original model $\mathcal{M}$ and the constraints Γ' of the repaired model $\mathcal{M}'$ obtained as output of the approach. The metrics are defined as follows:

- Complexity Distance (CD) as difference of formula sizes $CD(\Gamma, \Gamma') = \mathit{literals}(\Gamma) - \mathit{literals}(\Gamma')$, where *literals* returns the number of literals in a formula. As in [205], we also considered other measures (number of operators and node count in the parsed tree representation), but they do not change the overall results, therefore we do not report them here.
- Edit Distance (ED) computed between the syntactic trees of the two formulas Γ and Γ'. $ED(\Gamma, \Gamma')$ is defined as the number of *edits* (addition, substitution, or

elimination) that we have to apply to Γ in order to obtain Γ'. A node of the tree can either be a literal or an operator. We use APTED as a tool to efficiently compute tree edit distances [166].

Experiments

We run experiments on the two benchmark sets $\texttt{BENCH}_{\texttt{MUT}}$ and $\texttt{BENCH}_{\texttt{REAL}}$: namely, we applied the naïve approach (see Sect. 5.2.2), the optimized approach (see Sect. 5.2.2) without the simplification phase (onlySelection), and with the simplification phase (employing the ATGT, Espresso, JBool and QM methods). Experiment code was written in Java and experiments were executed on a Linux PC with Intel(R) i7-3770 CPU (3.4 GHz) and 16 GB of RAM. All reported results are the average of 10 runs with a timeout for a single repair of 1 hour. The code and the benchmarks are available at `https://github.com/fmselab/VMConstraintsRepair`.

Results of the experiments are reported in Table 17.

			fccs and repairs		Naïve			onlySelection			simplification												
											ATGT			Espresso			JBool			QM			
	name	mut.	# (s.+w.)	size (s./w.)	CD	ED	t	CD	ED	t	CD	ED	t	CD	ED	t	CD	ED	t	CD	ED	t	
$\text{BENCH}_{\text{MUT}}$	example	RC	1.0+0.0	2.0 / –	2.0	5.0	0.1	2.0	5.0	0.4	2.0	5.0	1064	2.0	5.0	53.4	2.0	4.0	5.3	2.0	4.0	24.2	
		RL	0.3+1.0	2.0 / 1.0	3.8	10.8	0.2	2.2	6.0	0.3	2.2	6.0	1371	2.2	6.0	68.2	1.6	4.2	1.0	1.6	4.2	30.9	
		SL	0.5+0.3	2.0 / 1.0	1.2	3.2	0.0	1.0	2.6	0.0	1.0	2.6	1060	1.0	3.0	52.4	1.0	2.6	1.1	1.0	2.6	23.3	
	register	RC	1.0+0.0	2.6 / –	2.3	5.6	0.0	2.3	5.6	0.3	2.3	5.6	1065	2.3	5.6	52.4	2.3	4.9	1.0	2.3	4.9	24.0	
		RL	0.2+1.1	2.6 / 1.3	5.5	14.6	0.0	2.9	7.7	0.3	2.5	6.9	1388	2.9	8.5	68.1	2.2	6.6	0.6	2.2	6.6	30.8	
		SL	0.9+0.3	2.1 / 1.4	2.9	7.4	0.2	2.1	5.2	0.2	2.1	5.2	1433	2.1	6.0	69.2	2.1	5.7	1.2	2.1	5.7	32.9	
	django	RC	0.5+0.0	1.7 / –	0.8	2.2	0.0	0.8	2.2	0.0	0.8	2.2	1062	0.8	2.2	52.2	0.8	1.3	1.0	0.8	1.3	24.2	
		RL	0.4+1.4	2.0 / 4.0	8.0	22.8	0.0	1.6	4.4	0.6	1.2	3.2	2424	1.6	4.4	119.9	1.2	3.0	2.5	1.2	3.2	58.1	
		SL	0.8+2.3	1.0 / 4.0	33.3	94.4	0.0	6.5	18.3	2.1	5.8	16.6	3720	6.5	19.2	180.3	7.4	21.7	4.2	5.8	18.4	116.0	
	tight_vnc	RC	4.7+0.0	2.7 / –	14.0	39.1	0.1	14.0	39.1	11.0	14.0	39.1	8252	14.0	39.1	267	14.0	27.1	23.5	14.0	39.6	11199	
		RL	0.7+31.8	1.9 / 14.2	2422	6000	1.2	202.2	499.5	402	–	–	T/O	202.2	499.5	2275	–	–	T/O	–	–	T/O	
		SL	1.5+18.5	4.1 / 11.2	3734	8860	0.7	244.0	585.8	165	–	–	T/O	244.0	585.8	1369	–	–	T/O	–	–	T/O	
$\text{BENCH}_{\text{REAL}}$	rhiscom	–	9+6	1.2 / 35.3	13977	30634	2	197	504	128	–	–	T/O	197	511	2717	–	–	T/O	–	–	T/O	
	ERP-SPL	–	9+232	2.0 / 37.1	723426	1604116	15	16562	37273	8555	–	–	T/O	16562	37273	16562	–	–	T/O	–	–	T/O	
	windows	–	989+55	2.3 / 453.8	8537492	17028426	87	175380	1918927	114028	–	–	T/O	174200	1917875	245532	–	–	T/O	–	–	T/O	

Table 17: Experimental results (mut.: mutation type; s.: strengthening repairs; w.: weakening repairs; ED: edit distance; CD: complexity distance; t: time in milliseconds, T/O: timeout occurred). In gray the best results (CD and ED over all the approaches, time over the simplification approaches)

For benchmarks $\texttt{BENCH}_{\texttt{MUT}}$, results are categorized by the type of mutation. For each benchmark model, the table reports the number and size of strengthening and weakening repairs (note that each repair corresponds to one *fcc*); moreover, for each process setting, it reports the execution time, and the quality of the final model $\mathcal{M}'$ in terms of *CD* and *ED* distances. Values of the strategies ATGT, JBool and QM for repairing RL and SL mutations of tight_vnc and for all the benchmarks of $\mathtt{BENCH}_{\mathtt{REAL}}$ are not reported, because the experiment exceeded the timeout (T/O) of 1 hour.

We evaluate the process using three research questions.

RQ6: *Which quality do the constraints repaired by the process have?*

The main goal of this repair process is to not destroy domain knowledge. We consider the quality measures ED and CD to be proxies for domain knowledge preservation, under the assumption that having fewer edits means more preservation of the domain knowledge contained in the constraints. We therefore consider an approach *better* than another approach if it has smaller values of the quality measures.

The process (in all its versions) completely repairs all the benchmarks models, as the *fccs* in *FCC* are complete (see Thms. 5.2 and 5.3). However, the quality of the repaired models depends on the adopted repair approach. The optimized approach only using selection (onlySelection) always outperforms the naïve process in terms of quality of the repairs, as it modifies a subset of the constraints, and so the two measures CD and ED for it are always lower. The simplification approaches sometimes allow to obtain better repairs than onlySelection, meaning that they remove some redundancy introduced by the repair; however, there is no simplification method that is always better than the others on all the benchmarks for both measures (except for ATGT that is never worse than Espresso). We observe that, in a few cases, CD and ED are higher for a simplification method w.r.t. onlySelection (e.g., ED of Espresso for register RL): we have checked the example and we found that the simplification has removed some redundancy that was already present in the original model $\mathcal{M}$ so modifying the model more than what done by onlySelection.

RQ7: *How efficient is the repair approach?*

Computational time varies significantly for the different approaches and models: from 0-0.1ms of the smallest models, up to 245 seconds for the windows model (the biggest model having 335 features and 943 constraints) repaired with the Espresso simplifier.

The naïve approach, and the optimized approach without simplification (onlySelection), have been the fastest approaches; execution times of onlySelection are higher than the naïve approach for big models, as it uses a SAT solver for identifying the constraints that must be repaired (see Sect. 5.2.2). Simplification algorithms are the main responsible for slow performance in terms of computation time. ATGT is the slowest one, as it internally calls a SAT solver several times, and the algorithm is yet in a prototypical stage. The second slowest simplification method is Espresso, but we noticed that it is relatively faster than other methods for large models; indeed, it is the only approach able to simplify all the benchmark models, while the others cannot simplify the biggest mutations obtained for tight_vnc and all the models in $\texttt{BENCH}_{\texttt{REAL}}$ in the given timeout. For small models, JBool is the fastest simplification method, followed by QM; however, they both timeout for large models.

RQ8: *Is there a repair type that our process handles more efficiently?*

We are here interested in investigating whether there is an effect of the type of repair on the performances of the process. In order to better understand the computational cost of the type of repairs, Table 18 reports, for $\texttt{BENCH}_{\texttt{REAL}}$, the average execution times of strengthening and weakening repairs in the selection and simplification phases, and also the average time of any repair (regardless of the type).

	avg. repair time per single repair (ms)					
	selection			simplification		
name	str.	wea.	avg.	str.	wea.	avg.
rhiscom	2.6	4.7	3.4	57	54.3	55.9
ERP-SPL	4.3	17.7	17.2	119.8	123.3	123.2
windows	49.5	697.2	83.6	106.9	104.6	106.8

Table 18: Detailed results of the execution time for $\texttt{BENCH}_{\texttt{REAL}}$

We only report the results of Espresso, as it is the only tool that completes before the timeout of 1 hour.

We observe that, in the selection phase, strengthening repairs are faster than weakening repairs: indeed, the former ones only do a syntactical analysis of the constraint, while the latter ones need to call a SAT solver (see Sect. 5.2.2). The average repair time in the selection phase is then influenced by the number of repairs of the two types: in ERP-SPL, since almost all the repairs are weakening (see Table 17), the average time is mostly influenced by them; in windows, instead, most of the repairs are strengthening (see Table 17) and so the average repair time is influenced by them.

Regarding the simplification phase, there is no significant difference between the two types of repairs.

5.2.4 Threats to Validity

We discuss the threats to the validity of our results along two dimensions.

External Validity Regarding external validity, a first threat comes from the choice of the variability models on which we performed the experiments. In the benchmarks, we totally selected models of seven applications of different sizes, among them two industrial applications from the SPLOT public repository. Although we have not tested our process on bigger feature models, we believe that the number and variety of input data make the results of our evaluation generalizable to other models of similar size.

In $\texttt{BENCH}_{\texttt{REAL}}$, we *simulated* real faults in constraints by enumerating the failure-containing combinations (and thus the single repairs) between two versions of constraints. Such simulated faults may not be accurate with respect to real usages in some scenarios. However, we believe that such results may be generalizable in cases when the faults are automatically detected by testing the *updated* system implementation, with respect to an *outdated* model, as we believe that the second version of the model accurately reflects the underlying system implementation.

Internal Validity Regarding internal validity, a first threat involves the number of experiments and the accuracy of results. To this aim, we executed the experiments 10 times.

Another threat comes from the metrics used to assess the effectiveness and efficiency of our approach, not being a good proxy for domain knowledge preservation. We believe, however, that the chosen metrics well represent the concepts of formula readability and impact of the changes (ED), that may be useful for successive reasoning, for a reverse engineering process from propositional formula to feature model, and for comprehension by the user.

Regarding our approach in general, we have identified the following two threats to validity. The first one regards the applicability of our repair technique. We assume that the variability model is given as a set of constraints, while in general other formats (like feature models) are widely used. However, it is almost always possible to extract the set of features and the constraints among them, so our approach is generally applicable. It is true that it may be not easy to go back from the repaired model to the original format (see Fig. 24), but we try to change the model as little as possible. This should ease the identification of the applied repairs and facilitate the reverse process to extract the final variability model in another format.

The second threat regards the assumption of the *fccs* completeness. In general, we may find some conformance faults, but it may be not enough, since our process assumes that the *fccs* are complete. However, we can notice that every failing test is a complete *fcc* regardless of the test suite. Trying to extract a smaller failing combination from a failing test possibly requires new tests, but there already exist several techniques for fault localization that efficiently can do that [96].

5.2.5 Related Work

There exist methods to statistically infer constraints from sampled configurations [60, 8, 194, 195, 196, 16]: they use a classifier to infer the conditions among parameter values, that determine a particular property, either a parameter above/below a certain threshold (like in [195]), or directly the configuration being accepted or rejected by the system (as in [196]). These machine-learning based methods are well-documented and supported by application studies to real scenarios, such as learning constraints among parameters in SCAD programs that may cause defects of configurable objects to 3D print [16]; and, in the case of LATEX, showing that it is possible to obtain constraints among Boolean or numerical values, to format the paper to meet desired properties, such as a defined page limit [9]. Another interesting application of inferring constraints using machine learning is the case of mining temporal and value constraints from rich logs, for event-based monitoring in industrial SoS (Systems of Systems) [122]. The approach, integrated also with techniques from process mining and specification mining, was applied to the automation system of a metallurgical company, and it consists of a *ranking* phase, based on some validity ratio metrics on the classification tree outcome, in which constraints that are more likely to be accepted by the users appear first.

The constraints learned (or *mined*) with such machine learning methods achieve a good accuracy (greater than 80% on average [195]), and they are able to completely *infer* constraints from scratch, or to *specialize* the model by adding the new inferred constraints to the existing ones. Our approach, however, is focused on performing any kind of repair to an existing set of constraints, and is also able to *generalize* the model, or apply *arbitrary edits* (as described in the classification in [199]). Moreover, our proposed method focuses only on the *manipulation* of existing constraints, and not on the actual detection of the failure-containing combinations, that we assume as input of our process. Unlike our approach, that takes the failure-containing combinations as input, those ML processes also include automatic detection of such *fccs* (in the form of constraints), given a sample of configurations classified as valid or non-valid [195]. For these reasons, the approaches are not alternative but complementary, as our approach is not comparable to those ML methods; however, as future work, we believe that it could be interesting to combine our process with those machine learning approaches, to have a more complete process for real case scenarios.

A quality-based model refactoring framework assessing the quality of merging operations among SPL models, expressed in UML [177], supports maintainability of models describing relations among features. It represents another approach to model repair, although it is not focused on repairing constraints in propositional logic. There is a comprehensive general work on repair of models by Reder et al. [174], with a method to detect inconsistencies using a validator, and to generate a repair tree representing in a compact way all the different viable actions to repair the model. This approach has been evaluated on UML models and OCL design rules, and is currently integrated in the Model/Analyzer plug-in for the IBM Rational Software Architect (RSA).

We believe that our approach, instead, is a particular case of such repair framework in which the repair actions are fixed and determined by the selection and simplification algorithms (i.e., Espresso, QM, etc.), whereas the inconsistency detection is left to the engineer, who has to provide a set of failure-containing combinations in input to our process. However, despite our process has a fixed repair type and in this work we evaluated its application, it may be possible to integrate the idea of [174] and build a sort of repair-action tree in which the *fccs* are applied in different order, for example, or with a different simplification method for each *fcc*, and we believe that the result of following another path in that repair-action tree could give slightly different results (that could be better or could also be worse).

Program repair techniques, such as SemFix [154], GenProg [132], and Par [118] already apply successive patch transformations, but to repair single faults in the code directly.

A process to detect and repair feature models from *conformance faults* with respect to another model has been presented in [33]. Our work, however, is able to handle arbitrary constraints of a variability model, and adds also the simplification of such modified constraints. The need for a fault-driven constraint repair process was already envisioned in [107], but no experiments were yet performed.

In the classification of edits to variability models presented in [141], our process fits the categories *build fix* and *adherence to changes in code*; in the classification of edits to variability models presented in [199], our process is able to address all kind of edits: in the case of *arbitrary edits*, it achieves them by applying *specialization* (what we call *strengthening repair*) and *generalization* (what we call *weakening repair*) sequentially.

A different technique for feature model repair in the context of system evolution, with different versions of systems, used mutation operators to make the model meet a specific *update request* [27, 26]; however, that approach does not handle arbitrary constraints, and does not guarantee to completely fulfil the update request, and thus to repair the model. We believe that that approach could be extended with our process, to be able to *repair* not only the feature *tree*, but the constraints as well.

Repair of constraints has usage also in other contexts, such as the repair of parameter values of timed automata clock guards, by applying tests and specializing the constraints [20]; and in the detection of constraints among parameters that let the built attack string trigger an XSS vulnerability in the system [93].

5.3 Conclusion

We proposed a process that, given a (faulty or outdated) variability model, and the faults in terms of failure-containing combinations, identifies the constraints involved in the fault, repairs them according to the oracle value, and simplifies them to make the edit minimal. We conducted an empirical evaluation on 7 models of different sizes, and found that the process of selecting only some constraints is indeed more effective than the naïve approach that modifies all of them. Moreover, we observed that simplification approaches can further improve the quality of the repair. However, their applicability is limited by the model size, as most of them do not scale on big models.

As future work, we plan to adapt our approach to larger models and to include in the evaluation the performances of reverse engineering the final constraints into a variability model (in case the repairs affected the structure of the initial variability model). As future work, we also want to address the current limitations; for example, by designing better selection and simplification strategies, by extending the method to non-boolean variables, and by including new simplification techniques. Furthermore, in order to better preserve domain knowledge, we plan to design an approach that interacts with domain engineers, for instance by highlighting implicit constraints as in [19].

Chapter 6. Repair of Timed Automata

In this Chapter, we present the application of the envisioned repair process to timed automata. Timed automata (TAs) [13] are a widely used formalism to specify systems having temporal requirements.

Timed automata is proposed by Alur and Dill [13] in the early nineties as an extension to the classical automata, which is extended with clock variables. Timed automata theory has become an important research area and been widely studied in the context of both formal languages and verification of real time systems [120].

One main application of timed automata is the verification of properties using model checking methods [143], by traversing through all reachable states. Another application is robustness analysis of systems that can be modeled by a TA [145].

Timed automata have academic and industrial applications for the modeling and verification of protocols: such as TDMA (Time Division Multiple Access) protocol [135], power controller protocols [105], and security protocols such as Kerberos, TMN, Neumann Stubblebine, Andrew Secure and Wide Mouthed Frog [113, 112].

Timed automata are equipped with of clock variables that record the passage of time since they have been reset. All clocks are synchronized and they run at the same speed. The transitions from one state to another are assumed to be instantaneous; in other words, time passes only in states, not on edges.

An example of time automaton is given in Fig. 26a.

A formal definition of timed automata is given in Sect. 6.1.

One problem of timed automata modeling, however, is that exactly specifying the system may be difficult, as the user may not know the exact clock constraints triggering state transitions. To offer a support for the user in his/her modeling tasks, under the assumption that the user only wrote wrong guard transitions (i. e., the structure of the TA is correct, but the guard constraints may not be correct), the search space for the *correct* TA (w.r.t. the system) can be represented by a Parametric Timed Automaton (PTA), i. e., a TA in which some constants are parametrized.

We selected to repair only an aspect of timed automata, namely clock guards, as they have a direct impact that can be revealed by testing (using *timed words* as tests, as described later in the chapter), tests are meaningful because only one aspect is supposed to change (the structure of the timed automata is assumed to remain the same, although we also perform an experiment on a small structure change), and we believe they are in reality subject to change in system evolution, together with the number of locations and transitions.

This chapter reflects the content of the paper [20], that introduces a process that (i) abstracts the initial (faulty) TA ta_{init} in a PTA *pta*; (ii) generates some test data (i. e., timed traces) from *pta*; (iii) assesses the correct evaluation of the traces with the oracle; (iv) uses the IMITATOR tool for synthesizing some constraints φ on the parameters of *pta*; (v) instantiate from φ a TA ta_{rep} as final repaired model. Experiments show that the approach is successfully able to partially repair the initial design of the user.

Problem A common usage of timed automata is to develop a TA describing the running system and then apply analysis techniques to it (e. g., [40]). However, exactly specifying the system under analysis may be difficult, as the user may not know the exact clock constraints that trigger state transitions, or may perform errors at design time. Therefore, validating the produced TA against the real system is extremely important to be sure that we are analyzing a faithful representation of the system. Different testing techniques have been proposed for timed automata, based on different coverage criteria as, e. g., transition coverage [190] and fault-based coverage [11, 12], and they can be used for TA validation. However, once some failing tests have been identified, it remains the problem of detecting and removing (*repair*) the fault from the TA under validation. How to do this in an automatic way is challenging. One possible solution could be to use mutation-based approaches [11, 12] in which mutants are considered as possible repaired versions of the original TA; however, due to the continuous nature of timed automata, the number of possible mutants (i. e., repair actions) is too big also for small TAs and, therefore, such approaches do not appear to be feasible. We here propose to use a *symbolic representation* of the possible repaired TAs and we reduce the problem of repairing to finding an assignment of this symbolic representation.

Contribution In this work, we address the problem of testing/validating TAs under the assumption that only clock guards may be wrong, that is, we assume that the structure (states and transitions) is correct. Moreover, we assume to have an oracle that we can query for acceptance of timed traces, but whose internal structure is unknown: this oracle can be a Web-service, a medical device, a protocol, etc. In order to symbolically represent the search space of possible repaired TAs, we use the formalism of parametric timed automata (PTAs) [15] as an abstraction to represent all possible behaviors under all possible clock guards.

We propose a framework for automatic repair of TAs that takes as input a TA ta_{init} to repair and an *oracle*. The process works as follows:

1. starting from ta_{init}, we build a PTA *pta* where to look for the repaired TA;

2. we build a symbolic representation of the language accepted by *pta* in terms of an *extended parametric zone graph* $\mathcal{EPZG}$;

3. we then generate some test data *TD* from $\mathcal{EPZG}$;

4. we assess the correct evaluation of *TD* by querying the *oracle*, so building the test suite *TS*;

5. we feed the tests *TS* to the IMITATOR tool that finds some constraints φ that restrict *pta* only to those TAs that correctly evaluate all the tests in *TS*;

6. as the number of TAs that are correct repairs may be infinite, we try to obtain, using a constraint solver based on local search, the TA ta_{rep} closest to the initial TA ta_{init}. Note that trying to modify as little as possible the initial TA is reasonable if we assume the competent programmer hypothesis [162].

To evaluate the feasibility of the approach, we performed some preliminary experiments showing that the approach is able to (partially) repair a faulty TA.

Outline In the rest of the chapter, Sect. 6.1 explains the definitions we need in our approach. Then 6.2 presents the process we propose that combines model abstraction, test generation, constraint generation, and constraint solving. 6.3 describes experiments we performed to evaluate our process. Finally, 6.4 reviews some related work, and 6.5 concludes the chapter.

6.1 Definitions

[20] A *timed word* [13] over an alphabet of actions Σ is a possibly infinite sequence of the form $(a_0, d_0)(a_1, d_1) \cdots$ such that, for all integer $i \geq 0$, $a_i \in \Sigma$ and $d_i \leq d_{i+1}$. A timed language is a (possibly infinite) set of timed words.

We assume a set $\mathbb{X} = \{x_1, \dots, x_H\}$ of *clocks*, i. e., real-valued variables that evolve at the same rate. A clock valuation is $\mu : \mathbb{X} \to \mathbb{R}_{\geq 0}$. We write $\vec{0}$ for the clock valuation assigning 0 to all clocks. Given $d \in \mathbb{R}_{\geq 0}$, $\mu + d$ is s.t. $(\mu + d)(x) = \mu(x) + d$, for all $x \in \mathbb{X}$. Given $R \subseteq \mathbb{X}$, we define the *reset* of a valuation μ, denoted by $[\mu]_R$, as follows: $[\mu]_R(x) = 0$ if $x \in R$, and $[\mu]_R(x) = \mu(x)$ otherwise.

We assume a set $\mathbb{P} = \{p_1, \dots, p_M\}$ of *parameters*. A parameter *valuation* v is $v : \mathbb{P} \to \mathbb{Q}_+$. We assume $\bowtie \in \{<, \leq, =, \geq, >\}$. A *clock guard* g is a constraint over $\mathbb{X} \cup \mathbb{P}$ defined by a conjunction of inequalities of the form $x \bowtie \sum_{1 \leq i \leq M} \alpha_i p_i + d$, with $p_i \in \mathbb{P}$, and

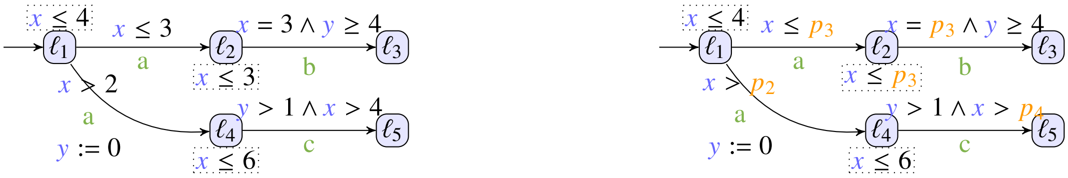
$\alpha_i, d \in \mathbb{Z}$. Given g, we write $\mu \models v(g)$ if the expression obtained by replacing each x with $\mu(x)$ and each p with $v(p)$ in g evaluates to true.

6.1.1 Parametric timed automata

Definition 6.1 (PTA)**.** *A PTA $\mathcal{A}$ is a tuple $\mathcal{A} = (\Sigma, L, \ell_0, F, \mathbb{X}, \mathbb{P}, I, E)$, where: i) Σ is a finite set of actions, ii) L is a finite set of locations, iii) $\ell_0 \in L$ is the initial location, iv) $F \subseteq L$ is the set of accepting locations, v) $\mathbb{X}$ is a finite set of clocks, vi) $\mathbb{P}$ is a finite set of parameters, vii) I is the invariant, assigning to every $\ell \in L$ a clock guard $I(\ell)$, viii) E is a finite set of edges $e = (\ell, g, a, R, \ell')$ where $\ell, \ell' \in L$ are the source and target locations, $a \in \Sigma$, $R \subseteq \mathbb{X}$ is a set of clocks to be reset, and g is a clock guard.*

Given $e = (\ell, g, a, R, \ell')$, we define $\mathsf{Act}(e) = a$.

Example 6.1. *Consider the PTA in 26b, containing two clocks x and y and three parameters p_2, p_3 and p_4. The initial location is ℓ_1.*



(a) A TA to be repaired **(b) An abstract PTA**

Figure 26: Running example

Given v, we denote by $v(\mathcal{A})$ the non-parametric structure where all occurrences of a parameter p_i have been replaced by $v(p_i)$.

We denote as a *timed automaton* any such structure $v(\mathcal{A})$.

The *synchronous product* (using strong broadcast, i. e., synchronization on a given set of actions), or *parallel composition*, of several PTAs gives a PTA.

Definition 6.2 (Concrete semantics of a TA)**.** *Given a PTA $\mathcal{A} = (\Sigma,\ L,\ \ell_0,\ F,\ \mathbb{X},\ \mathbb{P},\ I,\ E)$, and a parameter valuation v, the semantics of $v(\mathcal{A})$ is given by the timed transition system (TTS) $(S, s_0, \rightarrow)$, with*

- $S = \{(\ell, \mu) \in L \times \mathbb{R}^H_{\geq 0} \mid \mu \models v(I(\ell))\}$, $s_0 = (\ell_0, \vec{0})$,

- $\rightarrow$ *consists of the discrete and (continuous) delay transition relations: i) discrete transitions:* $(\ell, \mu) \overset{e}{\mapsto} (\ell', \mu')$*, if* $(\ell, \mu), (\ell', \mu') \in S$*, and there exists* $e = (\ell, g, a, R, \ell') \in E$*, such that* $\mu' = [\mu]_R$*, and* $\mu \models v(g)$*. ii) delay transitions:* $(\ell, \mu) \overset{d}{\mapsto} (\ell, \mu + d)$*, with* $d \in \mathbb{R}_{\geq 0}$*, if* $\forall d' \in [0, d], (\ell, \mu + d') \in S$*.*

Moreover we write $(\ell, \mu) \xrightarrow{(e,d)} (\ell', \mu')$ for a combination of a delay and discrete transition if $\exists \mu'' : (\ell, \mu) \overset{d}{\mapsto} (\ell, \mu'') \overset{e}{\mapsto} (\ell', \mu')$.

Given a TA $v(\mathcal{A})$ with concrete semantics $(S, s_0, \rightarrow)$, we refer to the states of S as the *concrete states* of $v(\mathcal{A})$. A *run* of $v(\mathcal{A})$ is an alternating sequence of concrete states of $v(\mathcal{A})$ and pairs of edges and delays starting from the initial state s_0 of the form $s_0, (e_0, d_0), s_1, \cdots$ with $i = 0, 1, \ldots$, $e_i \in E$, $d_i \in \mathbb{R}_{\geq 0}$ and $s_i \xrightarrow{(e_i,d_i)} s_{i+1}$. The *associated timed word* is $(\mathsf{Act}(e_0), d_0)(\mathsf{Act}(e_1), \sum_{0 \leq i \leq 1} d_i) \cdots$. A run is *maximal* if it is infinite or cannot be extended by any discrete action. The (timed) language of a TA, denoted by $\mathcal{L}(v(\mathcal{A}))$, is the set of timed words associated with maximal runs of $v(\mathcal{A})$. Given $s = (\ell, \mu)$, we say that s is reachable in $v(\mathcal{A})$ if s appears in a run of $v(\mathcal{A})$. By extension, we say that ℓ is reachable in $v(\mathcal{A})$; and by extension again, given a set T of locations, we say that T is reachable if there exists $\ell \in T$ such that ℓ is reachable in $v(\mathcal{A})$.

Example 6.2. *Consider the TA* $\mathcal{A}$ *in 26a. Consider the following run* ρ *of* $\mathcal{A}$*:*

$$\left(\ell_1, \begin{pmatrix} x = 0 \\ y = 0 \end{pmatrix}\right), (a, 2.5), \left(\ell_4, \begin{pmatrix} x = 2.5 \\ y = 0 \end{pmatrix}\right), (c, 2), \left(\ell_5, \begin{pmatrix} x = 4.5 \\ y = 2 \end{pmatrix}\right)$$

We write "$x = 2.5$*" instead of "*μ *such that* $\mu(x) = 2.5$*". The associated timed word is* $(a, 2.5)(c, 4.5)$.

6.1.2 Reachability synthesis

We will use reachability synthesis to solve the problem in 6.2. This procedure, called EFsynth, takes as input a PTA $\mathcal{A}$ and a set of target locations T, and attempts to synthesize all parameter valuations v for which T is reachable in $v(\mathcal{A})$. EFsynth$(\mathcal{A}, T)$ was formalized in e. g., [117] and is a procedure that traverses the parametric zone graph of $\mathcal{A}$; EFsynth may not terminate (because of the infinite nature of the graph), but computes an exact result (sound and complete) if it terminates.

Example 6.3. *Consider again the PTA $\mathcal{A}$ in 26b. EFsynth$(\mathcal{A}, \{\ell_5\})$ returns $0 \leq p_2 < 4 \wedge 0 \leq p_4 \leq 6 \wedge p_3 \geq 0$. Intuitively, this corresponds to all parameter constraints in the parametric zone graph in 28 associated to symbolic states with location ℓ_5 (there is a single such state).*

6.2 A repairing process using abstraction and testing

In this work, we address the *guard-repair* problem of timed automata.

Given a reference TA ta_{init} and an oracle $\mathcal{O}$ knowing an unknown timed language $\mathcal{TL}$, our goal is to modify ("repair") the timing constants in the clock guards of $\mathcal{A}$ such that the repaired automaton matches the timed language $\mathcal{TL}$. The setting assumes that the oracle $\mathcal{O}$ can be queried for acceptance of timed words by $\mathcal{TL}$; that is, $\mathcal{O}$ can decide whether a timed word belongs to $\mathcal{TL}$, but the internal structure of the object leading to $\mathcal{TL}$ (e. g., an unknown timed automaton) is unknown. This setting makes practical sense when testing black-box systems.

guard-repair problem:

Input: an initial TA ta_{init}, an unknown timed language $\mathcal{TL}$

Problem: Repair the constants in the clock guards of ta_{init} so as to obtain a TA ta_{rep} such that $\mathcal{L}(ta_{rep}) = \mathcal{TL}$

While the ultimate goal is to solve this problem, in practice the best we can hope for is to be *as close as possible* to the unknown oracle TA, notably due to the undecidability of language equivalence of timed automata [13] (e. g., if $\mathcal{TL}$ was generated by another TA).

6.2.1 Overview of the method

From now on, we describe the process we propose to automatically repair an initial timed automaton ta_{init}. 27 describes the approach:

Step ① a PTA *pta* is generated starting from the initial TA ta_{init}.

Step ② the extended parametric zone graph $\mathcal{EPZG}$ (an extension of $\mathcal{PZG}$) is built.

Step ③ a test generation algorithm generates relevant test data *TD* from $\mathcal{EPZG}$.

Step ④ *TD* is evaluated using the oracle, therefore building the test suite *TS*.

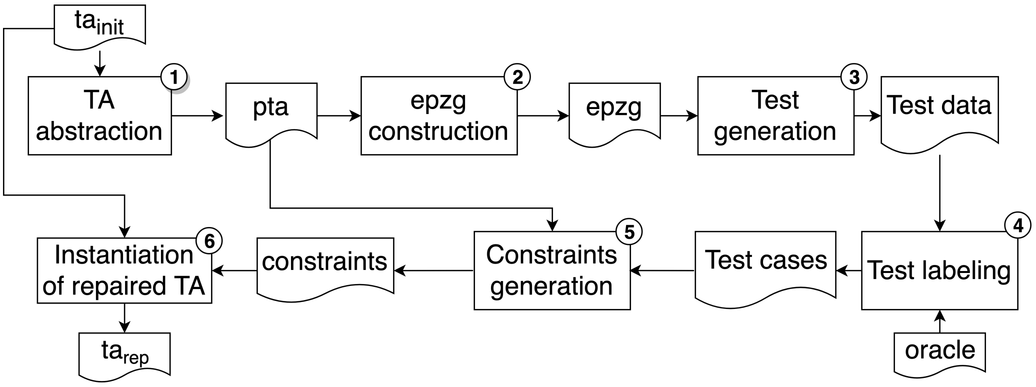


Figure 27: Automatic repair process

Step ⑤ some constraints φ are generated, restricting *pta* to the TAs that evaluate correctly the generated tests *TS*.

Step ⑥ one possible TA satisfying the constraints φ is obtained.

1 formalizes steps ①–⑤ for which we can provide some theoretical guarantees (i. e., the non-emptiness of the returned valuation set, and its inclusion of $\mathcal{TL}$).

Algorithm 1 Automatic repair process $\mathsf{Repair}(ta_{init}, \mathcal{O})$

```
Require: $ta_{init}$: initial timed automaton to repair
Require: $\mathcal{O}$: an oracle assessing the correct evaluation of timed words
Ensure: $\varphi$: set of valuations repairing $ta_{init}$
1: $pta \leftarrow \mathsf{AbstractInPta}(ta_{init})$
2: $\mathcal{EPZG} \leftarrow \mathsf{BuildEpzg}(pta)$
3: $TD \leftarrow \mathsf{GenerateTestData}(\mathcal{EPZG})$        ▷ Generate test data from $\mathcal{EPZG}$
4: $TS \leftarrow \mathsf{LabelTests}(TD, \mathcal{O})$        ▷ A test is a pair (trace, assessment)
       return $\varphi \leftarrow \mathsf{GenConstraints}(pta, TS)$
```

Algorithm 2 GenConstraints(pta, TS)

```
1: $MBA \leftarrow \{w \mid (w, \text{true}) \in TS\}$        ▷ Tests that must be accepted
2: $MBR \leftarrow \{w \mid (w, \text{false}) \in TS\}$        ▷ Tests that must be rejected
       return $\bigwedge_{w \in MBA} \mathsf{ReplayTW}(pta, w) \wedge \bigwedge_{w \in MBR} \neg\mathsf{ReplayTW}(pta, w)$
```

For step ⑥, instead, different approaches could be adopted: in this work, we discuss a possible one. We describe each phase in details in the following sections.

6.2.2 Step ①: Abstraction

Starting from the initial ta_{init}, through *abstraction*, the user obtains a PTA *pta* that generalizes ta_{init} in all the parts that can be possibly changed in order to repair ta_{init} (1 in 1). For instance, a clock guard with a constant value can be parametrized. Therefore, *pta* represents the set of all the TAs that can be obtained when repairing ta_{init}. *pta* is built on the base of the domain knowledge of the developer who has a guess of the guards that may be faulty.

Example 6.4. *Consider again the TA in 26a. A possible abstraction of this TA is the PTA in 26b, where we chose to abstract some of the timing constants with parameters. Note that not all timing constants must necessarily be substituted with parameters; also note that a same parameter can be used in different places (this is the case of p_3).*

Assumption We define below an important assumption for our method; we will discuss in 6.2.8 how to lift it.

Assumption 1. *We here assume that pta is a correct abstraction, i. e., it contains a TA that precisely models the oracle. That is, there exists v_{oracle} such that $\mathcal{L}(v_{oracle}(pta)) = \mathcal{TL}$.*

Note that this assumption is trivially valid if faults lay in the clock guards (which is the setting of this work), and if all constants used in clock guards are turned to parameters.

6.2.3 Step ②: construction of the extended parametric zone graph

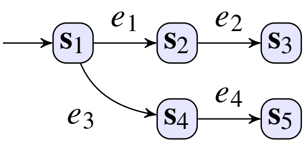
Starting from *pta*, we build a useful representation of its computations in terms of an *extended parametric zone graph* $\mathcal{EPZG}$ (2 in 1). This original data structure will be used for test generation. In the following, we describe how we build $\mathcal{EPZG}$ from $\mathcal{PZG}$.

We extend the parametric zone graph $\mathcal{PZG}$ with the two following pieces of information:

the parameter constraint characterizing each symbolic state: from a state (ℓ, C), the parameter constraint is $C{\downarrow_{\mathbb{P}}}$ and gives the exact set of parameter valuations for which there exists an equivalent concrete run in the automaton. That is, a state (ℓ, C) is reachable in $v(\mathcal{A})$ iff $v \models C$ (see [117] for details).

the minimum and maximum arrival times: that is, we compute the minimum (m_i) and maximum (M_i) over all possible parameter valuations of the possible absolute times reaching this symbolic state.

While the construction of the first information is standard, the second one is original to our work and requires more explanation. We build for each state a (possibly unbounded) interval that encodes the absolute minimum and maximum arrival time. This can be easily obtained from the parametric zone graph by adding an extra clock never reset (that



$$
\begin{aligned}
\mathbf{s}_1 &= (\ \ell_1\ , 0 \leq x = y \leq 4 \land p_2 \geq 0 \land p_3 \geq 0 \land p_4 \geq 0\ )\\
\mathbf{s}_2 &= (\ \ell_2\ , 0 \leq x = y \leq p_3 \land p_2 \geq 0 \land p_3 \geq 0 \land p_4 \geq 0\ )\\
\mathbf{s}_3 &= (\ \ell_3\ , x = y \geq p_3 \land p_2 \geq 0 \land p_3 \geq 4 \land p_4 \geq 0\ )\\
\mathbf{s}_4 &= (\ \ell_4\ , p_2 < x \leq 6 \land y \geq 0 \land p_2 < x - y \leq 4 \land 4 > p_2 \geq 0 \land p_3 \geq 0 \land p_4 \geq 0\ )\\
\mathbf{s}_5 &= (\ \ell_5\ , p_2 < x \land p_4 < x \land y > 1 \land p_2 < x - y \leq 4 \land 4 > p_2 \geq 0 \land p_3 \geq 0 \land 6 > p_4 \geq 0\ )
\end{aligned}
$$

Figure 28: Parametric zone graph of 26b

encodes the absolute time), and projecting the obtained constrained on this extra clock, thus giving minimum and maximum times over all possible parameter valuations.

Example 6.5. *Consider again the PTA $\mathcal{A}$ in 26b and its parametric zone graph in 28. The parameter constraints associated to each of the symbolic states, and the possible absolute reachable times, are given in 19.*

Symbolic states	Parameter constraint	Reachable times
$\mathbf{s}_1$	$p_2 \geq 0 \land p_3 \geq 0 \land p_4 \geq 0$	$x_{abs} = 0$
$\mathbf{s}_2$	$p_2 \geq 0 \land p_3 \geq 0 \land p_4 \geq 0$	$x_{abs} \in [0, 4]$
$\mathbf{s}_3$	$p_2 \geq 0 \land p_3 \geq 4 \land p_4 \geq 0$	$x_{abs} \in [4, \infty)$
$\mathbf{s}_4$	$4 > p_2 \geq 0 \land p_3 \geq 0 \land p_4 \geq 0$	$x_{abs} \in (0, 4]$
$\mathbf{s}_5$	$4 > p_2 \geq 0 \land p_3 \geq 0 \land 6 > p_4 \geq 0$	$x_{abs} \in (1, 6]$

Table 19: Description of the states of the extended parametric zone graph

Remark 6.1. *If all locations of the original PTA contain an invariant with at least one inequality of the form $x \triangleleft p$ or $x \triangleleft d$, with $\triangleleft \in \{<, \leq\}$, and if the parameters are bounded, then the maximum arrival time in each symbolic state will always be finite. Note that this condition is not fulfilled in 6.5 because ℓ_2 features an invariant $x \leq p_3$, with p_3 unbounded, thus allowing to remain arbitrarily long in ℓ_2 for an arbitrarily large value of p_3. Therefore, the arrival time in ℓ_3 is $x_{abs} \in [4, \infty)$.*

6.2.4 Step ③: Test data generation

Starting from $\mathcal{EPZG}$, we generate some test data (3 in 1) in terms of timed words.

Constructing timed words

We use the minimal and maximum arrival times in the abstract PTA to generate test data. That is, we will notably use the *boundary* information, i. e., runs close to the fastest and slowest runs, to try to discover the actual timing guards of the oracle.

The procedure to generate a timed word from the EPZG is as follows:

1. Pick a run $\mathbf{s}_0 e_0 \mathbf{s}_1 \cdots \ldots \mathbf{s}_n$ from $\mathcal{EPZG}$.

2. Construct the timed word $(a_0, d_0)(a_1, d_1) \cdots (a_{n-1}, d_{n-1})$, where $a_i = \mathsf{Act}(e_i)$ and d_i belongs to the interval of reachable times associated with symbolic state $\mathbf{s}_{i+1}$, for $0 \leq i \leq n-1$. Note that, depending on the policy (see below), we sometimes choose on purpose valuations *outside* of the reachable times.

Given an EPZG, we generate, for each finite path of the EPZG up to a given depth K, one timed word. In order to chose a timed word from a (symbolic) path of the EPZG, we identified different policies.

Policies

For each $k < K$, we instantiate $(a_0, d_0)\ (a_1, d_1) \cdots\ (a_k, d_k)$ by selecting particular values for each d_i using different policies:

- $\mathsf{P}_{\pm 1}$: $d_j \in I_{\pm 1}$, where $I_{\pm 1} = \{m_i - 1, m_i, m_i + 1, M_i - 1, M_i, M_i + 1\}$ and m_i and M_i are the minimum and maximum arrival times of the symbolic state.
- $\mathsf{P}_{\mathsf{minMax2}}$: $d_j \in I_{minMax2}$ with $I_{minMax2} = I_{\pm 1} \cup \{(m_i + M_i)/2\}$.
- $\mathsf{P}_{\mathsf{minMax4}}$: $d_j \in I_{minMax4}$ with $I_{minMax4} = I_{minMax2} \cup \{m_i + (M_i - m_i)/4, m_i + ((M_i - m_i)/4) * 3\}$.
- $\mathsf{P}_{\mathsf{rnd}}$: d_j being a random value such that $m_i \leq d_j \leq M_i$.

Example 6.6. *Consider again the PTA $\mathcal{A}$ in 26b and its parametric zone graph in 28 together with reachable times in 19. Pick the run $\mathbf{s}_1 e_3 \mathbf{s}_4 e_4 \mathbf{s}_5$. First note that $\mathsf{Act}(e_3) = \mathrm{a}$ and $\mathsf{Act}(e_4) = \mathrm{c}$. According to 19, the reachable times associated with $\mathbf{s}_4$ are $(0, 4]$ while those associated with $\mathbf{s}_5$ are $(1, 6]$. Therefore, a possible timed word generated with $\mathsf{P}_{\pm 1}$ is $(\mathrm{a}, 1)(\mathrm{c}, 5)$. Note that this timed word does not belong to the TA to be repaired (26a) because of the guard $x > 2$; however, it does belong to an instance TA of 26b for a sufficiently small value of p_2 (namely $v(p_2) < 1$).*

6.2.5 Step ④: Test labeling

Then, every test sequence in *TD* is checked against the oracle in order to label it as accepted or not (4 in 1), therefore the test suite *TS*; a test case in *TS* is a pair $(w, O(w))$, being w a timed word, and $O(w)$ the evaluation of the oracle, i. e., $O(w)$ is defined as a Boolean the value of which is $w \in \mathcal{TL}$.[1] A test case *fails* if $ta_{init}(w) \neq O(w)$, i. e., the initial TA and

[1] To limit the number of tests, we only keep the maximal accepted traces (i. e., we remove accepted traces included in longer accepted traces), and the minimal rejected traces (i. e., we remove rejected traces having as prefix another rejected trace).

the oracle timed language disagree. Note that, if is no test case fails, ta_{init} is considered correct[2] and the process terminates.

In different settings, different oracles can be used. In this work, we assume that the oracle is the real system of which we want to build a faithful representation; the system is black-box, and it can only be queried for acceptance of timed words. In another setting, the oracle could be the user who can easily assess which words should be accepted, and wants to validate their initial design. Of course, the type of oracle also determines how many test data we can provide for assessment: while a real implementation can be queried a lot (modulo the time budget and the execution time of a single query), a human oracle usually can evaluate only few tests.

6.2.6 Step ⑤: Generating constraints from timed words

Given the test suite *TS*, our approach generates constraints φ that restrict *pta* to only those TAs that correctly evaluate the tests (4 in 1).

In this section, we explain how to "replay a timed word", i. e., given a PTA $\mathcal{A}$, how to synthesize the exact set of parameter valuations v for which a finite timed word belongs to the timed language of $v(\mathcal{A})$. Computing the set of parameter valuations for which a given *finite* timed word belongs to the timed language can be done easily by exploring a small part of the symbolic state space. Replaying a timed word is also very close to the ReplayTrace procedure in [24] where we synthesized valuations corresponding to a trace, i. e., a timed word without the time information—which is decidable.

From timed words to timed automata

First, we convert the timed word into a (non-parametric) timed automaton. This straightforward procedure was introduced in [23], and simply consists in converting a timed word of the form $(a_1, d_1), \cdots, (a_n, d_n)$ into a sequence of transitions labeled with a_i and guarded with $x_{abs} = d_i$ (where x_{abs} measures the absolute time, i. e., is an extra clock never reset). Let TW2PTA denote this procedure.[3]

[2] This does not necessarily mean that both TAs have the same language, but that the tests did not exhibit any discrepancy.

[3] This procedure transforms the word to a non-parametric TA; we nevertheless use the name TW2PTA for consistency with [23].

Example 6.7. *Consider again the timed word w mentioned in 6.6: $(a, 0.5)(c, 5)$. The result of TW2PTA(w) is given in 29.*

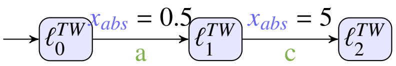


Figure 29: Translation of timed word $(a, 0.5)(c, 5)$

Synchronized product and synthesis

The second part of step ⑤ consists in performing the synchronized product of TW2PTA(w) and $\mathcal{A}$, and calling EFsynth on the resulting PTA with the last location of the timed word as the target of EFsynth. Let ReplayTW(pta, w) denote the entire procedure of synthesizing the valuations associated that make a timed word possible.

Example 6.8. *Consider again the PTA $\mathcal{A}$ in 26b and the timed word $(a, 0.5)(c, 5)$ translated to a (P)TA in 29. The result of EFsynth applied to the synchronized product of these two PTAs with $\{\ell_2^{TW}\}$ as target location set is*

$$0 \leq p_4 < 5 \wedge 0 \leq p_2 < \frac{1}{2} \wedge p_3 \geq 0.$$

This set indeed represents all possible valuations for which $(a, 0.5)(c, 5)$ is a run of the automaton. Note that the result can be non-convex. If we now consider the simpler timed word $(a, 3)$, then the result of ReplayTW$(\mathcal{A}, w)$ becomes

$$p_3 \geq 3 \wedge p_2 \geq 0 \wedge p_4 \geq 0 \quad \vee \quad p_2 < 3 \wedge p_3 \geq 0 \wedge p_4 \geq 0$$

This comes from the fact that the action a can correspond to either e_1 (from ℓ_1 to ℓ_2) or e_3 (from ℓ_1 to ℓ_4) in 26b.

Remark 6.2. *Despite the non-guarantee of termination of the general EFsynth procedure, ReplayTW not only always terminates, but is also very efficient in practice: indeed, it only explores the part of the PTA corresponding to the sequence of (timed) transitions imposed by the timed word.*

Lemma 6.1. *Let pta be a PTA, and w be a timed word. Then ReplayTW(pta, w) terminates.*

6.2.7 Correctness

Recall that 1 assumes that there exists a valuation v_{oracle} such that $\mathcal{L}(v_{oracle}(pta)) = \mathcal{TL}$. We show that, under 1, our resulting constraint is always non-empty and contains the valuation v_{oracle}.

Theorem 6.1. *Let* $\varphi = \mathsf{Repair}(ta_{init}, O)$. *Then* $\varphi \neq \bot$ *and* $v_{oracle} \models \varphi$.

6.2.8 Step ⑥: Instantiation of a repaired TA

Any assignment satisfying φ characterizes a correct TA w.r.t. the generated tests in *TS*; however, not all of them exactly capture the oracle behavior. If the user wants to select one TA, (s)he can select one assignment v_{rep} of φ, and use it to instantiate the final repaired TA ta_{rep}.

In order to select one possible assignment v_{rep}, different strategies may be employed, on the base of the assumptions of the process. In this work, we assume the *competent programmer hypothesis* [162] that the developer produced an initial TA ta_{init} close to be correct; therefore, we want to generate a final TA ta_{rep} that is *not too different* from ta_{init}. In particular, we assume that the developer did small mistakes on setting the values of the clock guards.

In order to find the closest values of the clock guards that respect the constraints, we exploit the local search capability of the constraint solver Choco [171]:

1. we start from the observation that ta_{init} is an instantiation of *pta*. We therefore select the parameter evaluation v_{init} that generates ta_{init} from *pta*, i. e., $ta_{init} = v_{init}(pta)$;
2. we initialize Choco with v_{init}; Choco then performs a local search trying to find the assignment closest to v_{init} that satisfies φ.

Discussing 1

1 assumes that the user provides a PTA *pta* that contains the oracle. If this is not the case, the test generation phase (6.2.6) may generate a negative test (i. e., not accepted by any instance of *pta*) that is instead accepted by the oracle or a positive test that is not accepted by the oracle; in this case, the constraints generation phase would produce an unsatisfiable constraint φ. In this case, the user should refine the abstraction by parameterizing some other clock guards, or by relaxing the constraints on some existing parameters.

Moreover, it could be that the correct oracle has a different structure (additional states and transitions): as future work, we plan to apply other abstractions as CoPtA models [142] that allow to parametrize states and transitions.

Note that, even if the provided abstraction is wrong, our approach could still be able to refine it. In order to do this, we must avoid to use for constraint generation (step

⑤) tests that produce unsatisfiable constraints. We use a greedy incremental version of GenConstraints in which ReplayTW is called incrementally: if the constraint generated for a test w is not compatible with the other constraints generated previously, then it is discarded; otherwise it is conjuncted.

6.3 Experimental evaluation

In order to evaluate our approach, we selected some benchmarks from the literature to be used as initial TA ta_{init}: the model of a coffee machine (CF) [12], of a car alarm system (CAS) [12], and the running case study (RE). For each benchmark model, 20 reports its number of locations and transitions.

Benchmark	size of ta_{init}		# params	SD	SC (%)
	#locs.	#trans.			
RunningEx (RE)	5	3	5	2	98.33
Coffee (CF)	5	6	9	11	99.18
CarAlarmSystem (CAS)	16	10	10	12	84.24
RunningEx – different oracle (RE$_{do}$)	5	3	5	-	98.72

Table 20: Benchmarks

The proposed approach requires that the developer, starting from ta_{init}, provides an abstraction in terms of a PTA pta. For the experiments, as we do not have any domain knowledge, we took the most general case and we built pta by adding a parameter for each guard constant; the only optimization that we did is to use the same parameter when the same constant is used on entering and/or exiting transitions of the same location (as in 26b).

In the approach, the oracle should be the real system that we can query for acceptance; in the experiments, the oracle is another TA ta_o that we obtained by slightly changing some constants on the guards. The oracle has been built in a way that it is an instance of pta, following 1.

In order to measure *how much* a TA (either the initial one ta_{init} or the final one ta_{rep}) is different from the oracle, we introduce a syntactic and a semantic measure, that provide different kinds of comparison with the oracle ta_o.

Given a model *ta*, the oracle ta_o, and a PTA *pta* having parameters $p_1, \ldots, p_n$, let v and v_o be the corresponding evaluations, i. e., $ta = v(pta)$ and $ta_o = v_o(pta)$. We define the *syntactic distance* of *ta* to the oracle as follows:

$$SD(ta) = \sum_{i=1}^{n} |v(p_i) - v_o(p_i)|$$

The syntactic distance roughly measures how much *ta* must be changed (under the constraints imposed by *pta*) in order to obtain ta_o.

The *semantic conformance*, instead, tries to assess the distance between the languages accepted by *ta* and the oracle ta_o. As the set of possible words is infinite, we need to select a representative set of test data TD_{SC}; to this aim, we generate, from ta_{init} and ta_o, sampled test data in the two TAs; moreover, we also add negative tests by extending the positive tests with one forbidden transition at the end. The semantic conformance is defined as follows:

$$SC(ta) = \frac{|\{t \in TD_{SC} | (t \in \mathcal{L}(ta) \land t \in \mathcal{L}(ta_o)) \lor (t \notin \mathcal{L}(ta) \land t \notin \mathcal{L}(ta_o))\}|}{|TD_{SC}|}$$

20 also reports *SD* and *SC* of each benchmark ta_{init}.

Experiments have been executed on a Mac OS X 10.14, Intel Core i3, with 4 GiB of RAM. Code is implemented in Java, IMITATOR 2.11 “Butter Kouign-amann” is used for constraint generation, and Choco 4.10 for constraint solving. The code and the benchmarks are available at `https://github.com/ERATOMMSD/repairTAsThroughAbstraction`.

6.3.1 Results

21 reports the experimental results.

For each benchmark and each test generation policy (see 6.2.4), it reports the execution time (divided between the different phases), the total number of generated tests, the number of tests that fail on ta_{init}, and *SD* and *SC* of the final TA ta_{rep}.

We now evaluate the approach answering the following research questions.

RQ1: *Is the approach able to repair faulty TAs?*

Bench.	Policy	time (s)					# failed tests/	ta_{rep}	
		total	Steps ②-③	Step ④	Step ⑤	Step ⑥	# tests	SD	SC (%)
RE	$\mathsf{P}_{\pm 1}$	1.070	0.010	0.008	1.030	0.019	1/ 38	0	100.00
RE	$\mathsf{P}_{\mathsf{minMax2}}$	1.148	0.007	0.006	1.130	0.005	1/ 41	0	100.00
RE	$\mathsf{P}_{\mathsf{minMax4}}$	1.191	0.004	0.004	1.177	0.004	1/ 41	0	100.00
RE	$\mathsf{P}_{\mathsf{rnd}}$	0.006	0.006	0.001	0.000	0.000	0/ 3	2	98.33
CF	$\mathsf{P}_{\pm 1}$	25.921	0.050	0.267	25.546	0.045	45/293	8	99.86
CF	$\mathsf{P}_{\mathsf{minMax2}}$	32.717	0.129	0.578	31.845	0.147	62/422	7	100.00
CF	$\mathsf{P}_{\mathsf{minMax4}}$	76.137	0.857	1.907	73.058	0.769	102/737	7	100.00
CF	$\mathsf{P}_{\mathsf{rnd}}$	0.134	0.098	0.035	0.000	0.000	1/ 11	8	99.96
CAS	$\mathsf{P}_{\pm 1}$	59.511	0.043	0.160	59.261	0.037	174/392	2	100.00
CAS	$\mathsf{P}_{\mathsf{minMax2}}$	61.791	0.040	0.159	61.544	0.036	199/416	2	100.00
CAS	$\mathsf{P}_{\mathsf{minMax4}}$	68.341	0.716	0.467	67.037	0.584	245/464	2	100.00
CAS	$\mathsf{P}_{\mathsf{rnd}}$	0.024	0.017	0.007	0.000	0.000	0/ 20	12	84.24

Table 21: Experimental results

We evaluate whether the approach is actually able to (partially) repair ta_{init}. From the results, we observe that, in three cases out of four, the process can completely repair RE since SD becomes 0, meaning that we obtain exactly the oracle (therefore, also SC becomes 100%). For CF and CAS, it almost always reduces the syntactical distance SD, but it never finds the exact oracle. On the other hand, the semantic conformance SC is 100% in five cases. Note that SC can be 100% with SD different from 0 for two reasons: either the test data TD_{SC} we are using for SC are not able to show the unconformity, or ta_{rep} is indeed equivalent to the oracle, but with a different structure of the clock guards.

RQ2: *Which is the best test generation strategy?*

In 6.2.4, we proposed different test generation policies over $\mathcal{EPZG}$. We here assess the influence of the generation policy on the final results. $\mathsf{P}_{\pm 1}$, $\mathsf{P}_{\mathsf{minMax2}}$, and $\mathsf{P}_{\mathsf{minMax4}}$ obtain the same best results for two benchmarks (RE and CAS), meaning that the most useful tests are those on the boundaries of the clock guards: those are indeed able to expose the failure if the fault is not too large. On the other hand, for CF, $\mathsf{P}_{\pm 1}$ performs slightly worse than the other two, meaning that also generating tests inside the intervals (as done by $\mathsf{P}_{\mathsf{minMax2}}$ and $\mathsf{P}_{\mathsf{minMax4}}$) can be beneficial for repair. $\mathsf{P}_{\mathsf{rnd}}$ is able to improve (but not totally repair) only CF; for the other two benchmarks, instead, it is not able to improve neither SD nor SC.

RQ3: *How long does the approach take?*

The time taken by the process depends on the size of ta_{init} and on the test generation policy. The most expensive phase is the generation of the constraints, as it requires to call IMITATOR for each test that must be accepted. As future work, we plan to optimize this phase by modifying IMITATOR to synthesize valuations guaranteeing the acceptance of multiple timed words in a single analysis. In the experiments, we use as oracle another TA that we can visit for acceptance; this visit is quite fast and so step ④ does not take too much time. However, in the real setting, the oracle is the real system whose invocation time may be not negligible; in that case, the invocation of the oracle could become a bottleneck and we would need to limit the number of generated tests.

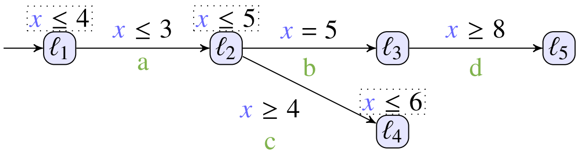


Figure 30: Repairing TAs with different structures – Another oracle TA

RQ4: *Which is the process performance if pta does not include the oracle?*

1 assumes that the user provides a PTA that contains the oracle. In 6.2.8, we discussed about the possible consequences when this assumption does not hold. We here evaluate whether the approach is still able to partially repair ta_{init} using an oracle having a different structure. We took the TA shown in 30 as oracle of the running example, that is structurally different from ta_{init} and *pta* shown in 26 (we name this experiment as RE_{do}); the semantic conformance *SC* of ta_{init} w.r.t. the new oracle is shown at the last row of 21[4]. We performed the experiments with the new oracle using the greedy approach described in 1, and results are reported in 22. We observe that policies $\mathsf{P}_{\pm 1}$, $\mathsf{P}_{\mathsf{minMax2}}$, and $\mathsf{P}_{\mathsf{minMax4}}$,

Bench.	Policy	time (s)					# failed tests/	ta_{rep}
		total	Steps ②-③	Step ④	Step ⑤	Step ⑥	# tests	*SC* (%)
RE_{do}	$\mathsf{P}_{\pm 1}$	2.083	0.007	0.005	2.055	0.013	10/ 44	98.72
RE_{do}	$\mathsf{P}_{\mathsf{minMax2}}$	2.718	0.005	0.004	2.693	0.013	11/ 47	98.72
RE_{do}	$\mathsf{P}_{\mathsf{minMax4}}$	2.686	0.004	0.003	2.658	0.012	11/ 47	98.72
RE_{do}	$\mathsf{P}_{\mathsf{rnd}}$	0.763	0.007	0.001	0.676	0.075	1/ 3	99.5

Table 22: Experimental results – Different oracle

[4] Note that it does not make sense to measure the syntactical distance, as the structure of the oracle is different.

although they find some failing tests, they are not able to improve *SC*. This is partially due to the fact that *SC* is computed on some timed words TD_{SC} that may be not enough to judge the improvement. On the other hand, as the three policies try to achieve a kind of coverage of *pta* (so implicitly assuming 1), it could be that they are not able to find *interesting* failing tests (i. e., they cannot be repaired); this seems to be confirmed by the fact that the random policy $\mathsf{P}_{\mathsf{rnd}}$ is instead able to partially repair the initial TA using only three tests, out of which one fails. We conclude that, if the assumption does not hold, trying to randomly select tests could be more efficient.

6.4 Related Work

Testing timed automata

Works related to ours are approaches for test case generation for timed automata. In [12, 10], a fault-based approach is proposed. The authors defined 8 mutation operators for TAs and a test generation technique based on bounded-model checking; tests are then used for model-based testing to check that System Under Test (SUT) is conformant with the specification. Our approach is different, as we aim at building a faithful representation of the SUT (i. e., the oracle). Their mutation operators could be used to repair our initial TA, as done in [27]; however, due to continuous nature of TAs, the possible mutants could be too many. For this reason, our approach symbolically represents all the possible variations of the clock guards (similar to “change guard” mutants in [12]). Other classical test generation approaches for timed automata are presented in [190, 108]; while they aim at coverage of a single TA, we aim at coverage of a family of TAs described by *pta*.

Learning timed systems In a different direction, learning timed systems has been studied in the past. Learning consists in retrieving an unknown language, with membership and equivalence queries made to a teacher. The (timed) language inclusion is undecidable for timed automata [13], making learning impossible for this class. In [100, 134], timed extensions of the L^* algorithm [25] were proposed for event-recording automata [14], a subclass of timed automata for which language equivalence can be decided. Learning is essentially different from our setting, as the system to be learned is usually a white-box system, in which the equivalence query can be decided. In our setting, the oracle does not necessarily know the structure of the unknown system, and simply answers membership queries. In addition, we address in our work the full class of timed automata, for which learning is not possible.

6.5 Conclusions

In this section we propose an approach for automatically repairing timed automata clock guards. Our approach first parameterizes the initial timed automata, abstracting it into a PTA, generates some tests, and then refines the abstraction by identifying only those TAs contained in the PTA that correctly evaluate all the tests.

As future work, we plan to adopt also other formalisms to build the abstraction where to look for the repaired timed automata; The CoPtA model [142], for example, extends timed automata with feature models and allows to specify additional/alternative states and transitions. In addition, when the oracle acts as a white-box, i. e., when the oracle is able to test language equivalence, we could also make use of learning techniques for timed automata despite the undecidability of the language inclusion problem, using the almost-always terminating procedure for language inclusion in [207].

Chapter 7. CTWEDGE: Migrating Combinatorial Interaction Test Modeling and Generation to the Web

This chapter presents a tool to support combinatorial interaction testing (CIT) generation. Our model repair process can be performed only when faults are localized, and testing, in particular combinatorial testing, is a way to generate highly informational tests (i.e., it allows to achieve high coverage with very few tests) for configurable models, in an automated way. The presence of tools for combinatorial interaction testing (CIT) is of fundamental importance because performing CIT activities manually can be error prone and time consuming. The site pairwise.org[1] lists at least 43 tools supporting several CIT activities while the paper [156] reviewed the CIT literature and found around 20 tools.

Most of the tools are classical programs or plugins of existing programs/platforms. In all these cases, the user has to download, install, and execute the program on his/her machine. Some tools offer a GUI interface, for example, for defining the models and run the test generation, while others rely on other programs for some activities (like PICTMaster, based on PICT [67], that works entirely inside Microsoft Excel). In [90] the authors presented a plugin for the eclipse IDE that helps the user in writing CIT models with constraints by leveraging all the expected features of a modern IDE, and it generates combinatorial test suites by using third party programs (like CASA or ACTS). However, this classical approach poses some challenges. First, the user must install the chosen CIT tool, which in turn may require some dependencies, like java, or in case of plugin, it requires the installation of another program or application like eclipse, excel and so on. Then the user must use his/her machine for running the test generation algorithms. For experienced users with powerful machines they can administer, this is not a real problem. However, for novice users with not so powerful computer, or students that are just learning the CIT principles, or software developers using computer they cannot administer, this can become a cost to be considered and it may be an obstacle for the use of CIT.

Second, from the point of view of tool developers, the distribution of programs means that they have little or no control on the software once is installed: when a bug is discovered the developer has to fix the bug, publish a new version of the tool and hope that the users will update their software. Moreover, the developer has no idea about how the software is used: what are the typical scenarios of use (big or small models, for example), what are the features that are mostly used (for example, what modeling features are more used). Furthermore, it is difficult for tool developers to apply a cost model able to

[1] See `http://www.pairwise.org/tools.asp`

reward their effort in developing and deploying the CIT tool. Indeed, most of the tools are given away for free by researchers supported by their own organization.

A possible solution of these problems could be the offer of CIT features as software as a service (SaaS). SaaS is software that is accessed through Internet by using a classical web browser. The SaaS software is actually hosted on the vendor's servers, and the customers log in and perform tasks as necessary. The vendor is the one who is ultimately responsible for hosting, upgrading and maintaining the program as needed. There is an extensive literature about SaaS and the advantages (together with limitations) are well known [152].

There are already experiences in using the web for CIT. For instance, CTWeb Classic [203], CTWeb Plus [5], TestCover [197], Hexawise [2], and PairWiser [3].

However, as we will discuss in the related work section, each tool has its own specific language to define combinatorial models, with its own predefined output formats and a (limited) set of supported existing or in-house algorithms for test case generation. Not all those tools are equally powerful or easy to use, and many of them are commercial or require registration.

For this reason, starting from CitLab [90], we have worked on a web based application that allows the user to write CIT models in a similar way he/she would do in a classical IDE and it offers test suite generation by means of server using known and community evaluated test generation algorithms. Our system, called ctwedge (Combinatorial Testing Web EDiting and GEneration), introduces a rich language for combinatorial models, offers a powerful web editor, and allows the user to generate the CIT test suites on a server. The only software needed to use ctwedge is a modern web browser.

The chapter reflects the content of the paper [89] and it is organized as follows. In Sect. 7.1 we present the modeling language (in an abstract way) we use to define CIT models. In Sect. 7.2 we introduce our tool, its architecture, its web editing capabilities, and the generator engine. A detailed comparison with other similar web based tools is presented in Sect. 7.3. Future works and possible directions of extension are presented in Sect. 7.4. Conclusions are reported in Sect. 7.5.

7.1 A simple language for CIT models

> *[89]* We have devised a simple textual language for CIT models which is suitable to be used in web editors. It allows the definition of parameters, each with its name and (finite) domain, and it is based on our previous language defined for CitLab [90]. We allow the following parameter types:

1. Boolean with only two possible values true and false (all lower or all upper case). The two boolean constants are also considered of Boolean type.

2. Ranges that are integer intervals defined by their lower bound l and upper bound u. With $[l..u]$ we denote all the integers between l and u included.

3. Enumerative that are a list of possible values between { }. We are rather liberal about the elements and we allow identifier starting also with a number, natural numbers, and strings. For example, one could define a enumerative values in this way:

```
values : {100, 1M, "my name", cit}
```

A simple example of combinatorial model with three parameters is shown in Fig. 31.

```
Model Phone
Parameters:
emailViewer : Boolean
textLines: [ 25 .. 30 ]
display : {16MC, 8MC, BW}
```

Figure 31: A smartphone example

There are some semantic rules about the parameters and their definitions, defined as follows:

1. The name of each parameter must be unique.

2. In Ranges, the lower bound l must be less than the upper bound u.

3. The elements in each Enumerative must be distinct. We allow two enumerative parameters to share some elements, though.

7.1.1 Constraints

A distinctive feature of our language is the support of modeling constraints among parameters. In most configurable systems, constraints or dependencies exist between parameters. Constraints may be introduced for several reasons, for example, to model inconsistencies between certain hardware components, limitations of the possible system configurations, or simply design choices [64]. In our approach, tests that do not satisfy the constraints

Expression	**Case**	**Correct iff**
$e_1\ op\ e_2$ with $op \in \{=, \neq\} \rightarrow boolean$ $e_2\ op\ e_1$ with $op \in \{=, \neq\} \rightarrow boolean$	e_1 enumerative, e_2 element	e_2 belongs to e_1 elements
	e_1 enumerative, e_2 enumerative	e_1 and e_2 share at least one element
	e_1 range, e_2 range	e_1 and e_2 share at least one number
	e_1 range, e_2 number	always
	e_1 number, e_2 number	always
	e_1 boolean, e_2 boolean	always
$e_1\ op\ e_2$ with $op \in \{<, \leq, >, \geq\} \rightarrow boolean$ $e_2\ op\ e_1$ with $op \in \{<, \leq, >, \geq\} \rightarrow boolean$	e_1 range, e_2 range	e_1 and e_2 share at least one number
	e_1 range, e_2 number	always
	e_1 number, e_2 number	always
$e_1\ op\ e_2$ with $op \in \{\wedge, \vee, \rightarrow\} \rightarrow boolean$ $e_2\ op\ e_1$ with $op \in \{\wedge, \vee, \rightarrow\} \rightarrow boolean$	e_1 boolean, e_2 boolean	always
$\neg e \rightarrow boolean$	e boolean	always
$e_1\ op\ e_2$ with $op \in \{+, -, *, /, \%\} \rightarrow number$	e_1 number, e_2 number	$e_2 \neq 0$ if $op = /$ or $op = \%$
	e_1 range, e_2 range	always
	e_1 range, e_2 number	$e_2 \neq 0$ if $op = /$ or $op = \%$
	e_1 number, e_2 range	always

Table 23: Rules of CTWEDGE Language Validator for Constraints

are considered *invalid* and do not need to be produced. For this reason, the presence of constraints may reduce the number of tests of the final test suite (but it may also increase it [64]). However, the generation of tests considering constraints is generally more challenging than the generation without them, and several test generation techniques still do not support constraints, at least not in a direct manner. In CTWEDGE we decided to focus more on techniques supporting constraints.

In CTWEDGE, we adopt the language of propositional logic (with the usual logical operators) with equality and arithmetic to express constraints. To be more precise, we use propositional calculus, enriched by the arithmetic over the integers and enumerative symbols. As operators, we admit the use of equality and inequality for any variable, the usual Boolean operators for Boolean terms, and the relational and arithmetic operators for numeric terms. To be more precise, Table 23 reports all the rules we have defined to check if a constraint is semantically correct.

For example, we can write constraints in this way:

```
Model Phone
..
Constraints:
# emailViewer => textLines > 28 #
# emailViewer and display != 16MC => textLines > 28 + 3#
```

Many test suite generation tools provide a limited support for constraints. For instance, AETG [62, 140] allows only simple constraints of type if then else or requires. The language of CTWEDGE is in this aspect more expressive, as it is targeted to be more general than existing tools. In the specific case of AETG, the translation of those templates into our logic is straightforward. For example the if then else constraint can be translated by two implications. Other tools [64] allow constraints only in the form of forbidden combinations [99]. Our language is more general, as a forbidden tuple would be translated as a *not* statement. For instance, a forbidden pair emailViewer = false; display = 16MC would be represented by the following constraint:

# not (emailViewer = false and display = 16MC) #

However, the explicit list of the forbidden combinations may explode and it may become impractical and error-prone to represent it. For example, if the model of mobile phones presented in Fig. 1 had a constraint that

"A front video camera requires also a 16MC display". This constraint would be translated into two forbidden tuples:

(emailViewer = true, display = 8MC);

(emailViewer = true, display = BW);

However, the translation as constraint in general form would be simply:

# emailViewer => display = 16MC #

which is more compact and more similar to the informal requirement.

In our language semantics, a test case is valid only if it does not contradict any constraint in the specification. Others [52] distinguish between combinations to be avoided *if possible* (soft constraints), and the forbidden combinations (hard constraints), which must always be avoided (our case).

Other tools, like CASA [59], support only constraints in conjunctive normal form, without arithmetic or relational operators.

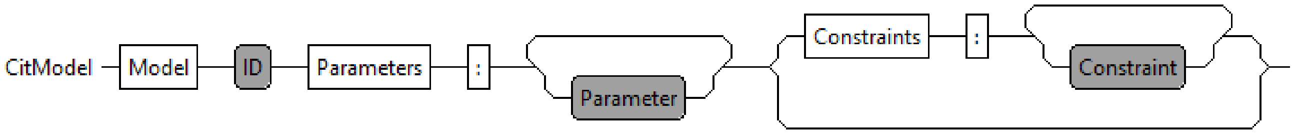


Figure 32: `CitModel` rule diagram

7.1.2 Xtext

There are countless ways to define a language together with its parser. One emerging technique for Domain Specific Language modeling is the use of Xtext [82]. By defining the grammar of the DSL of choice by means of a Xtext grammar, the language designer obtains a parser, APIs to programmatically access models, a serializer and a smart editor for it. The editor provides many features out-of-the-box, such as syntax highlighting, content-assist, folding, jump-to-declaration and reverse-reference lookup across multiple files. We had already used Xtext for defining the language in CitLab [90]. Xtext support also the generation of a web editor, as we present in the following section.

Grammar rules written in Xtext are very close to the standard (E)BNF production rules. For instance, the main grammar rule that defines the whole CIT model is defined as follows:

```
CitModel:
'Model' name=ID
'Parameters' ':' (parameters+=Parameter)+
('Constraints' ':' (constraints+=Constraint)+)?
```

Xtext can display the production rules by means of syntax diagrams, also known as railroad diagrams. For example, the rule presented before for `CitModel` is shown in Fig. 32.

Because Xtext is based on ANTLR, it does not allow left recursive parser rules and parsing nested expressions is not as simple as writing a EBNF rule. The ctwedge language parses the constraints by defining the precedence among operators implicitly by left-factoring expression definitions. For example, in order to parse the AND operators before the OR operators, ctwedge introduces the following two rules that are not left recursive:

```
OrExpression returns Expression:
AndExpression ({OrExpression.left=current}
OR_OPERATOR (right=AndExpression))*;

AndExpression returns Expression:
EqualExpression ({AndExpression.left=current}
AND_OPERATOR (right=EqualExpression))*;
```

The definitions of semantic constraints in Xtext is performed by user-defined validator classes written in Java or in Xtend containing methods annotated by `@Check`. For instance, to check that in the definition of any range domain of our CIT models the upper bound is greater than the lower bound, we have introduced the following checking method:

```
@Check
def checkRangeIsCorrect(Range range) {
if (range.getBegin() >= range.getEnd())
error("The second term must be greater ...");
}
```

7.2 CTWEDGE: CT Web Editor and Generator

In this section, we present our tool CTWEDGE. As we can see in Fig. 33, the tool is composed by a language definition component (with its Xtext parser and validator), a web-based editor for the CTWEDGE language, with some options for test generation and a test suite visualizer and exporter, and a test suite generator that exploits third-party test generation tools.

The tool is written in Java and Xtext, and can be deployed on any Web-application server, such as Apache Tomcat. It is publicly available at: `http://foselab.unibg.it/ctwedge/`.

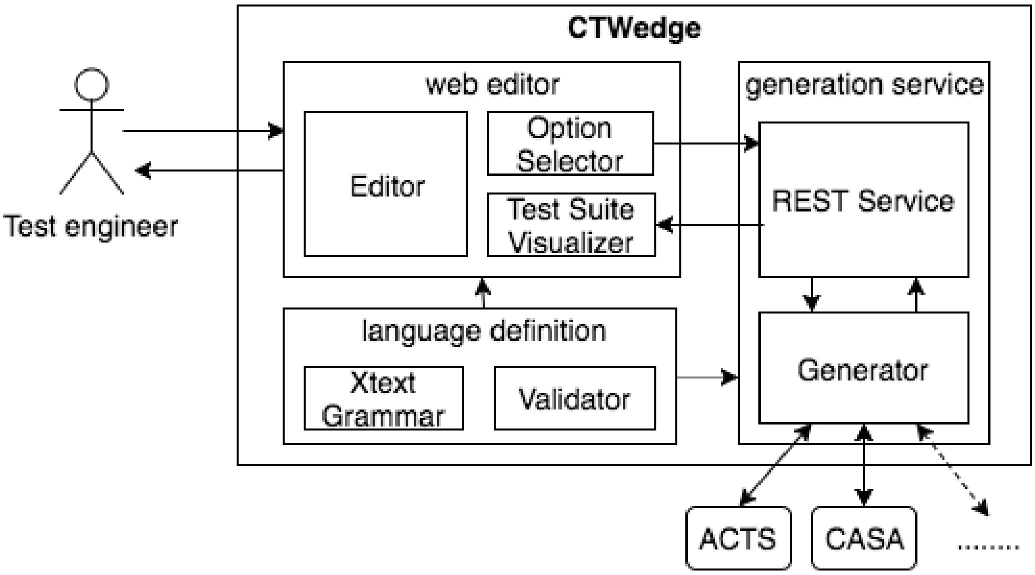


Figure 33: CTWEDGE **architecture**

7.2.1 Combinatorial Testing Web Editor

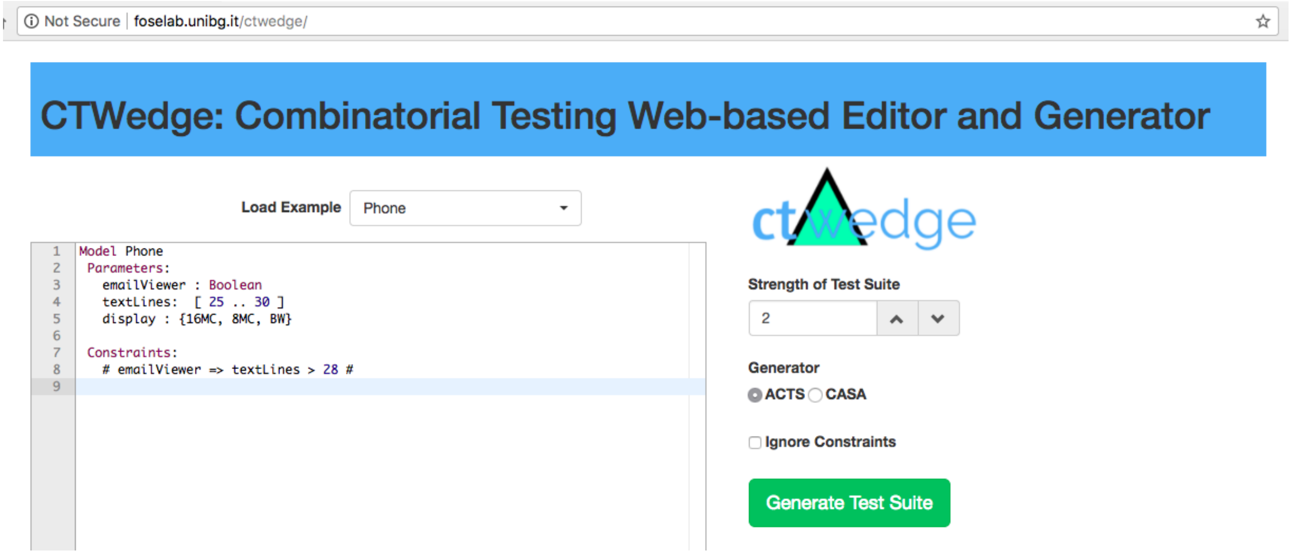
In order to implement a web-based editor, we can leverage the Xtext framework, since Xtext starting from version 2.9 offers an interface for integration of text editors in web applications. The text editors are implemented in JavaScript, and language-related services such as code completion are realized through HTTP requests to a server-side component.

The Xtext web-based editor provides several features, like content assist to help the user to complete the models, validation to check the correctness, syntax coloring, and formatting.

The CTWEDGE web editor is based on Ace (Ajax.org Cloud9 Editor)[2], but other web editors (like Orion and CodeMirror) are available. Xtext does not yet provide support for the recently standardized Language Server Protocol [1], which we plan to include in our tool as a future work. A screenshot of the CTWEDGE web editor is shown in Fig. 34. The web editor provides an immediate feedback while writing by means of syntax highlighting, auto completion, and errors markings.

The validation of the model is performed run-time while the user writes it. If the validator finds an error in the model, it generates an error message. The nature of the error is indicated in the pop-up box appearing when positioning the cursor over the error sign, and the point in which the error occurs is marked in the editor. Fig. 35 shows how model validation errors are displayed to the test engineer.

[2] See `https://ace.c9.io/`


Not Secure | foselab.unibg.it/ctwedge/
CTWedge: Combinatorial Testing Web-based Editor and Generator
ctwedge
Generate Test Suite

Load Example: Phone

```
Model Phone
 Parameters:
   emailViewer : Boolean
   textLines:  [ 25 .. 30 ]
   display : {16MC, 8MC, BW}

 Constraints:
   # emailViewer => textLines > 28 #
```

Strength of Test Suite: 2

Generator

● ACTS ○ CASA

□ Ignore Constraints

Figure 34: CTWEDGE web editor

The editor allows to load predefined examples of combinatorial models, selected from literature [180] and converted into CTWEDGE language format (with extension *.ctw*).

Graphical components (buttons and option selectors) are built using the JavaScript frameworks JQuery[3] and Bootstrap[4]. The web application is fully compatible also with mobile devices (Android and iOS), from any recent Web browser with JavaScript enabled.

7.2.2 Test generator web service

The function of actually generating a test suite from the test model and option parameter is web-served and performed by the test generation service which is the component of CTWEDGE. The test generation service is composed by two modules: a REST[5] service and a language translator.

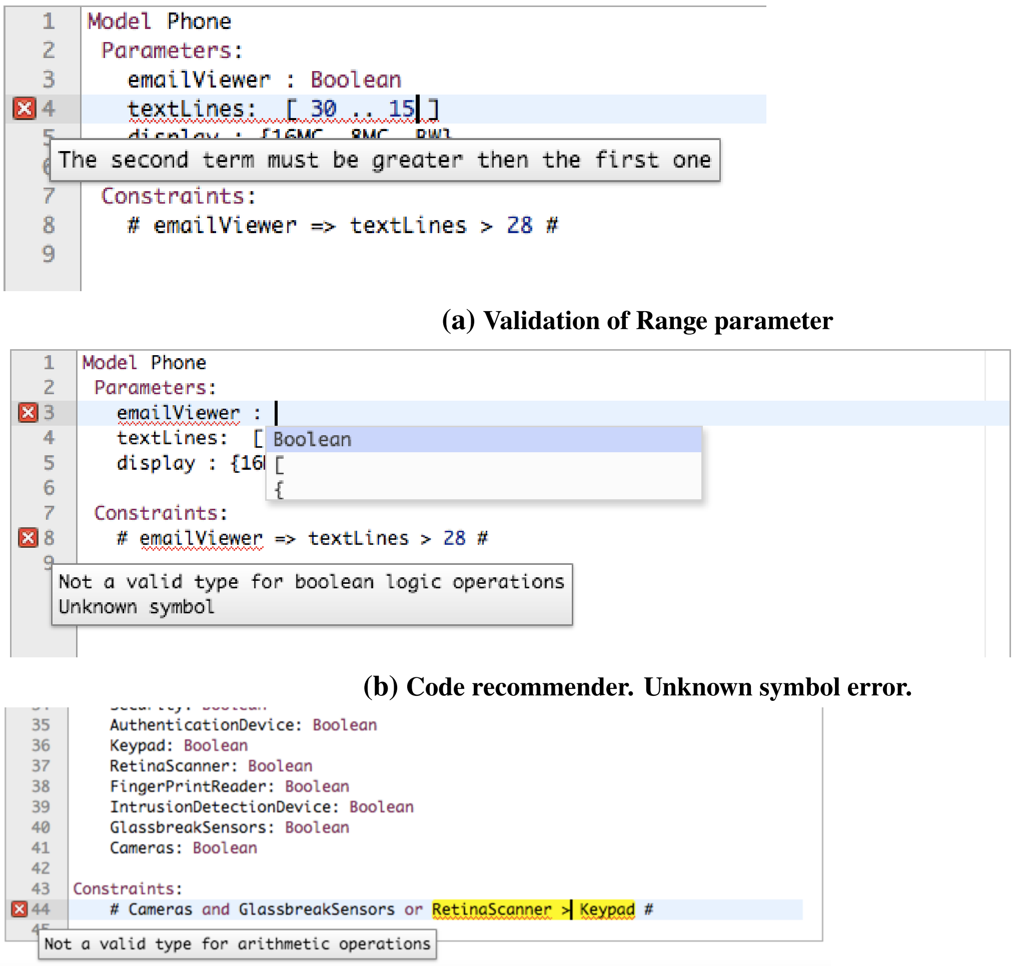
The REST service handles input model and option parameters for the generation, calls the generators from which it gets back the tests and it is responsible to deliver them to the web browser.

The test generator acts as a *driver* between CTWEDGE and the various third party combinatorial test generation tools, which are usually accessible via their own APIs, or via command line. The test generator exports the combinatorial model along with the generator options, into the the specific language of the external tool, or directly into the tool's

[3] JQuery: `https://jquery.com/`

[4] Bootstrap: `https://getbootstrap.com/`

[5] REST: Representational State Transfer



```
Model Phone
 Parameters:
   emailViewer : Boolean
   textLines:  [ 30 .. 15 ]
```

```
The second term must be greater then the first one
```

```
 Constraints:
   # emailViewer => textLines > 28 #
```

(a) Validation of Range parameter

```
Model Phone
 Parameters:
   emailViewer : 
   textLines:  [
   display : {16
 Constraints:
   # emailViewer => textLines > 28 #
```

```
Not a valid type for boolean logic operations
Unknown symbol
```

(b) Code recommender. Unknown symbol error.

```
    AuthenticationDevice: Boolean
    Keypad: Boolean
    RetinaScanner: Boolean
    FingerPrintReader: Boolean
    IntrusionDetectionDevice: Boolean
    GlassbreakSensors: Boolean
    Cameras: Boolean

Constraints:
    # Cameras and GlassbreakSensors or RetinaScanner > Keypad #
```

```
Not a valid type for arithmetic operations
```

(c) Validation of relational operations

Figure 35: Examples of CTWEDGE validation errors

APIs. Then, it waits for the generator to compute the test suite, and once ready, it passes it back to the REST service. If necessary, the generator also maps any parameter values back to the original CTWEDGE model format. Each external program needs its own translator.

The REST service accepts an HTTP GET or POST request with parameters as described in Tab. 24. For sake of brevity, an example URL request to the web generator service is shown in Fig. 36.

```
http://foselab.unibg.it/ctwedge.generator?model=Model%20Phone%20Parameter:%
↪ 20...>28%20#&strength=2&generator=acts&ignConstr=false
```

Figure 36: Generator URL example

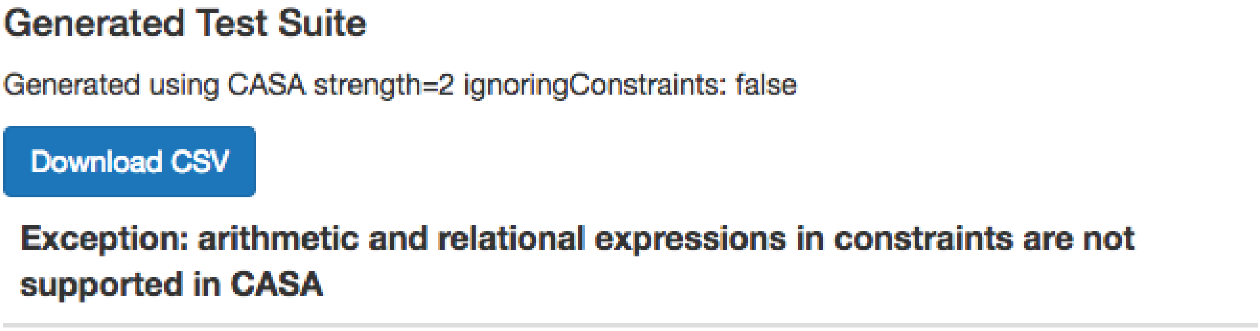
Table 24: Request parameters to CTWEDGE generation service

Parameter	type	description	values
model	String	the combinatorial model	as written and validated by the editor (see Sec. 7.1)
strength	Int	the combinatorial interaction strength	any integer above 1 (default is 2: pairwise)
generator	Enum	the tool to be used	["acts", "casa"] (so far)
ignConstr	Boolean	if constraints should be ignored in test generation	["true", "false"] (default is false)

So far, we interfaced CTWEDGE with ACTS [216] and CASA [59]. ACTS is accessed via it internal APIs whereas CASA is called via command line.

The resulting test suite is returned in CSV[6] format. The header line contains the parameter names, and each following line represents a single test, with parameter values. The web front-end allows to download CSV file to further local use, and shows the test suite in the browser by converting it into an HTML table. Conversion is straightforward and done client-side via Javascript.

Arithmetic operations and relational operators (>, <, ≤, ≥) are not natively supported by CASA, and in presence of such constraints, an error is reported to the user, as in Fig. 37.



Generated Test Suite

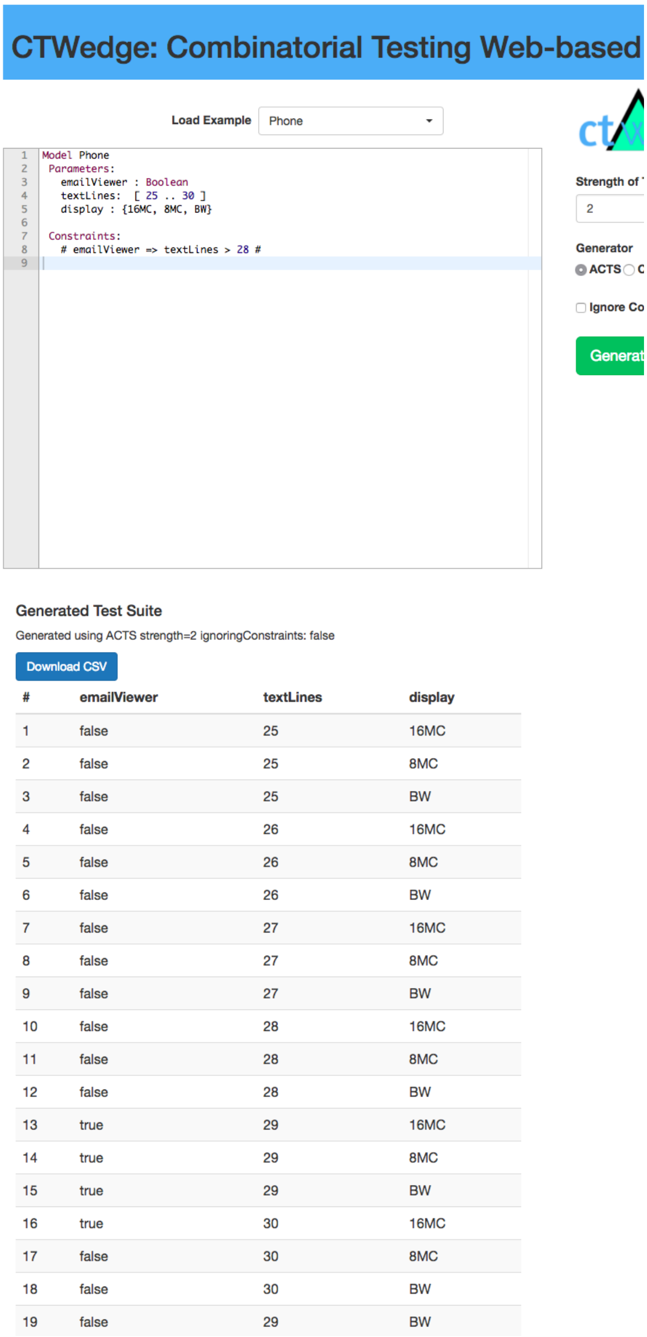
Generated using CASA strength=2 ignoringConstraints: false

Exception: arithmetic and relational expressions in constraints are not supported in CASA

Figure 37: Message of operations not supported in constraints for CASA generator tool

The HTML table showing the generated test cases is located below the editor on the same window. This location is less invasive than a brand new tab as it does not hide or replace the current combinatorial model in the editor. Despite it is not immediately visible to the

[6] CSV: Comma Separated Values



Load Example: Phone

```
Model Phone
 Parameters:
   emailViewer : Boolean
   textLines:  [ 25 .. 30 ]
   display : {16MC, 8MC, BW}

 Constraints:
   # emailViewer => textLines > 28 #
```

Strength of: 2

Generator: ● ACTS ○ C

□ Ignore Co

Generated Test Suite

Generated using ACTS strength=2 ignoringConstraints: false

#	emailViewer	textLines	display
1	false	25	16MC
2	false	25	8MC
3	false	25	BW
4	false	26	16MC
5	false	26	8MC
6	false	26	BW
7	false	27	16MC
8	false	27	8MC
9	false	27	BW
10	false	28	16MC
11	false	28	8MC
12	false	28	BW
13	true	29	16MC
14	true	29	8MC
15	true	29	BW
16	true	30	16MC
17	false	30	8MC
18	false	30	BW
19	false	29	BW

Figure 38: CTWEDGE visualization of the generated test suite

user, who may think the output is hidden, we believe that it preserving the access to the current screen is the most important aspect to be preserved.

The generator is called by the editor by AJAX, with an asynchronous XmlHttpRequest.

A screenshot of how the generated test suite is presented to the user is shown in Fig. 38. It is shown as plain HTML table, with the possibility to download the data as CSV.

7.3 Related Work

With the success of the SaaS pattern, web-based tools have rapidly gained popularity due to their portability and ease of use. There exist some web-based services also for combinatorial test case generation, each with its own peculiarities. SaaS tools, however, still represents a small number of all the available tools for test case generation: IDE plugins and desktop applications represent almost the totality of the current tools. We looked for tools from the pairwise.org[7] and softwaretesters.net[8] tool catalogs, and from the web, to the best of our searching skills.

Tool	URL	Documentation
TestCover	`https://testcover.com`	`https://testcover.com/sub/instructions.php` (visible after registration)
CTWebClassic	`http://alarcostest.esi.uclm.es/CombTestWeb/combinatorial.jsp`	`http://alarcostest.esi.uclm.es/CombTestWeb/stuff/usersManual.pdf`
CTWebPlus	`http://www.testcasegeneration.com` or `http://www.ctwebplus.com/`	`http://www.ctwebplus.com/stuff/userManual.pdf`
HexaWise	`https://hexawise.com/`	`https://hexawise.com/Hexawise_Introduction.pdf`
PairWiser	`https://inductive.no/pairwiser/`	`https://inductive.no/pairwiser/knowledge-base/`

Table 25: Tool resource links

We compare the following five SaaS for CIT, listed in Tab. 25:

- TestCover [197] is a commercial web-based combinatorial test case generator supported by Testcover.com, LLC, founded in 2003. The tool was also presented at IWCT 2016 [182].
- CTWeb Classic [203] is a free online tool for combinatorial testing and state machine test case generation, developed at University of Castilla-La Mancha (Spain).
- CTWeb Plus [5], an academic combinatorial test generation tool developed as improvement of CTWeb Classic. CTWeb Plus is now commercially supported.

[7] See `http://www.pairwise.org/tools.asp`

[8] See `https://softwaretesters.net/zbxe/index.php?mid=downloadtool&category=4258006&sort_index=readed_count&order_type=desc`

- HexaWise [2], a commercial combinatorial test case editor and generator, launched in 2009 by Hexawise, Inc.

- PairWiser [3], a commercial web-based tool provided by Inductive AS. The online version was shut down January 15th, 2018. After that date, only the standalone application, for own-server installation, is available.

All tools are well documented, with examples and tutorials. Links of on-line editors and official documentation resources of these tools are shown in Tab. 25

	TestCover [197]	**CTWeb Classic** [203]	**CTWeb Plus** [5]	**HexaWise** [2]	**PairWiser** [3]	**CTWEDGE**
Language						
Parameter Definition	Enumerative	Enumerative	Enumerative	Enumerative, Ranges (via value expansion)	Enumerative	Boolean, Enumerative, Ranges
Constraints format	in DPB notation: via *blocks* (i.e., sets of allowed combinations)	as *if-then-else*	AND, OR, Else operators, not nested	invalid pairs (*if..then..*)	guided by select boxes with rich choice of operators	arbitrary formula
Numeric operators	✓(in PHP functions)	✗	✓	✓	✗	✓
State Machine support	✓	✓	✓	✗	✗	✗
Editing						
Web-based editor	text area	text fields and buttons + file upload	text fields, buttons, drawing area	text fields and buttons	text fields and buttons	text area
Model Import/Export	✓(Copy&Paste as text)	✓	✗	✗	✗	✓(Copy&Paste as text)
Helping facilities	✗	button-guided (no facilities to build input file to upload)	button-guided	button-guided	button-guided	content-assist, syntax highlight, in-line error reporting
Example Models	✓	✓	to be rebuilt from documentation file, not one-click loadable	✓	in the documentation	✓
Generation						
n-wise	pairwise	pairwise	pairwise	up to 6-way interaction + mixed strength	up to 3-way interaction + mixed strength	✓
Supported generators	All-pairs	AETG, PROW, All combinations, Each choice, Random, Bacteriologic	AETG, Pairwise, All-combinations, Each-choice, Comb, Random	not specified	not specified	ACTS, CASA
Export formats	HTML, WSDL interface	HTML, CSV	HTML	HTML, Excel, CSV, OPML	Excel, Jira issues	CSV, HTML
Generate test scripts	✓(for Selenium)	✓(custom)	✓(custom)	✗	✓(custom)	✗
Coverage visualization	✓	✗	✗	✓	✓	✗
Other information						
License	Commercial	Free	Commercial	Commercial	Commercial	Free
Registration	subscription required	optional	subscription required	subscription required	own-server installation	✗
Online storage	✓	✗	✓	✓	✓(on own server)	✗
Additional notes	Functions in PHP into constraints. Also accessible via WSDL interface.	Registration required for models with more than 5 parameters	Features a drawing area to represent states and transition of a state machine. The combinatorial model must be in the form of a state machine.	Has also a chart showing the interaction coverage after each test	Pairwiser online was shut down January 15, 2018. Available only for own-server installation. Allows to specify combinations to include in test suite.	–

Table 26: A comparison with other SaaS for CT

To compare the SaaS tools among them and w.r.t. CTWEDGE, we consider the following aspects, that we believe to be among the most relevant for a test engineer interested in using a web-based combinatorial test generation tool:

- **Language**. We look into the expressiveness of the accepted format for the combinatorial model in input. This evaluation includes:
  - Parameter definition: how the parameter types and values can be defined. For instance, a tool may support Boolean parameters or ranges of integers to express an enumerative made of all integers between two numbers.
  - Constraint format: if the constraints can be expressed as free combinations of logical and arithmetic operations among parameters, or have special formats, such as a set of forbidden tuples, a set of implications, or a set of if-then-else conditions.
  - Numeric Operations: if constant numbers and basic numeric operations (+, -, *, /) are allowed in the constraints and/or in the generated code of test cases.
  - State Machine support: if the language supports an easy input of state machines, to generate combinatorial tests for their execution.
- **Editing**. We evaluate how simple is for the test-engineer to input the combinatorial model into the tool. This category includes the following aspects:
  - Web-based editor structure: how is the GUI of the web editor for writing the combinatorial model to be given in input to the tool; for example, if it is made by a single text area, or some buttons and text fields. Some tools use a single text area for the whole model, whereas some other tools use text inputs for individual parameters, reducing the need for parsing and text-highlighting.
  - Import and export models: how the models can be exported to the file system and imported. For example, a tool may allow importing a text file written with another editor.
  - Helping facilities: how the test engineer is guided in the input of the model in the web editor; for example with syntax highlighting, content assist, inline error reporting, warning messages, or single text input fields to fill, and self-explanatory buttons to click.
  - Predefined example models: if there are examples of combinatorial models that can be easily loaded into the tool and executed to generate a test suite.

- **Generation**. We evaluate how the test suite generation is performed and how the output is presented to the user. This category includes the following aspects:
  - n-wise: which interaction strengths of the generated test suite are allowed
  - supported generators: which existing combinatorial test generation algorithms are supported
  - export formats: in which formats the output is made available to the test-engineer
  - Test-script generation: if there is a mechanism to allow the generated test vectors to be directly inserted into test cases written in custom code.
  - Coverage visualization: if there is indication (textual or with charts) of the coverage reached after the execution of each test in the generated test suite.
- **Other information**. We consider aspects about the accessibility of the tool, and related features. We look the following aspects
  - License: if commercial, free, or open source.
  - Registration: if it is mandatory, optional or not made available.
  - Online storage: if any data (input models, or output test suites) can be stored online.
  - Additional notes: any other additional information that we consider worth being noted.

Table 26 compares the five tools and CTWEDGE according to all these aspects.

Concerning the web editor, while TestCover uses, as CTWEDGE, a single text area for combinatorial model input, all the others (CTWeb Classic, CTWeb Plus, HexaWise and PairWiser) feature a composer of combinatorial model guided by multiple selectors, text fields, and buttons. This approach of using buttons and text input fields, has the advantage of a quicker learning curve, and it does not need a language grammar, nor a parser, nor the helping facilities typical of text-based editors, such as auto-completion and syntax highlighting. However, it is not always the preferable way to input combinatorial models in the tool. In fact, the availability of a domain-specific language makes it possible to quickly and easily write, edit and copy-paste combinatorial input models, and export, translate or port them to other platforms and tools.

CTWeb Classic comes both with a guided editor and a form to upload a text file containing the input of the tool, written in a domain specific language. Regarding the textual way of proving input for test case generation, however, although CTWeb Classic and TestCover have a good documentation, they have no facilities to help the test engineer in writing models. CTWeb Classic does not have an online editor for its own language (as it comes with just a file upload button), and TestCover has a simple text area, lacking support for auto-completion, syntax-highlighting and all the features proper of an IDE.

All the tried tools offer support for test case generation with constraints, to be specified in their specific formats. TestCover even allows to specify custom functions - in PHP code - to express constraints [182].

TestCover and CTWeb (Classic and Plus) offer pairwise test case generation, that is very often the chosen interaction strength by test-engineers. For some applications, however, higher interaction strength is preferred. HexaWise supports up to 6-way interaction strength, while PairWiser up to 3-way. CTWEDGE is, instead, the only tool that does not pose limitation (in theory) on the interaction strength of the test suite. However, HexaWise and PairWiser come with the additional possibility to specify a mixed test suite strength, i.e., values of each single parameter may be covered with different strength.

Still none of the tools supports the Microsoft Language Server Protocol [1], a new common open protocol for language servers which provides programming language-specific features to source code editors or integrated development environments (IDEs). The main goal of the standard is to support programming in any given language independently of editors or IDEs. We plan to extend CTWEDGE in order to support LSP.

7.4 Future Work

There are several directions in which we plan to work.

Language extensions Adding expressive power to the language for combinatorial models, in particular to express test seeds and goals, represents a direction for future work. Test seeds allow a tester to force the inclusion of certain test cases in the generated test suite [52]. Test seeds may be complete or partial. Test goals are extra-constraints: relations among parameters to be satisfied by at least one test in the generated test suites.

Some CIT approaches [180] introduce weights for parameter values. Weights reflect the importance of different values for a given parameter. The user can express further requirements over the solution involving weights. Even if the same constraints may be

> expressed in our language, it may become impractical. We plan to extend the CTWEDGE language in order to include user defined functions depending on parameter values. A possible function could be the weight of a parameter. Constraints and test goals could use such functions to express complex testing requirements.
>
> **Combinatorial model editor** To make the transition to CTWEDGE easier, a possible direction for future improvement is an importer that translates models written in other generator formats, into CTWEDGE language format.
>
> Secondly, although CTWEDGE already follows the SaaS approach, there are still several features that could be added in order to offer new cloud-based services. The web site could offer a storage and persistence service, and the logged user could save his/her models on the CTWEDGE server and later recall the saved files and export/import them in other formats. The CTWEDGE could offer analysis services, like those presented in [31], able to find modeling faults. The user could use such techniques to check that the constraints are consistent, that there is no constraint implied by other constraints, and that the parameters and their values are really necessary. Also coverage measurement and analysis on the generated tests could be useful in order to check that they actually cover all the testing requirements.
>
> Another future direction is the visualization of the individual t-tuples, as covered by each test in the test suite.

Another direction regards the way tests are generated. CTWEDGE produces the tests in a synchronous mode: when the user calls the generator, the service starts producing the tests. This approach is not feasible in case there are many requests or the models become big. We plan to move to an asynchronous approach, in which the user requests the test generation, the server queues all the requests and make available on the server the test suites when they are generated.

Test case generation

> *[89]* Another direction for future work regards test case generation. The server is configured to run CTWEDGE generator in a synchronous mode: the generator starts producing the test suite immediately, trying to serve all the requests. The server could have performance issues due to overloading in case for example there are many requests with large models. The web service could be improved by attaching a process scheduler and a load balancer.

Test suite generation becomes therefore asynchronous also on the server, which queues the requests and makes the test suites available as soon as they are generated.

External generation tool support An additional feature direction consists in the expansion of the support for test case generators, as PROW [131], PICT [4], HSST[9] [157], Medici [91], as well as an expansion on the customizations of each selected test generator tool, such as the selection of the test generation algorithm inside ACTS: IPOG [133], IPOG-D [133], IPOG-F [85], IPOG-F2 [85] and PaintBall [17]. The possibility to download the translated input file along with the executable command parameters for each of the generators allows further customization and therefore can be an interesting future extension of the tool.

Offline extensions Combinatorial test case generation is used as a part of an automated process for application testing. Thus running the test generation tool off-line is needed, in certain scenarios, to ease interfacing with automated tools or with IDEs during development process. We therefore plan to release a version of the CTWEDGE editor as Eclipse plugin. Xtext, already generates an eclipse-based development environment providing editing experience known from modern IDEs, featuring a content assist, quick fixes, a project wizard, template proposal, outline view, hyperlinking, and syntax coloring.

7.5 Conclusions

Generation of combinatorial test suites via web offers great advantages w.r.t. classical desktop applications. It is nowadays supported by a pool of tools, both open source and commercial. However, to the best of our knowledge, none of them has an integrated web editor support and a complete support of constraints. To work, they have their own language, with often just examples as unique description, and expect the user to write a file locally before uploading to the web-based tool. This process requires the test engineer to use another tool, may it be just a stand-alone text editor, with no auto-completion for that particular language, or a custom stand-alone editor with some grade of code recommendation.

To offer a complete SaaS environment for CIT, we have developed and deployed CTWEDGE. CTWEDGE was designed with three principles in mind: (1) installation-free and download-

[9] HSST: Heuristic based on solution space tree

free, (2) ease of use, and (3) extensibility to support more generators. By using Xtext, we have defined a simple textual language which includes also the possibility to define complex constraints. Thanks to Xtext, a web editor can be easily deployed and it offers classical editing features like syntax highlighting and coloring, syntax validation, auto completion, and error messages. We have also developed a REST service that is able to generate CIT test suites exploiting third-party test generator programs. This test generators runs on the server and it can be called from the editor, thus providing a complete SaaS experience to the tester.

Chapter 8. MixTgTe: Efficient and Guaranteed Detection of t-Way Failure-Inducing Combinations

This chapter presents a tool to support our model repair process by improving the fault localization effectiveness, namely the failure-inducing combination detection.

By testing all the interactions till a given strength t, we can validate the system or discover if it contains parameter combinations that cause failure. An interaction of size t (or t-way interaction) is an assignment of a specific value to each of the selected t parameters. Although the size of combinations causing faults is almost always unknown, experiments show that generally even a low degree of interaction is enough to discover faults [129]. One of the main goal of CIT research is to find techniques that are able to cover all the interactions of a given strength with as few tests as possible. In this way, the faults can be found by running only a possibly small number of tests.

While test generation for CIT is a well-studied topic, fault detection and localization is still an open problem, although there are now some works targeting diagnosis and bug characterization [114]. When a failing test has been found, it remains unclear which combination in the failing test is responsible for the failure, since a test contains many combinations of different sizes. Knowing the interaction (and, therefore, also all the input configurations) that trigger failures is of help not only in correcting bugs, but also in understanding the impact of them. The particular interacting configuration that induces a failure, often directly reflects a use case scenario, which can be traced back from the input configuration. Knowing only the test that causes a fault, instead, often makes it impossible to trace back to the general use-case scenario (maybe involving several other input configurations) that caused the fault.

The problem is how to locate the combination that causes a failure when a fault is discovered [158, 159]. Indeed, in case of failure, there is a *masking* effect among the interactions [158] that makes hard the precise localization of the right combination. In order to avoid this masking effect, new tests are needed besides those necessary to cover all the interactions as required by CIT. Moreover, the test suite size optimization can play against fault localization: having each test to cover as many interactions as possible reduces the size of the test suite, but it may make the fault localization harder.

Classically, test generation and fault localization are done *sequentially*. First, a combinatorial test suite is generated and then executed against the real system. Then, if a fault has been found, new tests are built to try to locate the faulty combinations. This process is not very efficient (no information about failures is used during generation) and it generally does not guarantee to discover all the faulty combinations. Lately, there have been some approaches that *interleave* test generation, test execution, and fault localization [159]. Our approach follows this new trend and tries to efficiently build test suites taking into account possible test failures during test execution.

Most works do not guarantee to detect the real failure inducing combinations. Most approaches show that they are able to identify very suspicious combinations that are likely to be failure inducing [96], but no guarantee is given. However, under some precise assumptions, also testing can locate bugs. For instance, if one knows the maximum number and maximum strength of failure inducing interactions in advance [65], also particular combinatorial test suites statically generated (called *locating arrays*) can be used to identify those interactions. Our approach follows this trend as well: under some rather general assumptions, we are able to (dynamically) generate test suites that guarantee the detection of failure-inducing combinations.

The contribution is therefore twofold:

1. detect and identify *all* the failure-inducing interactions in a system, up to a certain size t, if they exist;
2. doing it with a *smaller* test suite (compared to other techniques) by exploiting information coming during test execution about possible failure of tests.

During test generation, our approach collects tuples that seem to cause failures and tries to *isolate* them by finding those tuples that instead are *passing*. By using this information, tests are generated and immediately executed, and the knowledge base of the system updated accordingly.

The chapter reflects the content of the paper [28], and it is structured as follows. Sect. 8.1 introduces the necessary background, Sect. 8.2 explains the definitions we need in our approach, Sect. 8.3 presents the iterative process we propose that combines test generation, test execution, and identification of failure inducing combinations. Sect. 8.4 discusses about the assumptions of the process and introduces some theorems about its capabilities. Finally, Sect. 8.5 describes some experiments we performed to evaluate the process, Sect. 8.6 reviews some related work, and Sect. 8.7 concludes the chapter.

8.1 Background

[28] We assume that the software under test (SUT) has m input parameters that are described by a combinatorial model defined as follows.

Definition 8.1 (Combinatorial Model). *Let $P = \{p_1, \ldots, p_m\}$ be the set of parameters. Every parameter p_i assumes values in the domain $D_i = \{v^i_1, \ldots, v^i_{o_i}\}$.*

```
Model totinfo

Parameters:
tables: {t0, t1, t2}
row: {r0, r1, r2}
column: {c0, c1, c2}
table_attribute: {ta0, ta1, ta2, ta3, ta4, ta5}
input_attribute: {i0, i1, i2, i3, i4}
maxline: {m1, m2, m3, m4, m5}
```

Code 1: A combinatorial model of the input of *totinfo* program, in CTWedge

In order to model and manipulate combinatorial models, we use the tool CTWedge [89]. Note that a combinatorial model may also contain constraints that, however, are not considered in this work.

Example 8.1. *Let's consider the combinatorial model (originally proposed by Ghandehari et al. [98]) of the* totinfo *program from the Siemens Suite in the Software Infrastructure Repository (SIR) [72]; the model has m=6 parameters, namely P={tables, row, column, table_attribute, input_attribute, maxline}, having enumerative values. Code 1 shows the representation of such model in CTWedge. In the following, for presentation purposes, we consider as running example a simpler combinatorial model M having 3 boolean parameters* $P=\{A, B, C\}$.

Definition 8.2 (Test case). *Given a combinatorial model M, a test case is an assignment of values to every parameter* $\{p_1, \ldots, p_m\}$ *of* M. *Formally, a test* f *is an m-tuple* $f = (p_1{=}v_1,\ p_2{=}v_2,\ \ldots,\ p_m{=}v_m)$, *where* $v_i \in D_i$, *for* $i \in \{1, \ldots, m\}$. *We identify with* $f(p_i)$ *the value* v_i *of parameter* p_i *in test* f. *We use the function* $result(f)$ *to indicate whether a test in a test suite TS passes or fails on a system SUT:*

$$result : TS \rightarrow \{pass, fail\}$$

i.e., $result(f)$ *is fail if and only if the test* f *fails, for example, because the SUT executed with* f *produces a wrong value or because an error or an exception occurs;* $result(f)$ *is pass otherwise.*

Note that other approaches [158] assume that different tests can fail in a different way, i.e., they can be distinguished by exception traces, state conditions, etc. In our setting, all the failing tests fail *in the same way*.

Example 8.2. *Given the model M in Ex. 8.1, a possible test case is f=(A=0, B=1, C=0). When a parameter is boolean, it can be denoted just with its name if its value is true (1), and with a bar above its name if its value is false (0). The example then becomes f=$\bar{A}B\bar{C}$.*

Definition 8.3 (Combination). *A combination (or partial test, or tuple or schema) c is an assignment to a subset $Dom(c)$ of all the possible parameters P, formally $Dom(c) \subseteq P$. A test is thus a particular combination in which $Dom(c) = P$. We identify with C_t the set of all the combinations of size t for a given set of parameters P.*

Example 8.3. *For the model M introduced in Ex. 8.1, a possible combination is c=$\bar{A}B$.*

Definition 8.4 (Combination Containment). *A combination c_1 contains a combination c_2 if all the parameters of c_2 are also parameters of c_1, and their values are the same. Formally: $Dom(c_2) \subseteq Dom(c_1) \wedge (\forall p_i \in Dom(c_2) \colon c_1(p_i) = c_2(p_i))$.*

Example 8.4. *For instance, for the running example M, the test f=$\bar{A}B\bar{C}$ contains the combination c=$\bar{A}B$.*

Definition 8.5 (Test Suite). *A test suite TS is a set of test cases. We denote with ETS the Exhaustive Test Suite that contains all the possible tests that can be formed from the specified combinatorial model; with CTS_t, instead, we identify a Combinatorial Test Suite of strength (at least) t, i.e., a test suite in which all the possible t-way interactions are covered by at least one test.*

Definition 8.6 (Failure-inducing combination). *A combination c is a failure-inducing combination (fic) for a test suite TS if each test that* contains *c fails. Formally, $\forall f \in TS \colon c \subseteq f \rightarrow result(f)=fail$. We identify with $isFic(c, TS)$ the predicate that tells whether the combination c is a failure inducing combination for a certain test suite TS.*

We call c a true-failure-inducing combination if we consider all the tests in the exhaustive test suite ETS, i.e., if $isFic(c, ETS)$ holds. We call c a t-failure-inducing combination (fic_t), if we consider all the tests in a CTS_t, i.e., if $isFic(c, CTS_t)$ holds.

Observation 1. From Def. 8.6, we observe that a combination c is guaranteed not to be failure-inducing if there exists a test that contains it and does not fail.

Example 8.5. *Let's consider the model M introduced in Ex. 8.1 and the test suite shown in Table 27a.*

(a) *ETS*

test	A	B	C	*result*
1	0	0	0	pass
2	0	0	1	pass
3	0	1	0	fail
4	0	1	1	pass
5	1	0	0	fail
6	1	0	1	fail
7	1	1	0	fail
8	1	1	1	fail

(b) *CTS_2*

test	A	B	C	*result*
7	1	1	0	fail
6	1	0	1	fail
4	0	1	1	pass
1	0	0	0	pass

Table 27: Test suites for running example

By definition, all the failing tests (# 3, 5, 6, 7, and 8) are failure-inducing combinations. In addition, we can notice that also the 2-way combinations AB, $A\bar{B}$, AC, $A\bar{C}$, and $B\bar{C}$ are failure-inducing combinations, since all the tests containing them fail. Moreover, also the 1-way combination A is failure-inducing. In this example, the test suite is an exhaustive test suite, therefore these combinations are also true-failure-inducing combinations.

Definition 8.7 (Minimal failure-inducing combination). *A failure-inducing combination c is minimal (*mfic*) if and only if all the combinations in c (except c itself) are not failure inducing in the test suite TS. Formally, $isMfic(c, TS)$ if and only if $isFic(c, TS) \wedge (\forall c' \subset c\colon \neg isFic(c', TS))$. If we consider a combinatorial test suite CTS_t, we call c a t-minimal-failure-inducing combination (mfic_t).*

Example 8.6. *In the test suite shown in Table 27a, the combinations $c_1{=}A$ and $c_2{=}B\bar{C}$ are both minimal.*

8.2 Definitions

First we want to define when a failure-inducing combination has been *located* and *isolated* by a suitable test suite *TS*.

Definition 8.8 (Isolated mfic). *An mfic c is isolated by a test f of a test suite TS if and only if c is the only fic in f, i.e.,*

$$\begin{aligned}&\mathit{isIsoMfic}(c, f, TS) \equiv \\ &\mathit{isMfic}(c, TS) \land c \subseteq f \land (\forall (c' \neq c) \subset f \colon \neg \mathit{isMfic}(c', TS))\end{aligned}$$

We say that a test suite TS isolates an mfic c iff

$$\mathit{isIsoMfic}(c, TS) \equiv (\exists f \in TS \colon \mathit{isIsoMfic}(c, f, TS))$$

Note that being able to isolate an mfic is particularly important for fault localization (that should follow our process); indeed, if two or more mfics are present in each failing test, it is more difficult to localize the fault as the mfics mask each other [158]. However, it is not always possible to isolate mfics. Consider, for example, a SUT with boolean parameters $\{A, B, C\}$, whose true-mfics are AB, AC, and $B\bar{C}$: AB cannot be isolated in this case.

Theorem 8.1. *If the SUT has a* true-*mfic of strength t, in any combinatorial test suite CTS_t there exists a failing test case.*

Proof. By definition of combinatorial test suite. □

A consequence of the theorem is the next corollary.

Corollary 1. *If there is no failing test in a combinatorial test suite CTS_t, then there is no true-mfic of size t.*

However, this property is not sufficient to isolate mfics of size t, as stated by the following theorem.

Theorem 8.2 (Insufficient accuracy of CTS_t)**.** *A CTS_t does not guarantee to isolate mfics of size t.*

Proof. Consider the running example whose true-mfics are A and $B\bar{C}$. The combinatorial test suite of strength t=2 shown in Table 27b only has $AB\bar{C}$ and $A\bar{B}C$ as failing tests. The detected mfics are A, $B\bar{C}$, and $\bar{B}C$. We would need at least one more passing test containing $\bar{B}C$ to correctly classify it (test #2 in Table 27a). Moreover, in order to isolate $B\bar{C}$, we would need one more test in which it fails alone (tests #3 in Table 27a). □

8.3 The MixTgTe method

Finding true-mfics can only be obtained using the exhaustive test suite *ETS*. However, exhaustive testing is in general not possible; therefore, we propose the approach MixTgTe (Mix Test Generation and Test Execution) that tries to identify and isolate mfics up to a given strength t. In order to do this, it uses combinatorial test suites CTS_t.

MixTgTe is an iterative process, as shown in Fig. 39 and Alg. 3. It starts from identifying combinations of size t=1 using the procedure MixTgTe$_t$, and progressively repeats the search algorithm to combinations with higher size, until the user (who, at every iteration, can inspect the set *ISOMFICS* of discovered isolated mfics of size less or equal to t) decides to stop the process, or t reaches the number of parameters $|P|$ of the system under test. The latter condition, however, is equivalent to exercising the exhaustive test suite, and it is normally infeasible in practice, except for trivial systems.

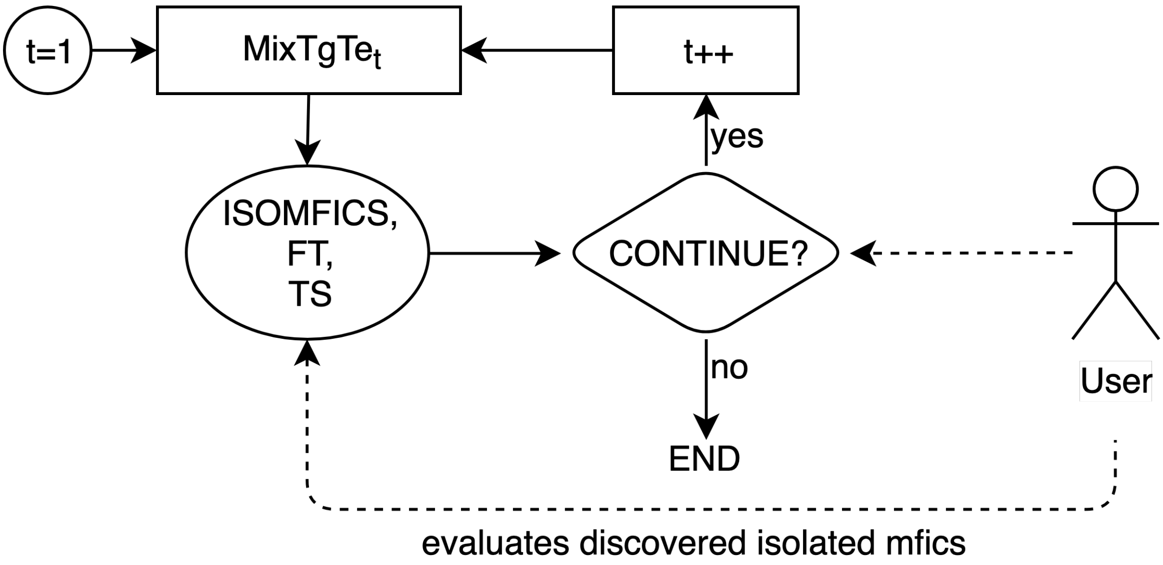


Figure 39: Overview of the user-driven iterations of the process alternating test generation and detection of isolated mfics

Algorithm 3 MixTgTe

```
ISOMFICS ← ∅
FT ← ∅
TS ← ∅
t ← 1
while t ≤ |P| ∧ User decides to continue do
  MixTgTe_t(t, ISOMFICS, TS)
  t ← t + 1
end while
```

At each step, to keep limited the number of tests to execute on the SUT, the *minimal* strength of the test suite used is equal to the size t of the detected combinations. Indeed, by Thm. 8.1, we can observe that this guarantees to have in the test suite all the mfics of size t. However, it could be some mfics are not isolated; therefore, at each iteration, we also generate additional tests to isolate all the discovered mfics.

At each step, the user checks the returned sets of *ISOMFICS* to determine if it is the case to continue to search for mfics of higher strength. The choice to continue or not is based on the available budget, but may also depend on the returned mfics in *ISOMFICS* and the test suite *TS*.

8.3.1 MixTgTe$_t$

Fig. 40 depicts the procedure MixTgTe$_t$ that, given a certain combination size t, and a set of previously executed tests, produces a combinatorial test suite of strength t able to detect and isolate mfics of strength up to t. The process is described in detail in Alg. 4

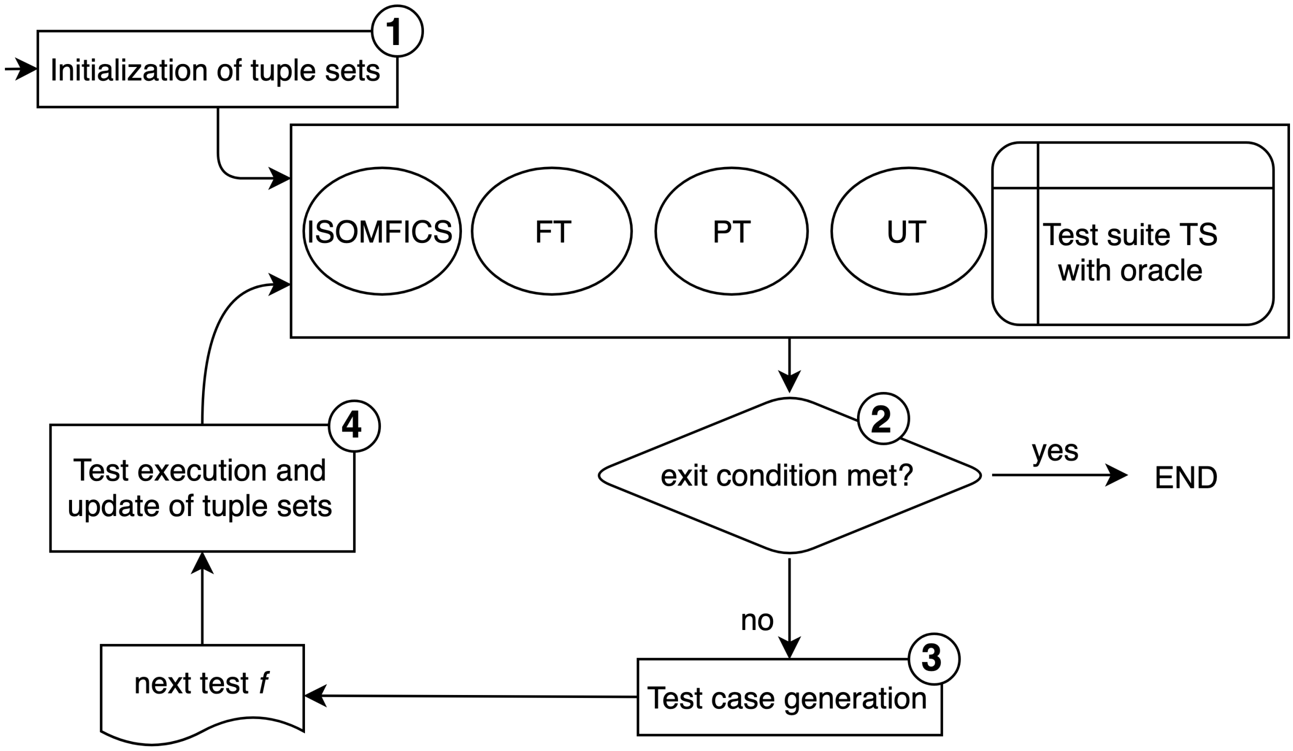

1
Initialization of tuple sets
ISOMFICS
FT
PT
UT
Test suite TS with oracle
2
exit condition met?
yes
END
no
3
Test case generation
next test f
4
Test execution and update of tuple sets

Figure 40: MixTgTe$_t$ process to find and isolate mfics up to accuracy of strength t

and in the rest of the section.

MixTgTe$_t$ works on the following sets of combinations:

- *UT* (Unknown Tuples): the combinations of size t not appeared yet in any test during the process;

Algorithm 4 MixTgTe$_t$

```
Require: $t$: strength
Require: $ISOMFICS$: isolated mfics computed at step $t$-1 (empty if $t$=1)
Require: $FT$: failing tuples found at step $t$-1 (empty if $t$=1)
Require: $TS$: test suite computed at step $t$-1 (empty if $t$=1)
1: $PT \leftarrow \{c \subseteq f | f \in TS \wedge result(f) = pass \wedge c \in C_t\}$
2: $FT \leftarrow FT \cup \{c \subseteq f | f \in TS \wedge result(f) = fail \wedge c \in C_t\} \setminus PT$
3: $UT \leftarrow C_t \setminus (PT \cup FT)$
4: while $\neg(UT = \emptyset \wedge (FT = \emptyset \vee (\forall c \in FT : isExplained(c, TS))))$ do
5:   $f \leftarrow buildTest(UT, FT)$
6:   $TS \leftarrow TS \cup \{f\}$
7:   updateTupleSets($f, PT, FT, UT, ISOMFICS$)
8:   updateMFICS($TS, FT, ISOMFICS$)
9: end while
```

- *PT* (Passing Tuples): the combinations of size $\leq t$ that were contained in at least one passing test executed so far in the process;
- *FT* (Failing Tuples): the combinations of size $\leq t$ that were contained only by failing tests, among all the tests executed so far in the process, excluding the isolated mfics.
- *ISOMFICS*: the set of isolated mfics detected so far (of size $\leq t$).

In addition, the process keeps track of the set of tests already run in the test suite *TS*, together with the value of their *result* (either pass or fail).

We give a further definition that is used in the process.

Definition 8.9 (Explained fic). *Given the set ISOMFICS and a fic $c \in FT$, c is said to be* explained *if it implies one or more isolated mfics, i.e.,*

$$isExplained(c, ISOMFICS) \equiv$$
$$\exists S \in \mathcal{P}(ISOMFICS) : c \rightarrow \bigwedge_{m \in S} m$$

The rationale is that if a fic c contains[1] one or more iso-mfics, the failure of the tests T_c in which c fails can be explained. Of course, this is just a heuristic, and some other test could show that actually c is the true mfic. The definition of explained fic will be used in the process to balance between *exploitation* at strength t and *exploration* of higher strengths.

Tuple sets initialization

Initially, *PT* contains all the tuples of size t that are contained in a passing test of *TS* (line 1); *FT*, instead, inherits the failing tuples from previous iteration, and is enriched

[1] Note that, for conciseness, in the definition we use the propositional representation of tuples.

with t-tuples that are contained in a failing test, but not in a passing test (line 2). *UT* is initialized with the remaining tuples of size t (line 3). *ISOMFICS* is kept from the previous step.

Exit condition

The process exits as soon as no unknown tuples *UT* are present, and either there are no failing tuples or all the failing tuples are *explained* (see Def. 8.9), i.e.,

$$UT = \emptyset \wedge (FT = \emptyset \vee (\forall c \in FT\colon \mathit{isExplained}(c, TS))) \tag{8.1}$$

Test case generation

If the exit condition is not met (i.e., there is at least an unknown tuple (*UT*) or a failing tuple (*FT*) that is not explained), the function *buildTest* generates a test f that contains at least one tuple belonging to either *UT*, or to *FT* without being explained by any subset of mfics in *ISOMFICS*. The generation works as shown in Alg. 5 and described as follows:

Algorithm 5 BUILDTEST: Test case generation

```
Require: $TS$: the tests generated so far
Require: $UT$: unknown tuples
Require: $FT$: failing tuples
 1: $f \leftarrow \emptyset$
 2: if $UT \neq \emptyset$ then
 3:   for $c \in UT$ do
 4:     if $compatible(c, f)$ then
 5:       $f \leftarrow f \cup c$
 6:     end if
 7:     if $isCompleteTest(f)$ then
 8:       return $f$
 9:     end if
10:   end for
11:   $f \leftarrow completeRnd(f)$
12:   return $f$
13: else
14:   $c_{ne} \leftarrow pickRnd(\{c \in FT | \neg isExplained(c, TS)\})$
15:   $\phi \leftarrow c_{ne} \wedge \bigwedge_{\{f \in TS | c_{ne} \subseteq f\}} \neg f$
16:   return getModel($\phi$)                    ▷ A test is a satisfying assignment
17: end if
```

- if *UT* is not empty, we merge together as many tuples $c \in UT$ as possible (lines 3-12). Two tuples c and c' cannot be merged if, for a given parameter p_i, p_i has different values in c and in c' (this is captured by predicate *compatible* at line 4). If after this phase some parameters have no associated value, we randomly generate values for them (line 11);

- if instead *UT* is empty, we randomly select a not-explained failing tuple *c* (line 14); then, in lines 15-16 we ask the SMT solver to find a test that contains *c*, but it is different from previous tests in *TS* (this is guaranteed to exist, as shown in Thm. 8.3).

Test execution and tuple sets update

After each test f is generated, it is immediately executed, and, depending on the *result* (pass/fail), the tuple sets are updated as described in Alg. 6:

Algorithm 6 updateTupleSets: Tuple sets update

```
Require: $f$: a test
 1: if $result(f) = fail$ then
 2:   Move($f$, $UT$, $FT$)
 3: else
 4:   Move($f$, $UT$, $PT$)
 5:   Move($f$, $FT$, $PT$)
 6:   Move($f$, $ISOMFICS$, $PT$)
 7: end if
 8: procedure move($f$, $sourceSet$, $destSet$)
 9:   $toMove \leftarrow \{(c \in sourceSet) | c \subseteq f\}$
10:   $sourceSet \leftarrow sourceSet \setminus toMove$
11:   $destSet \leftarrow destSet \cup toMove$
12: end procedure
```

1. If the test f fails, all combinations that are contained both in f and in the set *UT*, are moved from *UT* to *FT*;
2. If the test passes, we can exploit Obs. 1 and modify the sets as follows:
   a) all combinations that are contained both in f and in the set *UT*, are moved from *UT* to *PT* (line 4);
   b) all combinations that are contained both in f and in the set *FT*, are moved from *FT* to *PT* (line 5);
   c) all combinations that are contained both in f and in the set *ISOMFICS*, are moved from *ISOMFICS* to *PT* (line 6).

At this point, we can evaluate whether there are new isolated mfics, with the procedure shown in Alg. 7.

If a tuple $c \in FT$ turns out to be the only one (amongst all the possible tuples) to explain the failure of a test f (i.e., it is isolated in f according to Def. 8.8), it is added to *ISOMFICS*.

Algorithm 7 updateMFICS: *ISOMFICS* set update

```
Require: $TS$: the tests generated so far
Require: $ISOMFICS$: isolated mfics
Require: $FT$: failing tuples
for $c \in FT$ do
  if $isIsoMfic(c, TS)$ then
    $FT \leftarrow FT \setminus \{c\}$
    $ISOMFICS \leftarrow ISOMFICS \cup \{c\}$
  end if
end for
```

In summary, the status evolution of a combination c is depicted in the state machine shown in Fig. 41.

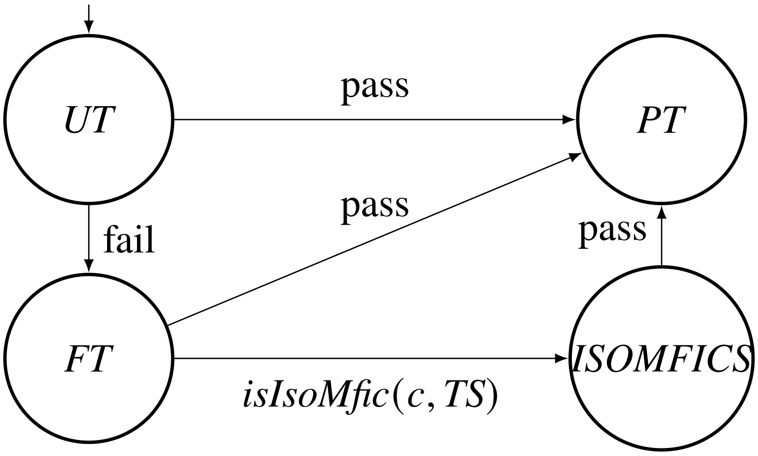


Figure 41: Status evolution of a tuple c throughout the process

1 **Example 8.7.** *On model M presented in Ex. 8.1, if the true-mfics were A and $B\bar{C}$, a possible trace table of the process, with tests separated by the incremental maximum strength t, would be the one presented in Table 28 (the scenario with test 8a).*

We observe that the true-mfics have been correctly identified with the first two executions of MixTgTe$_t$ (till test 6), i.e., tests 7 and 8a (for strength t=3) are not necessary, since the maximum strength of the true-mfics is 2.

Instead, if the true mfics were $A\bar{B}$, AC, and $\bar{A}B\bar{C}$, the process should be run three times for correctly identifying them (using test 8b), since there is a true-mfic of size 3.

8.4 Properties of the MixTgTe process

In this section, we introduce some theorems assessing the capabilities of the proposed process.

	#	A	B	C	*result*	*ISOMFICS*	*FT*	*PT*	*UT*
$t = 1$	Fill FT-PT-UT					$\{\}$	$\{\}$	$\{\}$	$\{A, B, C, \bar{A}, \bar{B}, \bar{C}\}$
	1	0	0	0	pass	$\{\}$	$\{\}$	$\{\bar{A}, \bar{B}, \bar{C}\}$	$\{A, B, C\}$
	2	1	1	1	fail	$\{\}$	$\{A, B, C\}$	$\{\bar{A}, \bar{B}, \bar{C}\}$	$\{\}$
	3	0	1	1	pass	$\{\}$	$\{A\}$	$\{\bar{A}, \bar{B}, \bar{C}, B, C\}$	$\{\}$
	updatedMfics					$\{A\}$	$\{\}$	$\{\bar{A}, \bar{B}, \bar{C}, B, C\}$	$\{\}$
$t = 2$	Fill FT-PT-UT					$\{A\}$	$\{AB, AC\}$	$\{\bar{A}\bar{B}, \bar{A}\bar{C}, \bar{B}\bar{C}, \bar{A}B, \bar{A}C, BC\}$	$\{A\bar{B}, A\bar{C}, \bar{B}C, B\bar{C}\}$
	4	1	0	0	fail	$\{A\}$	$\{AB, AC, A\bar{B}, A\bar{C}\}$	$\{\bar{A}\bar{B}, \bar{A}\bar{C}, \bar{B}\bar{C}, \bar{A}B, \bar{A}C, BC\}$	$\{B\bar{C}, \bar{B}C\}$
	5	0	1	0	fail	$\{A\}$	$\{AB, AC, A\bar{B}, A\bar{C}, B\bar{C}\}$	$\{\bar{A}\bar{B}, \bar{A}\bar{C}, \bar{B}\bar{C}, \bar{A}B, \bar{A}C, BC\}$	$\{\bar{B}C\}$
	updatedMfics					$\{A, B\bar{C}\}$	$\{AB, AC, A\bar{B}, A\bar{C}\}$	$\{\bar{A}\bar{B}, \bar{A}\bar{C}, \bar{B}\bar{C}, \bar{A}B, \bar{A}C, BC\}$	$\{\bar{B}C\}$
	6	0	0	1	pass	$\{A, B\bar{C}\}$	$\{AB, AC, A\bar{B}, A\bar{C}\}$	$\{\bar{A}\bar{B}, \bar{A}\bar{C}, \bar{B}\bar{C}, \bar{A}B, \bar{A}C, BC, \bar{B}C\}$	$\{\}$
$t = 3$	Fill FT-PT-UT					$\{A, B\bar{C}\}$	$\{AB, AC, A\bar{B}, A\bar{C}, ABC, \bar{A}B\bar{C}, A\bar{B}\bar{C}\}$	$\{\bar{A}\bar{B}\bar{C}, \bar{A}BC, \bar{A}\bar{B}C\}$	$\{A\bar{B}C, AB\bar{C}\}$
	7	1	0	1	fail	$\{A, B\bar{C}\}$	$\{AB, AC, A\bar{B}, A\bar{C}, A\bar{B}C, A\bar{B}\bar{C}, ABC, \bar{A}B\bar{C}\}$	$\{\bar{A}\bar{B}\bar{C}, \bar{A}BC, \bar{A}\bar{B}C\}$	$\{AB\bar{C}\}$
	a) Scenario in which test 8 fails								
	8a	1	1	0	fail	$\{A, B\bar{C}\}$	$\{AB, AC, A\bar{B}, A\bar{C}, A\bar{B}C, A\bar{B}\bar{C}, ABC, \bar{A}B\bar{C}, AB\bar{C}\}$	$\{\bar{A}\bar{B}\bar{C}, \bar{A}BC, \bar{A}\bar{B}C\}$	$\{\}$
	b) Scenario in which test 8 passes								
	8b	1	1	0	pass	$\{\}$	$\{A\bar{B}, AC, A\bar{B}C, A\bar{B}\bar{C}, ABC, \bar{A}B\bar{C}\}$	$\{A, B\bar{C}, AB, A\bar{C}, \bar{A}\bar{B}\bar{C}, \bar{A}BC, \bar{A}\bar{B}C, AB\bar{C}\}$	$\{\}$
	updatedMfics					$\{A\bar{B}, AC, \bar{A}B\bar{C}\}$	$\{A\bar{B}C, A\bar{B}\bar{C}, ABC\}$	$\{A, B\bar{C}, AB, A\bar{C}, \bar{A}\bar{B}\bar{C}, \bar{A}BC, \bar{A}\bar{B}C, AB\bar{C}\}$	$\{\}$

Table 28: Example of MixTgTe for detecting mfics of different sizes with a strength up to $t = 3$

We first make an assumption that is needed for our process.

Assumption 2. *All true-mfics can be isolated.*

Theorem 8.3 (Test case generation)**.** *In the test case generation (see Sect. 8.3.1), it is always possible to generate a test case.*

Proof. When *UT* is not empty, the test f is generated by merging compatible tuples from *UT* and then randomly selecting values for other parameters; since tuples in *UT* are those that have never been observed in any test, the new test f is guaranteed to exist. When *UT* is empty, the generated test must be an assignment satisfying formula ϕ at line 15 of Alg. 5. Let's assume that such test does not exist; it would mean that all the possible tests $T_{c_{ne}}$ containing c_{ne} have already been generated; there would be two cases:

- at least one of the tests in $T_{c_{ne}}$ passes; this is not possible, as, in this case, c_{ne} would be in *PT*;
- all the tests in $T_{c_{ne}}$ fail; also this is not possible, as, in this case, c_{ne} would be either in *ISOMFICS* or explained by a subset of tuples in *ISOMFICS*.

□

Theorem 8.4 (Termination)**.** *The process is guaranteed to terminate.*

Proof. The outer process MixTgTe terminates when the *user* (i.e., the test engineer) decides not to continue it, or when $t = |P|$.

The inner process MixTgTe$_t$ terminates when the exit condition (see Eq. 8.1) is met. The test generation phase (see Sect. 8.3.1) directly aims at emptying *UT* and explaining all the not explained tuples in *FT*. Since, by Thm. 8.3, the generation is always possible, the exit condition will be eventually met. □

We want to prove that, under the assumption that the SUT has only true mfics of limited strength, by running the process till that strength, we will find them.

Assumption 3. *Each true-mfic has maximum strength t.*

Theorem 8.5 (True-mfics found)**.** *If TRUE_MFICS is the set of true mfics and each c in TRUE_MFICS has maximum strength t, then by running the process with strength equal or greater than t, ISOMFICS is equal to TRUE_MFICS.*

Proof. Under the stated assumption, the property is twofold: if c is a true-mfic, the MixTgTe will find it and if the MixTgTe finds a c as mfic, then c is a true-mfic.

1. If c is a true-mfic then *ISOMFICS* will contain c. Let's assume that c is a true-mfic but *ISOMFICS* does not contain it at the end. By Thm. 8.1, c is contained in a failing test of *TS* and so it is in *FT* at a given point. When the process terminates, *FT* is either empty (and so c is in *ISOMFICS*), or all the tuples in *FT* are explained (by Def. 8.9), i.e., they contain one or more iso-mfic. However, the latter case is not possible, as c would not be minimal.
2. If c is in *ISOMFICS*, it is a true-mfic. Let's assume that c is in *ISOMFICS*, but it's not a true-mfic. If c is in *ISOMFICS*, all tests containing it fail, and there exists a test in which it is the only mfic; if it is not a true-mfic, it means that in each test containing c there must be a combination c' such that $c \subset c'$ and c' is a true-mfic and, therefore, added to *ISOMFICS* by point 1: in this case, c' would violate the minimality requirement. Note that, if c has size t, there cannot be a true-mfic c' of higher strength containing c by Assumption 3.

□

8.5 Evaluation

In this section, we evaluate the process and we compare it with other techniques for fault interaction detection.

8.5.1 Benchmarks

For the experiments, we selected some benchmarks, each one constituted by a faulty version of the SUT S_f and an oracle O. The assessment of the execution of a test f (i.e., *result* in Def. 8.2) is performed by comparing the evaluations of f over S_f and O. For practicality, we build a combinatorial model M having the same parameters of S_f[2] and constraints that accept only the tests for which S_f and O agree (the constraints are the negation of the true-mfics).[3] Therefore, for each benchmark, we also know the true-mfics in S_f.

We used two sets of benchmarks described in Table 29.

	name	$\|P\|$	size	*TRUE_MFICS* (size (#))
BENCH$_{\mathtt{ART}}$	runExA	3	2^3	1(1), **2**(1)
	runExB	3	2^3	2(2), **3**(1)
	art1	3	2^3	**2**(1)
	art2	3	2^3	**2**(2)
	art3	3	2^3	1(1), **2**(1)
	art4	7	2^7	2(1), **3**(1)
	art5	7	2^7	**2**(1)
	art6	5	$2^3 3^1 5^1$	**3**(1)
BENCH$_{\mathtt{REAL}}$	aircraft	8	$2^7 3^1$	3(1), **4**(1)
	tomcat	12	$2^8 3^1 4^1$	1(1), **2**(2)
	hsqldb	10	$2^9 6^1$	1(1), **3**(2)
	gcc	10	$2^8 3^1 4^1$	**3**(4)
	jflex	13	$2^{10} 3^2 4^1$	**2**(1)

Table 29: Benchmark properties

The first benchmark set, BENCH$_{\mathtt{ART}}$, is constituted by *artificial* models of systems; we generated some of these models with one true-mfic (art1, art5, and art6), and others with multiple true-mfics. BENCH$_{\mathtt{ART}}$ also contains the running example, in its two versions

[2] Note that S_f and O have the same parameters and they only differ on the behavior.

[3] Note that these constraints are not related to the combinatorial problem that, as stated in Sect. 8.1, is unconstrained in our setting.

shown in Table 28. The second benchmark set, $\texttt{BENCH}_{\texttt{REAL}}$, represents real systems: `aircraft` is a Software Product Line model presented in [204] and taken from the SPLOT repository[4], and the others four are benchmarks used in Niu et al. [159].

In Table 29, column *size* reports the size of model M, presented in the abbreviated form $k^{\#params_k} \times \ldots$, where $k \in N^+$ and $params_k$ are the parameters having k values; for example, $2^8 3^1 4^1$ indicates that the SUT has 8 parameters that can take 2 values, one parameter taking 3 values, and one parameter taking 4 values. Column *TRUE_MFICS* reports the number and size of true-mfics in M_f; we report each possible size with the number of mfics of that size in parentheses. We also mark in bold face the maximum strength of the true-mfic; in the experiments, we assume Assumption 3, and so we apply the approach only up to the known maximal strength (according to Thm. 8.5, this guarantees to find all the mfics).

Note that we introduced a set of artificial benchmarks, $\texttt{BENCH}_{\texttt{ART}}$, as part of our evaluation process, for different reasons, namely: (1) to study the behavior of our process, MixTgTe, on models known to have a single true-mfic, while varying the size of the true-mfic and of the model; (2) to have a fair comparison with other existing methods (in particular FIC and SOFOT, which will be presented below, in Sect. 8.5.2), that work accurately only when there is a single mfic; (3) to observe the behavior of MixTgTe when applied to models of *gradually increasing complexity*, from models as simple as the running example, till slightly smaller models than the ones in $\texttt{BENCH}_{\texttt{REAL}}$; (4) to collect and show experiment results on the different behavior of MixTgTe with models made only by boolean parameters (and true-mfic of different sizes), w.r.t. models with multi-value parameters, but a single true-mfic.

8.5.2 Compared approaches

[28] We compare our approach with some existing methods from literature, namely:

BEN: a process based on the first phase of the BEN tool proposed by Ghandehari et al. [96]. The process consists in calling BEN for failure-inducing combination detection, by providing an initial combinatorial test suite of a certain strength t, and iterating over the size of the failure-inducing combinations to try to detect them. This process has already been used for constraints validation and repair [88, 87]. The BEN tool is included in our experimental process as a jar file.

[4] http://www.splot-research.org/

SOFOT: the *Simplified One Factor One Time* method to infer the Minimal Failure-causing Schema (MFS) from a given failing test case, from Nie et al. [155]. This method takes as input a set of failing tests, and tries to reduce each test to an mfic. For each failing test f, it generates new tests by changing the value of each parameter in f one by one. Note that the source code of this method is available in Python from a later work by Zhang and Zhang [219]. As our automated evaluation script is written in Java, we program it so that it calls Python via command line. This causes some overhead which affects the total execution time in the experiments.

FIC: the *Faulty Interaction Characterization* method proposed by Zhang et al. [219]. It is similar to SOFOT, in the sense that it accepts in input a set of tests known to be failing, and it tries to isolate the minimal failure-inducing combination(s) from it. It proceeds by considering one failing test a time, and changing the value of a parameter at a time, but, unlike SOFOT, it keeps the value changed. Furthermore, it performs a few iterations until the original failing test, with the value of the detected minimal failure-inducing combinations changed, passes. If there are two different failure-inducing combinations in the same tests, it may find them, but without a guarantee to be correct. We made a Java implementation of the algorithm described in the paper [219].

ICT: the *Interleaving CT* approach proposed by Niu et al. [159]. It is a significant improvement of SOFOT that alleviates its three main problems: redundant test cases, multiple mfics, and masking effects (where multiple mfics are present in the same test). Like SOFOT, it is composed of two phases, generation and identification. Test generation is here made adaptive, one test at a time, in a similar way as the one of our approach. This reduces the amount of tests needed, by forbidding the generation of new tests containing already discovered failure-inducing combinations. The identification phase has a novel feedback checking mechanism (based on information coming from the execution of a few new proposed test cases), which can check, up to a certain extent, whether the identified mfic is a true-mfic or not; and it significantly improves the accuracy of the results w.r.t. SOFOT. The method is very recent, and, although we could not manage to re-run the tool on new benchmarks, we compared the results of MixTgTe with the results of ICT reported in that paper for a common set of benchmarks.

Since FIC and SOFOT require failing tests as input, but do not say how to find such failing tests, we need to build a test suite to find such failing tests. In order to try to make the comparison fair, we use, for all the methods[5], an initial combinatorial test suite CTS_t of strength $t = \max_{c \in TRUE\_MFICS} |size(c)|$, being *TRUE_MFICS* the set of true-mfics (as shown in Table 29). CTS_t is generated using ACTS[6] for FIC, BEN, and SOFOT. MixTgTe, instead, generates tests in an adaptive way, as described in Sect. 8.3.1. ICT, instead, uses AETG [62].

In the following, *DET_MFICS* denotes the set of mfics returned by a method; in our case, it corresponds to *ISOMFICS*.

8.5.3 Results

We run our method and the compared 4 methods 10 times for each benchmark; results are the average across the runs. Experiments have been executed on a Mac OS X 10.14, Intel Core i3, with 4GB of RAM. Code was written in Java, using CTWedge libraries for combinatorial modeling, test generation, and test execution [89]. The code and all the benchmarks are available online at https://github.com/fmselab/mixtgte.

Table 30 shows the results of the experiments.

	model	MixTgTe					FIC					BEN					SOFOT					ICT				
		tests	P	R	F	time	tests	P	R	F	time	tests	P	R	F	time	tests	P	R	F	time	tests	P	R	F	time
$BENCH_{ART}$	runExA	7.6	1	1	1	15.9	8	1.00	1.00	1.00	0.4	7	0.20	0.50	0.29	53.2	8	0.75	0.50	0.58	518	–	–	–	–	–
	runExB	8.0	1	1	1	4.9	8	0.75	1.00	0.86	0.6	8	0.25	0.33	0.29	9.4	8	0.75	1.00	0.86	735	–	–	–	–	–
	art1	6.6	1	1	1	3.2	5	1.00	1.00	1.00	0.5	7	1.00	1.00	1.00	13.0	7	1.00	1.00	1.00	269	–	–	–	–	–
	art2	8.0	1	1	1	5.4	7	1.00	1.00	1.00	0.6	7	0.50	0.50	0.50	13.8	8	0.50	0.50	0.50	396	–	–	–	–	–
	art3	7.6	1	1	1	2.7	8	1.00	1.00	1.00	0.7	7	0.00	0.00	–	15.1	8	1.00	0.50	0.67	403	–	–	–	–	–
	art4	36.9	1	1	1	79.7	29	0.67	1.00	0.80	1.8	26	0.00	0.00	–	64.8	61	1.00	1.00	1.00	2772	–	–	–	–	–
	art5	14.7	1	1	1	5.0	12	1.00	1.00	1.00	0.5	18	1.00	1.00	1.00	14.1	20	1.00	1.00	1.00	769	–	–	–	–	–
	art6	45.4	1	1	1	17.1	35	0.50	1.00	0.67	3.2	38	1.00	1.00	1.00	21.1	52	1.00	1.00	1.00	1830	–	–	–	–	–
$BENCH_{REAL}$	aircraft	89.8	1	1	1	156.5	71	1.00	1.00	1.00	18.4	62	0.00	0.00	–	56.4	115	1.00	1.00	1.00	4157	–	–	–	–	–
	gcc	88.5	1	1	1	134.5	64	0.50	0.50	0.50	8.9	60	1.00	0.50	0.67	56.1	142	0.75	0.75	0.75	4934	89.0	0.77	0.65	0.70	1118
	hsqldb	169.8	1	1	1	3752.8	97	1.00	1.00	1.00	36.6	55	0.00	0.00	–	532.2	443	1.00	0.67	0.80	19612	88.3	1.00	1.00	1.00	2094
	jflex	22.9	1	1	1	9.1	15	1.00	1.00	1.00	1.6	23	1.00	1.00	1.00	15.2	31	1.00	1.00	1.00	978	31.6	1.00	1.00	1.00	187
	tomcat	65.9	1	1	1	111.9	28	0.67	0.67	0.67	4.1	23	0.00	0.00	–	31.5	128	1.00	1.00	1.00	5675	128	1.00	1.00	1.00	5671
	Average	44.0	1	1	1	331	29.8	0.85	0.94	0.88	5.99	26.2	0.46	0.45	0.44	68.9	79.3	0.90	0.84	0.86	3311	67.4	0.94	0.91	0.92	988

Table 30: Experimental results (P: precision, R: recall, F: F-score, time is in ms)

For each method, it reports the total number of different tests required to complete the detection[7], and the execution time in milliseconds. Moreover, in order to measure the *quality* of the returned mfics, we use classical measures as *precision* (P), *recall* (R), and

[5] Note that MixTgTe, BEN, and ICT already require a combinatorial test suite.

[6] https://csrc.nist.gov/projects/automated-combinatorial-testing-for-software

[7] If a test is generated twice by a method, we count it only once.

F-score (F). Precision is defined as:

$$precision = \frac{|DET\_MFICS \cap TRUE\_MFICS|}{|DET\_MFICS|}$$

Precision measures the percentage of found mfics that are true-mfics. If precision is not 1, the developer will spend some time in doing fault localization for a fic that is not a true-mfic (those in *DET_MFICS* \ *TRUE_MFICS*).

Recall is defined as:

$$recall = \frac{|DET\_MFICS \cap TRUE\_MFICS|}{|TRUE\_MFICS|}$$

It measures how many true-mfics are actually identified. If the recall is not 1, the developer is not aware of a true-mfic that causes a fault (those in *TRUE_MFICS* \ *DET_MFICS*).

The F-measure is the combination of precision and recall, defined as follows:

$$F\text{-}score = \frac{2 \times precision \times recall}{precision + recall}$$

We now evaluate the approach answering the following three research questions.

RQ1: *How is the effectiveness (in terms of precision and recall) of MixTgTe w.r.t. other techniques?*

From the results presented in Table 30, we observe that MixTgTe always achieves maximum precision and recall; this is expected, as Thm. 8.5 guarantees that, under the assumption that we know the maximum strength t, executing MixTgTe till strength t produces an *ISOMFICS* set (i.e., *DET_MFICS*) equal to *TRUE_MFICS*. All the other techniques do not provide this theoretical guarantee.

Among the other methods, ICT has the highest values for precision, recall, and F-score (92% on average) on the 4 available benchmarks [159]. For 3 benchmarks, ICT correctly identified all the *TRUE_MFICS*; only for *gcc*, some are wrongly identified (precision 77%) and some are not found (recall 65%).

Also FIC and SOFOT showed to be able to correctly identify the true-mfics in many occasions, although not with the same overall accuracy as ICT in terms of F-score (88% and 86%). We believe that this is due not only to the fixed amount of tests asked for the *identification* phase of those methods (they change always one parameter at a time, and only once), but also to the *masking effect*, i.e., when there are two mfics present in a same test. This effect may happen in general, as explained in Thm. 8.2. As an example of this fact, consider the running example described in Ex. 8.1 and the test suite generated by SOFOT shown in Table 31. Let's recall that the SUT is made of three binary parameters

Generation of CTS_2 with ACTS (to have some failing test)

#	A	B	C	*result*
1	1	1	0	failing test ①
2	1	0	1	failing test ②
3	0	1	1	pass
4	0	0	0	pass

Identification by SOFOT (tests added to find mfics)

additional tests for failing test ①

#	A	B	C	*result*
5	0	1	0	fail
6	1	0	0	fail
7	1	1	1	fail

No mfic found

additional tests for failing test ②

#	A	B	C	*result*
8	0	0	1	pass
–	1	1	1	fail
–	1	0	0	fail

A identified as mfic

Table 31: Execution trace of SOFOT on example SUT

{A, B, C}, with two true-mfics, A and $B\bar{C}$. By providing to SOFOT the faulty test cases observed with a combinatorial test suite of strength $t = 2$ (that it is also the maximum strength of the true-mfics, so the correct settings for the experiments) generated with ACTS, the SOFOT method is able to correctly identify only the mfic A. Table 31 reports, at the beginning, the CTS_2 generated by ACTS; it contains two failing tests for which SOFOT tries to find the mfic. In test ①, both true-mfics A and $B\bar{C}$ are contained; all the additional tests generated by SOFOT for this test (obtained by changing one parameter at a time) fail. Therefore, for test ①, SOFOT does not find any mfic, i.e., it does not find A nor $B\bar{C}$. This is due to the masking effect in test ① between A and $B\bar{C}$. The tests generated for the failing test ②, instead, correctly identifies A as mfic.

BEN is the method with the lowest F-score; this is because BEN is configured to produce few additional tests and uses heuristics to measure the suspiciousness of a failing tuple, and this may lead to wrong results.

RQ2: *How does our approach compare with the others in terms of number of tests?*

Overall, the number of tests required by MixTgTe is comparable to ICT. On the four real benchmarks in common, MixTgTe requires slightly fewer tests for `gcc` and `jflex`, but more tests for the other two benchmarks. For `hsqldb`, MixTgTe requires almost the double of the tests. This is due to the fact that ICT applies efficient heuristics to limit the number of tests that are asked in addition to the initial combinatorial test suite; MixTgTe, instead, does not have such strong optimizations, that we plan to investigate as future work. For this particular benchmark `hsqldb`, ICT is better (or equal) than our approach on any aspect (it also achieves 100% F-score); however, it does not provide any particular correctness guarantee.

SOFOT requires the highest amount of tests, and it obtains a lower recall, but a higher precision than FIC. The other two analyzed methods (FIC and BEN) require fewer tests (almost half of the test of MixTgTe on average), but, as described in **RQ1**, they also achieve less precision and recall than both MixTgTe and ICT.

RQ3: *How does our approach compare with the others in terms of time?*

All the reported times (for all the approaches) do not include the time for actually exercising the real system to determine the *result* (pass/fail) of the test. Indeed, the real system has been mocked by a model, since we know the true-mfics beforehand.

We cannot directly compare the execution time of ICT and SOFOT. Indeed, we were not able to rerun ICT on our machine (we report the results of the original paper [159]). For SOFOT, instead, we need to perform calls to an external Python program from Java, that introduce a big overhead.

The execution time for our process varies a lot depending on the number of generated tests, and the maximum strength achieved. It is less than 20ms for more than half of the benchmarks; however, it takes around 3.8 secs for `hsqldb`, which has two true-mfics of size 3, and one of size 1. The mfic of size 1 causes several tests to fail, masking the effect of the 3-way mfics. Note that, although `gcc` has four 3-way true-mfics, it takes less computation time because less tests are needed to isolate the mfics from the other failing tuples, as more tests are passing.

Generally, BEN is quite fast as it does not produce too many tests, and the time is not affected too much by the model size; in our case, instead, time is more dependent on the benchmark characteristics (model size, number of true-mfics, presence of masking effect, etc.).

FIC is the fastest method, as it only requires, on average, around 6ms per benchmark, with a maximum time of 36.6ms for `hsqldb`.

8.6 Related Work

Identifying the real failure inducing combinations is an area of active research in combinatorial testing [155, 127].

Previous works in detecting failure-inducing interactions are based on post-analysis of the test results of covering arrays (CAs), or on adaptive or non-adaptive test generation techniques. Yilmaz et al. [214] applied a post-analysis classification tree technique to analyze the result of CAs to find the differences between passing and failing tests. However, CA is not suitable to detect mfics precisely. Among non-adaptive methods, there is an approach based on pseudo-Boolean constraint solving and optimization, but its accuracy is highly affected by the chosen test suite [218]. Locating and detecting arrays (LDAs) [65], and error locating arrays (ELAs) [147] are other non-adaptive approaches: they require a given strength t and a maximum number of faulty interactions d, and they can detect and locate at most d faulty interactions of size up to t. However, the size of the test suite often becomes very large. That is why, recently, adaptive methods appear to be more studied in literature. They include Wang's IterAIFL method [208], which is based on AIFL by Shi et al. [184], two adaptive algorithms proposed by Martinez et al. [147], and all the methods used to compare our process in the experiments: FIC (and also the variant FIC_BS) by Zhang et al. [219], BEN [97], SOFOT [155], and ICT [159].

While InterAIFL, FIC and SOFOT may not correctly identify multiple mfics in a system, since they may be overlapping or there is a masking effect, the two adaptive algorithms of Martinez work better but they can only locate mfics up to size 2. The ICT approach by Niu et al. [159], still derived from SOFOT, overcomes its limitations, making a significant improvement in the accuracy of the detected combinations. BEN [97] is tailored to locate faults in the code, but in the first phase it provides an algorithm to detect *suspicious* combinations and, with some heuristics, *failure-inducing* combinations. However, as implemented so far, it is not very accurate with the initial test suites provided as input: an initial test suite of higher strength could improve accuracy of the detected mfics. Unlike the other methods, MixTgTe does not distinguish between the two phases of the input test generation and additional adaptive test, but it merges those phases into one single process, that keeps track of the status of all the possible t-way tuples throughout the process. This way, MixTgTe has shown to correctly detect all the mfics of a system, up to a certain strength t decided by the user, and it guarantees them to be correct under the assumption that there are no faults caused by an interaction of strength higher than t.

8.7 Conclusions

In this chapter we propose an approach for finding minimal failure-inducing combinations (mfics), that alternates test generation and test execution. Under the assumption that the maximum strength of true-mfics is limited to t, running the process till strength t guarantees to find all and only the true-mfics; experimental comparison with state of the art approaches confirmed this fact. Achieving this total correctness does not affect too much the test suite size and the execution time: w.r.t. the second best approach (ICT) in terms of accuracy, MixTgTe produces slightly fewer tests in reasonable time.

The current work does not support constraints in the combinatorial model; their handling is planned as future work.

Conclusions

In this thesis, we have proposed a method consisting in using model-based software testing techniques to drive the inference and repair of models of software systems. It consists in novel applications of model-based testing that go beyond detecting and localizing faults in code, and that perform little repairs to preserves domain knowledge and support engineers in maintaining consistency between all the software artifacts, and localizing faults also in the model. We presented applications to repair combinatorial models, feature models, and timed automata. The works presented in this thesis have been published in different forms in a journal paper, and in conference papers. The list of publications is reported in Sect. 1.2. In the next sections, we summarize the contribution of the thesis and describe some possible future lines of research.

Contributions of the thesis

We introduced the goal of this thesis, summarized the contributions, and gave a list of the published papers, in Chapter . In Chapter 2, we described the state of the art of the main techniques in software engineering to support solutions to the problem of model repair.

In Chapter 3, we presented our novel iterative, test-driven automated approach to repair software models. It is composed by three main operations: test generation, fault localization, and model repair. This framework has been applied for three different types of software model, each of them is treated in a different chapter. In Chapter 4 describe the application of the automated approach to combinatorial models, having configuration constraints. Specifically, in Sect. 4.1 we evaluated the quality of combinatorial test generation policies, we described the actual CIT model repair process in Sect. 4.2, and in Sect. 4.3 we showed an application of it to detect constraints among parameters in XSS attack strings, that trigger vulnerability. Empirical evaluation shows that the process achieves on average 37% accuracy on the combinatorial model benchmarks, and 78.8% accuracy for XSS vulnerability condition detection.

In Chapter 5 we presented the application of the repairing process to feature models, in the context of software product line engineering. Specifically, the process is applied to the hierarchical and simple cross-tree constraints in Sect. 5.1 using mutations, and to arbitrarily complex constraints among features in Sect. 5.2 using logical manipulation of propositional formulas. Due to the complexity of the repair phase on its own, both techniques focus only on the model repair part, assuming an *update request* is given in input. Such *update request* can potentially obtained by a fault localization technique, or can come directly from a change in the requirements. Empirical analysis on the devised process that uses mutations to update feature models shows that, on average, around 89% of the requested changes are applied.

Chapter 4 presents the application of the devised repair approach to the scenario of timed automata. By parametrizing the clock guards of the timed automata, and exercising the real-time system with timed words as tests, we can constrain the clock-guards values into ranges, and pick a value inside these ranges to obtain the repaired timed automata. The approach has been evaluated on systems with up to 16 locations and 10 parameters, achieving almost 100% accuracy in most of the cases.

Finally, in Chapter 7 and Chapter 8 we presented two tools to support the first two phases of our process for model repair, respectively testing (CTWEDGE) and fault localization (more precisely, *failure-inducing combinations* detection: MIXTGTE).

Future Work

As future work, we plan to apply test-driven repair of other model types, if necessary through an abstraction and parametrization process (as did in Timed Automata, see Chapter. 6). A possible model is abstract state machines. In addition to following the future directions given in the single chapters, we also plan to improve the process of combinatorial models repair by improving the accuracy of the fault localization strategy.

Although these contributions have also been implemented, the tools are tailored towards producing experiments tables on the benchmarks proposed in the different papers: they need a revision to make them user-friendly. Apart from CTWedge (Chapter 7), which is even hosted and usable as a SaaS, the rest, although the code is accessible on Github, have room for improvement to make them more user-friendly for researchers and engineers: this step could further promote the further development of the findings of our research.

Another future work direction consists in making the processes more automated. In some cases, in fact, manual intervention is needed (to asses the oracle, to decide process parameters, and even just for moving, renaming some files, or translating between file formats), especially for timed automata repair (Chapter 6) and for XSS vulnerabilities conditions detection (Sect. 4.3).

Finally, further work needs to be done to improve scalability of the approch. So far, evaluation shows that these methods terminate in a *reasonable* amount of time with models having up to 100 parameters, and with small numbers in other parameter values (such as maximum test suite strength, or population size for the evolutionary approach of feature models). Further studies and improvements are needed to extend the approach to models of larger size.

List of Figures

List of Tables

References

[1] https://github.com/microsoft/language-server-protocol.

[2] https://hexawise.com/.

[3] https://inductive.no/pairwiser/.

[4] https://osdn.net/projects/pictmaster/.

[5] http://www.ctwebplus.com/.

[6] Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84, Dec 1990.

[7] *Software Product Lines: Practices and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[8] Hadil Abukwaik, Mohammed Abujayyab, Shah Rukh Humayoun, and Dieter Rombach. Extracting conceptual interoperability constraints from API documentation using machine learning. In *Proceedings of the 38th International Conference on Software Engineering Companion - ICSE '16*, pages 701–703. ACM Press, 2016. 00002.

[9] Mathieu Acher, Paul Temple, Jean-Marc Jézéquel, José A. Galindo, Jabier Martinez, and Tewfik Ziadi. Varylatex: Learning paper variants that meet constraints. In *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems*, VAMOS 2018, pages 83–88, New York, NY, USA, 2018. ACM.

[10] Bernhard K. Aichernig, Klaus Hörmaier, and Florian Lorber. Debugging with timed automata mutations. In Andrea Bondavalli and Felicita Di Giandomenico, editors, *SAFECOMP*, volume 8666 of *Lecture Notes in Computer Science*, pages 49–64. Springer, 2014.

[11] Bernhard K. Aichernig, Elisabeth Jöbstl, and Stefan Tiran. Model-based mutation testing via symbolic refinement checking. *Science of Computer Programming*, 97(P4):383–404, January 2015.

[12] Bernhard K. Aichernig, Florian Lorber, and Dejan Nickovic. Time for mutants – Model-based mutation testing with timed automata. In Margus Veanes and Luca Viganò, editors, *TaP*, volume 7942 of *Lecture Notes in Computer Science*, pages 20–38. Springer, 2013.

[13] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, April 1994.

[14] Rajeev Alur, Limor Fix, and Thomas A. Henzinger. Event-clock automata: A determinizable class of timed automata. *Theoretical Computer Science*, 211(1-2):253–273, 1999.

[15] Rajeev Alur, Thomas A. Henzinger, and Moshe Y. Vardi. Parametric real-time reasoning. In S. Rao Kosaraju, David S. Johnson, and Alok Aggarwal, editors, *STOC*, pages 592–601, New York, NY, USA, 1993. ACM.

[16] Benoit Amand, Maxime Cordy, Patrick Heymans, Mathieu Acher, Paul Temple, and Jean-Marc Jézéquel. Towards learning-aided configuration in 3d printing: Feasibility study and application to defect prediction. In *Proceedings of the 13th International Workshop on Variability Modelling of Software-Intensive Systems*, VAMOS '19, pages 7:1–7:9, New York, NY, USA, 2019. ACM.

[17] P. Ammann and J. Offutt. Using formal methods to derive test frames in category-partition testing. In *Computer Assurance, 1994. COMPASS '94 Safety, Reliability, Fault Tolerance, Concurrency and Real Time, Security. Proceedings of the Ninth Annual Conference on*, pages 69–79, Jun 1994.

[18] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2008.

[19] Sofia Ananieva, Matthias Kowal, Thomas Thüm, and Ina Schaefer. Implicit constraints in partial feature models. In *Proceedings of the 7th International Workshop on Feature-Oriented Software Development*, FOSD 2016, pages 18–27, New York, NY, USA, 2016. ACM.

[20] Étienne André, Paolo Arcaini, Angelo Gargantini, and Marco Radavelli. Repairing timed automata clock guards through abstraction and testing. In Dirk Beyer and Chantal Keller, editors, *Tests and Proofs*, pages 129–146, Cham, 2019. Springer International Publishing.

[21] Étienne André, Thomas Chatain, Emmanuelle Encrenaz, and Laurent Fribourg. An inverse method for parametric timed automata. *International Journal of Foundations of Computer Science*, 20(5):819–836, October 2009.

[22] Étienne André, Laurent Fribourg, Ulrich Kühne, and Romain Soulat. IMITATOR 2.5: A tool for analyzing robustness in scheduling problems. In Dimitra Giannakopoulou and Dominique Méry, editors, *FM*, volume 7436 of *Lecture Notes in Computer Science*, pages 33–36. Springer, August 2012.

[23] Étienne André, Ichiro Hasuo, and Masaki Waga. Offline timed pattern matching under uncertainty. In Anthony Widjaja Lin and Jun Sun, editors, *ICECCS*, pages 10–20. IEEE CPS, 2018.

[24] Étienne André and Shang-Wei Lin. Learning-based compositional parameter synthesis for event-recording automata. In Ahmed Bouajjani and Silva Alexandra, editors, *FORTE*, volume 10321 of *Lecture Notes in Computer Science*, pages 17–32. Springer, 2017.

[25] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.

[26] Paolo Arcaini, Angelo Gargantini, and Marco Radavelli. An evolutionary process for product-driven updates of feature models. In *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems*, VAMOS 2018, pages 67–74, New York, NY, USA, 2018. ACM, ACM.

[27] Paolo Arcaini, Angelo Gargantini, and Marco Radavelli. Achieving change requirements of feature models by an evolutionary approach. *Journal of Systems and Software*, 150:64–76, 2019.

[28] Paolo Arcaini, Angelo Gargantini, and Marco Radavelli. Efficient and guaranteed detection of t-way failure-inducing combinations. In *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 200–209. IEEE, IEEE, April 2019.

[29] Paolo Arcaini, Angelo Gargantini, and Marco Radavelli. A process for fault-driven repair of constraints among features. In *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume B*, SPLC '19, pages 71:1–71:9, New York, NY, USA, 2019. ACM.

[30] Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. How to optimize the use of SAT and SMT solvers for test generation of Boolean expressions. *The Computer Journal*, 58(11):2900–2920, 2015.

[31] Paolo Arcaini, Angelo Gargantini, and Paolo Vavassori. Validation of models and tests for constrained combinatorial interaction testing. In *The 3rd International Workshop on Combinatorial Testing (IWCT 2014) In conjunction with International Conference on Software Testing ICSTW*, pages 98–107. IEEE, 2014.

[32] Paolo Arcaini, Angelo Gargantini, and Paolo Vavassori. Generating tests for detecting faults in feature models. In *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*, pages 1–10, April 2015.

[33] Paolo Arcaini, Angelo Gargantini, and Paolo Vavassori. Automatic detection and removal of conformance faults in feature models. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 102–112, April 2016.

[34] Paolo Arcaini, Angelo Gargantini, and Paolo Vavassori. Automated repairing of variability models. In *Proceedings of the 21st International Systems and Software Product Line Conference - Volume A*, SPLC '17, pages 9–18, New York, NY, USA, 2017. ACM.

[35] Paolo Arcaini, Pavel Ježek, and Jan Kofroň. Modelling the hybrid ertms/etcs level 3 case study in s pin. In *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 277–291. Springer, 2018.

[36] Andrea Arcuri. Evolutionary repair of faulty software. *Applied Soft Computing*, 11(4):3494 – 3514, 2011.

[37] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, May 2015.

[38] Don Batory. Feature models, grammars, and propositional formulas. In *Proceedings of the 9th International Conference on Software Product Lines*, SPLC'05, pages 7–20, Berlin, Heidelberg, 2005. Springer-Verlag.

[39] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615–636, 2010.

[40] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Lectures on Concurrency and Petri Nets, Advances in Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 87–124. Springer, 2003.

[41] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. A survey of variability modeling in industrial practice. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, VaMoS '13, pages 7:1–7:8, New York, NY, USA, 2013. ACM.

[42] Armin Biere. Preprocessing and inprocessing techniques in SAT. In Kerstin Eder, João Lourenço, and Onn Shehory, editors, *Hardware and Software: Verification and Testing*, pages 1–1, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[43] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *International conference on tools and algorithms for the construction and analysis of systems*, pages 193–207. Springer, 1999.

[44] Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*, volume 185. IOS press, 2009.

[45] Darius Blasband. *The Rise and Fall of Software Recipes*. Reality Bites Publishing, 2016.

[46] Silvia Bonfanti, Andrea Bombarda, Angelo Gargantini, Marco Radavelli, Feng Duan, and Yu Lei. Combining model refinement and test generation for conformance testing of the ieee phd protocol using abstract state machines. In *Testing Software and Systems - 31th IFIP WG International Conference, ICTSS 2019, Paris, France, October 15-17, 2018, Proceedings*, 2019.

[47] Jan Bosch. Software product families in nokia. In Henk Obbink and Klaus Pohl, editors, *Software Product Lines*, pages 2–6, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[48] Josip Bozic, Bernhard Garn, Ioannis Kapsalis, Dimitris Simos, Severin Winkler, and Franz Wotawa. Attack pattern-based combinatorial testing with constraints for web security testing. In *IEEE Int. Conf. on Software Quality, Reliability and Security*, 2015.

[49] M. Brambilla, J. Cabot, and M. Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool, 2012.

[50] R. Brownlie, J.Prowse, and M.S. Phadke. Robust testing of AT&T PMX/starMAIL using OATS. *AT&T Technical Journal*, 71(3):41–47, 1992.

[51] Randal E Bryant. Binary decision diagrams. In *Handbook of model checking*, pages 191–217. Springer, 2018.

[52] Renée C. Bryce and Charles J. Colbourn. Prioritized interaction testing for pair-wise coverage with seeding and constraints. *Information and Software Technology*, 48(10):960 – 970, 2006. Advances in Model-based Testing.

[53] David Buchfuhrer and Christopher Umans. The complexity of boolean formula minimization. *J. Comput. Syst. Sci.*, 77(1):142–153, January 2011.

[54] Johannes Bürdek, Timo Kehrer, Malte Lochau, Dennis Reuling, Udo Kelter, and Andy Schürr. Reasoning about product-line evolution using complex feature model differences. *Automated Software Engineering*, 23(4):687–733, Dec 2016.

[55] K. Burr and W. Young. Combinatorial test techniques: Table-based automation, test generation, and code coverage. In *Proceedings of the Intl. Conf. on Software Testing Analysis and Review*, pages 503–513, October 1998.

[56] Andrea Calvagna and Angelo Gargantini. Combining satisfiability solving and heuristics to constrained combinatorial interaction testing. In Catherine Dubois, editor, *TAP*, volume 5668 of *Lecture Notes in Computer Science*, pages 27–42. Springer, 2009.

[57] Andrea Calvagna and Angelo Gargantini. A formal logic approach to constrained combinatorial testing. *Journal of Automated Reasoning*, 45(4):331–358, 2010. Springer.

[58] Andrea Calvagna, Angelo Gargantini, and Paolo. Vavassori. Combinatorial interaction testing with CitLab. In *Sixth IEEE International Conference on Software Testing, Verification and Validation - Testing Tool track*, 2013.

[59] CASA: Covering arrays by simulated annealing.

[60] Fei Chiang and Renee J. Miller. A unified model for data and constraint repair. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 446–457. IEEE, 2011. 00046.

[61] John Joseph Chilenski and Steven P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9:193–200(7), September 1994.

[62] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG system: An approach to testing based on combinatorial design. *IEEE Transactions On Software Engineering*, 23(7):437–444, Jul 1997.

[63] M.B. Cohen, M.B. Dwyer, and Jiangfan Shi. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. *Software Engineering, IEEE Trans. on*, 34(5):633–650, 2008.

[64] Myra Cohen, Matthew B. Dwyer, and Jiangfan Shi. Interaction testing of highly-configurable systems in the presence of constraints. In *ISSTA International symposium on Software testing and analysis*, ISSTA '07, pages 129–139, New York, NY, USA, 2007. ACM, ACM Press.

[65] Charles J. Colbourn and Daniel W. McClary. Locating and detecting arrays for interaction faults. *Journal of Combinatorial Optimization*, 15(1):17–48, January 2008.

[66] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. 01 2000.

[67] Jacek Czerwonka. Pairwise testing in the real world: Practical extensions to test-case scenarios. In *Proceedings of 24th Pacific Northwest Software Quality Conference, Citeseer*, pages 419–430, 2006.

[68] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *International Conference on Software Engineering ICSE*, pages 285–295, New York, May 1999. Association for Computing Machinery.

[69] S.R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, and C.M. Lott. Model-based testing of a highly programmable system. In *Software Reliability Engineering, 1998. Proceedings. The Ninth International Symposium on*, pages 174–179, 1998.

[70] Jean-Marc Davril, Edouard Delfosse, Negar Hariri, Mathieu Acher, Jane Clelang-Huang, and Patrick Heymans. Feature model extraction from large collections of informal product descriptions, August 22 2013.

[71] Edwin D. de Jong, Richard A. Watson, and Jordan B. Pollack. Reducing bloat and promoting diversity using multi-objective methods. In *Proceedings of the 3rd Annual Conference on Genetic and Evolutionary Computation*, GECCO'01, pages 11–18, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[72] Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[73] Fabien Duchene, Roland Groz, Sanjay Rawat, and Jean-Luc Richier. Xss vulnerability detection using model inference assisted evolutionary fuzzing. In *IEEE Int. Conference on Software Testing, Verification and Validation (ICST)*, pages 815–817, 2012.

[74] I. S. Dunietz, W. K. Ehrlich, B.D. Szablak, C.L. Mallows, and A. Iannino. Applying design of experiments to software testing. In IEEE/Computer Society, editor, *Proc. Int'l Conf. Software Eng. (ICSE)*, pages 205–215, 1997.

[75] Amador Durán, David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-Cortés. FLAME: a formal framework for the automated analysis of software product lines validated by automated specification testing. *Software & Systems Modeling*, 16(4):1049–1082, Oct 2017.

[76] A. Egyed. Scalable consistency checking between diagrams - the VIEWINTEGRA approach. In *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, pages 387–390, San Diego, CA, USA, 2001. IEEE Comput. Soc.

[77] A. Egyed. A scenario-driven approach to trace dependency analysis. *IEEE Transactions on Software Engineering*, 29(2):116–132, February 2003.

[78] A. Egyed and P. Grunbacher. Automating requirements traceability: Beyond the record & replay paradigm. In *Proceedings 17th IEEE International Conference on Automated Software Engineering,*, pages 163–171, Edinburgh, UK, 2002. IEEE Comput. Soc.

[79] A. Egyed and P. Grunbacher. Identifying requirements conflicts and cooperation: how quality attributes and automated traceability can help. *IEEE Software*, 21(6):50–58, November 2004.

[80] Alexander Egyed. Fixing Inconsistencies in UML Design Models. In *29th International Conference on Software Engineering (ICSE'07)*, pages 292–301, Minneapolis, MN, USA, May 2007. IEEE.

[81] Agoston E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. Springer Verlag, 2003.

[82] Moritz Eysholdt and Heiko Behrens. Xtext: Implement your language faster than the quick and dirty way. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '10, pages 307–309, New York, NY, USA, 2010. ACM.

[83] Johnny Maikeo Ferreira, Silvia Regina Vergilio, and Marcos Antonio Quináiaferreia. A mutation approach to feature testing of software product lines. In *The 25th International Conference on Software Engineering and Knowledge (SEKE) Engineering, Boston, MA, USA, June 27-29, 2013*, pages 232–237. Knowledge Systems Institute Graduate School, 2013.

[84] Eduardo Figueiredo, Nelio Cacho, Claudio Sant'Anna, Mario Monteiro, Uira Kulesza, Alessandro Garcia, Sérgio Soares, Fabiano Ferrari, Safoora Khan, Fernando Castor Filho, and Francisco Dantas. Evolving software product lines with aspects: An empirical study on design stability.

In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 261–270, New York, NY, USA, 2008. ACM.

[85] Michael Forbes, Jim Lawrence, Yu Lei, Raghu N Kacker, and D Richard Kuhn. Refining the in-parameter-order strategy for constructing covering arrays. *Journal of Research of the National Institute of Standards and Technology*, 113(5):287, 2008.

[86] OWASP Foundation. OWASP Top 10 2017. `https://www.owasp.org/index.php/Top_10-2017_A7-Cross-Site_Scripting_(XSS)`. [Online; accessed 19-April-2018].

[87] Angelo Gargantini, Justyna Petke, and Marco Radavelli. Combinatorial Interaction Testing for Automated Constraint Repair. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 239–248. IEEE, March 2017.

[88] Angelo Gargantini, Justyna Petke, Marco Radavelli, and Paolo Vavassori. Validation of constraints among configuration parameters using search-based combinatorial interaction testing. In *Search Based Software Engineering - 8th International Symposium, SSBSE 2016, Raleigh, NC, USA, October 8-10, 2016, Proceedings*, pages 49–63, 2016.

[89] Angelo Gargantini and Marco Radavelli. Migrating combinatorial interaction test modeling and generation to the web. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 308–317, 2018.

[90] Angelo Gargantini and Paolo Vavassori. CitLab: a laboratory for combinatorial interaction testing. In *Workshop on Combinatorial Testing (CT) In conjunction with International Conference on Software Testing (ICST 2012, April 17-21)*, pages 559–568, Montreal, Canada, 2012.

[91] Angelo Gargantini and Paolo Vavassori. Efficient combinatorial test generation based on multivalued decision diagrams. In Eran Yahav, editor, *Hardware and Software: Verification and Testing, Haifa Verification Conference HVC 2014*, volume 8855 of *Lecture Notes in Computer Science*, pages 220–235, Cham, 2014. Springer International Publishing.

[92] Bernhard Garn, Ioannis Kapsalis, Dimitris E Simos, and Severin Winkler. On the applicability of combinatorial testing to web application security testing: a case study. In *Proceedings of the 2014 Workshop on Joining AcadeMiA and Industry Contributions to Test Automation and Model-Based Testing*, pages 16–21. ACM, 2014.

[93] Bernhard Garn, Marco Radavelli, Angelo Gargantini, Manuel Leithner, and Dimitris E. Simos. A fault-driven combinatorial process for model evolution in XSS vulnerability detection. In

Franz Wotawa, Gerhard Friedrich, Ingo Pill, Roxane Koitz-Hristov, and Moonis Ali, editors, *Advances and Trends in Artificial Intelligence. From Theory to Practice*, pages 207–215, Cham, 2019. Springer International Publishing.

[94] B. J. Garvin, M. B. Cohen, and M. B. Dwyer. An improved meta-heuristic search for constrained interaction testing. In *1st International Symposium on Search Based Software Engineering*, SSBSE '09, pages 13–22, Washington, DC, USA, May 2009. IEEE Computer Society.

[95] Daochuan Ge, Meng Lin, Yanhua Yang, Ruoxing Zhang, and Qiang Chou. Quantitative analysis of dynamic fault trees using improved sequential binary decision diagrams. *Reliability Engineering & System Safety*, 142:289–299, 2015.

[96] Laleh Sh Ghandehari, Jaganmohan Chandrasekaran, Yu Lei, Raghu Kacker, and D. Richard Kuhn. BEN: A combinatorial testing-based fault localization tool. In *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on*, pages 1–4. IEEE, 2015.

[97] Laleh Sh Ghandehari, Yu Lei, Raghu Kacker, D Richard Rick Kuhn, David Kung, and Tao Xie. A combinatorial testing-based approach to fault localization. *IEEE Transactions on Software Engineering*, 2018.

[98] Laleh Shikh Gholamhossein Ghandehari, Mehra N. Bourazjany, Yu Lei, Raghu N. Kacker, and D. Richard Kuhn. Applying Combinatorial Testing to the Siemens Suite. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, pages 362–371. IEEE, March 2013.

[99] Martin Charles Golumbic and Irith Ben-Arroyo Hartman. *Graph Theory, Combinatorics and Algorithms: Interdisciplinary Applications*. Springer Publishing Company, Incorporated, 2011.

[100] Olga Grinchtein, Bengt Jonsson, and Martin Leucker. Learning of event-recording automata. *Theoretical Computer Science*, 411(47):4029–4054, 2010.

[101] Mats Grindal, Jeff Offutt, and Sten F. Andler. Combination testing strategies: a survey. *Softw. Test, Verif. Reliab*, 15(3):167–199, 2005.

[102] Martin L. Griss. Implementing product-line features by composing aspects. In *Proceedings of the First Conference on Software Product Lines : Experience and Research Directions: Experience and Research Directions*, pages 271–288, Norwell, MA, USA, 2000. Kluwer Academic Publishers.

[103] Jamal El Hachem, Vanea Chiprianov, Ali Babar, and Philippe Aniorte. Towards methodological support for secure architectures of software-intensive systems-of-systems. In *Proceedings of the International Colloquium on Software-intensive Systems-of-Systems at 10th European Conference on Software Architecture*, page 9. ACM, 2016.

[104] M. Harman, Y. Jia, J. Krinke, W. B. Langdon, J. Petke, and Y. Zhang. Search based software engineering for software product line engineering: A survey and directions for future work. In *Proceedings of the 18th International Software Product Line Conference (SPLC '14)*, SPLC '14, pages 5–18, New York, NY, USA, 2014. ACM.

[105] Klaus Havelund, Kim Guldstrand Larsen, and Arne Skou. Formal verification of a power controller using the real-time model checker uppaal. In Joost-Pieter Katoen, editor, *Formal Methods for Real-Time and Probabilistic Systems*, pages 277–298, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[106] Edith Hemaspaandra and Henning Schnoor. Minimization for generalized boolean formulas. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume One*, IJCAI'11, pages 566–571. AAAI Press, 2011.

[107] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, and Yves Le Traon. Towards automated testing and fixing of re-engineered feature models. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 1245–1248. IEEE Press, 2013.

[108] Anders Hessel, Kim Guldstrand Larsen, Marius Mikucionis, Brian Nielsen, Paul Pettersson, and Arne Skou. Testing real-time systems using UPPAAL. In Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors, *Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers*, volume 4949 of *Lecture Notes in Computer Science*, pages 77–117. Springer, 2008.

[109] D.E. Hinkle, W. Wiersma, and S.G. Jurs. *Applied Statistics for the Behavioral Sciences*. Number 663 in Applied Statistics for the Behavioral Sciences. Houghton Mifflin, 2003.

[110] Daniel Hinterreiter, Herbert Prähofer, Lukas Linsbauer, Paul Grünbacher, Florian Reisinger, and Alexander Egyed. Feature-oriented evolution of automation software systems in industrial software ecosystems. In *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, volume 1, pages 107–114. IEEE, 2018.

[111] Gunel Jahangirova, David Clark, Mark Harman, and Paolo Tonella. An Empirical Validation of Oracle Improvement. *IEEE Transactions on Software Engineering*, pages 1–1, 2019.

[112] Gizela Jakubowska and Wojciech Penczek. Modelling and checking timed authentication of security protocols. *Fundamenta Informaticae*, 79(3-4):363–378, 2007.

[113] Gizela Jakubowska, Wojciech Penczek, and Marian Srebrny. Verifying security protocols with timestamps via translation to timed automata. In *Proc. of the International Workshop on Concurrency, Specification and Programming (CS&P'05)*, pages 100–115. Citeseer, 2005.

[114] Rekha Jayaram and R. Krishnan. Approaches to Fault Localization in Combinatorial Testing: A Survey. In *Smart Computing and Informatics*, volume 78, pages 533–540. Springer Singapore, Singapore, 2018.

[115] Yue Jia, Myra B. Cohen, Mark Harman, and Justyna Petke. Learning combinatorial interaction test generation strategies using hyperheuristic search. In *Proceedings of the Int. Conf. on Software Engineering - Volume 1*, ICSE '15, pages 540–550, 2015.

[116] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Trans. Softw. Eng.*, 37(5):649–678, September 2011.

[117] Aleksandra Jovanović, Didier Lime, and Olivier H. Roux. Integer parameter synthesis for real-time systems. *IEEE Transactions on Software Engineering*, 41(5):445–461, 2015.

[118] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 802–811, San Francisco, CA, USA, May 2013. IEEE.

[119] Alexander Knüppel, Thomas Thüm, Stephan Mennicke, Jens Meinicke, and Ina Schaefer. Is there a mismatch between real-world feature models and product-line research? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 291–302, New York, NY, USA, 2017. ACM.

[120] A. Koltuksuz, B. Kulahcioglu, and M. Ozkan. Utilization of timed automata as a verification tool for security protocols. In *2010 Fourth International Conference on Secure Software Integration and Reliability Improvement Companion*, pages 86–93, June 2010.

[121] Roland Kretschmer, Djamel Eddine Khelladi, Andreas Demuth, Roberto E Lopez-Herrejon, and Alexander Egyed. From abstract to concrete repairs of model inconsistencies: An automated

approach. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pages 456–465. IEEE, 2017.

[122] Thomas Krismayer, Rick Rabiser, and Paul GrUnbacher. Mining constraints for event-based monitoring in systems of systems. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 826–831, Urbana, IL, October 2017. IEEE.

[123] John Krogstie. *Model-Based Development and Evolution of Information Systems: A Quality Approach*. Springer Publishing Company, Incorporated, 2012.

[124] D. R. Kuhn and M. J. Reilly. An investigation of the applicability of design of experiments to software testing. In IEEE/Computer Society, editor, *27th NASA/IEEE Software Engineering workshop*, pages 91–95, 2002.

[125] D. Richard Kuhn and Vadim Okun. Pseudo-exhaustive testing for software. In *30th Annual IEEE / NASA Software Engineering Workshop (SEW-30 2006), 25-28 April 2006, Loyola College Graduate Center, Columbia, MD, USA*, volume 0, pages 153–158, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[126] D. Richard Kuhn, Dolores R. Wallace, and Albert M. Gallo. Software fault interactions and implications for software testing. *IEEE Trans. Software Eng*, 30(6):418–421, 2004.

[127] D.R. Kuhn, R.N. Kacker, and Y. Lei. Practical combinatorial testing. Special publication, NIST, 2010.

[128] D.R. Kuhn, R.N. Kacker, and Y. Lei. *Introduction to Combinatorial Testing*. Chapman & Hall/CRC, 2013.

[129] R. Kuhn, R. Kacker, Yu Lei, and J. Hunter. Combinatorial software testing. *Computer*, 42(8):94 –96, aug. 2009.

[130] Jean-Marie Lagniez and Armin Biere. Factoring out assumptions to speed up MUS extraction. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 276–292. Springer, 2013.

[131] Beatriz Pérez Lamancha, Macario Polo, and Mario Piattini. PROW: A pairwise algorithm with constRaints, order and weight. *J. Syst. Softw.*, 99(C):1–19, January 2015.

[132] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 3–13. IEEE, 2012.

[133] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence. Ipog: A general strategy for t-way software testing. In *14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07)*, pages 549–556, March 2007.

[134] Shang-Wei Lin, Étienne André, Yang Liu, Jun Sun, and Jin Song Dong. Learning assumptions for compositional verification of timed systems. *Transactions on Software Engineering*, 40(2):137–153, mar 2014.

[135] H. Lonn and P. Pettersson. Formal verification of a tdma protocol start-up mechanism. In *Proceedings Pacific Rim International Symposium on Fault-Tolerant Systems*, pages 235–242, Dec 1997.

[136] Roberto E. Lopez-Herrejon, Lukas Linsbauer, and Alexander Egyed. A systematic mapping study of search-based software engineering for software product lines. *Information and Software Technology*, 61:33 – 51, 2015.

[137] Roberto E. Lopez-Herrejon, Lukas Linsbauer, José A. Galindo, José A. Parejo, David Benavides, Sergio Segura, and Alexander Egyed. An assessment of search-based techniques for reverse engineering feature models. *Journal of Systems and Software*, 103:353–369, may 2015.

[138] Roberto Erick Lopez-Herrejon, José A. Galindo, David Benavides, Sergio Segura, and Alexander Egyed. Reverse engineering feature models with evolutionary algorithms: An exploratory study. In *Search Based Software Engineering*, pages 168–182. Springer, 2012.

[139] Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Mauro Birattari, and Thomas Stützle. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58, 2016.

[140] C. Lott, A. Jain, and S. Dalal. Modeling requirements for combinatorial software testing. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–7, jul 2005.

[141] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wąsowski. Evolution of the linux kernel variability model. *Software Product Lines: Going Beyond*, pages 136–150, 2010.

[142] Lars Luthmann, Timo Gerecht, Andreas Stephan, Johannes Bürdek, and Malte Lochau. Minimum/maximum delay testing of product lines with unbounded parametric real-time constraints. *Journal of Systems and Software*, 149:535–553, 2019.

[143] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. 1999.

[144] Nuno Macedo, Tiago Guimaraes, and Alcino Cunha. Model repair and transformation with echo. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE'13, pages 694–697, Piscataway, NJ, USA, nov 2013. IEEE.

[145] Nicolas Markey. Robustness in real-time systems. *SIES*, 11:28–34, 2011.

[146] Joao Marques-Silva. Practical applications of Boolean Satisfiability. In *2008 9th International Workshop on Discrete Event Systems*, pages 74–80, Goteborg, Sweden, 2008. IEEE.

[147] Conrado Martínez, Lucia Moura, Daniel Panario, and Brett Stevens. Locating Errors Using ELAs, Covering Arrays, and Adaptive Testing Algorithms. *SIAM Journal on Discrete Mathematics*, 23(4):1776–1799, January 2010.

[148] E. J. McCluskey. Minimization of boolean functions*. *Bell System Technical Journal*, 35(6):1417–1444, 1956.

[149] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. *Mastering Software Variability with FeatureIDE*. Springer, 2017.

[150] William Melicher, Anupam Das, Mahmood Sharif, Lujo Bauer, and Limin Jia. Riding out DOMsday: Towards Detecting and Preventing DOM Cross-Site Scripting. In *Proceedings of Network and Distributed System Security Symposium*. Internet Society, 2018.

[151] Marcilio Mendonca, Moises Branco, and Donald Cowan. Splot: software product lines online tools. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 761–762. ACM, 2009.

[152] Ivanka Menken. *SaaS - The Complete Cornerstone Guide to Software As a Service Best Practices Concepts, Terms, and Techniques for Successfully Planning, Implementing and Managing SaaS Solutions*. Emereo Pty Ltd, London, UK, UK, 2008.

[153] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. Mining configuration constraints: static analyses and empirical results. In Pankaj Jalote, Lionel C. Briand, and André van der Hoek, editors, *ICSE*, ICSE 2014, pages 140–151, New York, NY, USA, 2014. ACM.

[154] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. SemFix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 772–781. IEEE, may 2013.

[155] Changhai Nie and Hareton Leung. The Minimal Failure-Causing Schema of Combinatorial Testing. *ACM Transactions on Software Engineering and Methodology*, 20(4):1–38, September 2011.

[156] Changhai Nie and Hareton Leung. A survey of combinatorial testing. *ACM Comput. Surv*, 43(2):11, 2011.

[157] Changhai Nie, Baowen Xu, Liang Shi, and Ziyuan Wang. A new heuristic for test suite generation for pair-wise testing. In Kang Zhang, George Spanoudakis, and Giuseppe Visaggio, editors, *SEKE*, pages 517–521, 2006.

[158] X. Niu, N. Changhai, Y. Lei, H. K. N. Leung, and X. Wang. Identifying failure-causing schemas in the presence of multiple faults. *IEEE Transactions on Software Engineering*, pages 1–1, 2018.

[159] X. Niu, N. Changhai, H. K. N. Leung, Y. Lei, X. Wang, J. Xu, and Y. Wang. An interleaving approach to combinatorial testing and failure-inducing interaction identification. *IEEE Transactions on Software Engineering*, pages 1–1, 2018.

[160] Sebastian Oster. Feature Model-based Software Product Line Testing. page 236. PhD Thesis.

[161] Pairwise web site. http://www.pairwise.org/.

[162] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Mutation testing advances: An analysis and survey. In *Advances in Computers*, Advances in Computers. Elsevier, 2018.

[163] Pavel Parizek, Frantisek Plasil, and Jan Kofron. Model checking of software components: Combining java pathfinder and behavior protocol model checker. In *2006 30th Annual IEEE/NASA Software Engineering Workshop*, pages 133–141. IEEE, 2006.

[164] Leonardo Passos, Rodrigo Queiroz, Mukelabai Mukelabai, Thorsten Berger, Sven Apel, Krzysztof Czarnecki, and Jesus Padilla. A study of feature scattering in the linux kernel. *IEEE Transactions on Software Engineering*, 2018.

[165] Mateusz Pawlik and Nikolaus Augsten. Efficient Computation of the Tree Edit Distance. *ACM Transactions on Database Systems*, 40(1):1–40, March 2015.

[166] Mateusz Pawlik and Nikolaus Augsten. Tree edit distance: Robust and memory-efficient. *Information Systems*, 56:157–173, March 2016.

[167] Gilles Perrouin, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves Le Traon. Automated and scalable t-wise test case generation strategies for software product lines. In *Proc. of the International Conference on Software Testing (ICST)*, pages 459–468, Paris, France, April 2010. IEEE.

[168] Justyna Petke, Myra B. Cohen, Mark Harman, and Shin Yoo. Practical combinatorial interaction testing: Empirical findings on efficiency and early fault detection. *IEEE Trans. Software Eng.*, 41(9):901–924, 2015.

[169] Justyna Petke, Shin Yoo, Myra B. Cohen, and Mark Harman. Efficiency and early fault detection with lower and higher strength combinatorial interaction testing. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 26–36, 2013.

[170] Andreas Pleuss, Goetz Botterweck, Deepak Dhungana, Andreas Polzer, and Stefan Kowalewski. Model-driven support for product line evolution on feature level. *Journal of Systems and Software*, 85(10):2261–2274, 2012.

[171] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco Documentation*. TASC - LS2N CNRS UMR 6241, COSLING S.A.S., 2017.

[172] M. Radavelli. Using testing to repair models. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 489–491, April 2019.

[173] Marco Radavelli. Using software testing to repair models. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, pages 1253–1255, New York, NY, USA, 2019. ACM.

[174] Alexander Reder and Alexander Egyed. Computing repair trees for resolving inconsistencies in design models. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering - ASE 2012*, page 220, Essen, Germany, 2012. ACM Press.

[175] Iris Reinhartz-Berger, Kathrin Figl, and Øystein Haugen. Comprehending feature models expressed in CVL. In Juergen Dingel, Wolfram Schulte, Isidro Ramos, Silvia Abrahão, and Emilio Insfran, editors, *Model-Driven Engineering Languages and Systems*, pages 501–517, Cham, 2014. Springer International Publishing.

[176] Dominik Rost, Matthias Naab, Crescencio Lima, and Christina von Flach Garcia Chavez. Software architecture documentation for developers: a survey. In *European Conference on Software Architecture*, pages 72–88. Springer, 2013.

[177] Julia Rubin and Marsha Chechik. Quality of merge-refactorings for product lines. In *International Conference on Fundamental Approaches to Software Engineering*, pages 83–98. Springer, 2013.

[178] Richard L. Rudell. Multiple-valued logic minimization for pla synthesis. Technical Report UCB/ERL M86/65, EECS Department, University of California, Berkeley, 1986.

[179] Alcemir Rodrigues Santos, Raphael Pereira de Oliveira, and Eduardo Santana de Almeida. Strategies for consistency checking on software product lines: A mapping study. In *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*, EASE '15, pages 5:1–5:14, New York, NY, USA, 2015. ACM.

[180] Itai Segall, Rachel Tzoref-Brill, and Eitan Farchi. Using binary decision diagrams for combinatorial test design. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 254–264, New York, NY, USA, 2011. ACM.

[181] Daniel Sheridan. The optimality of a fast cnf conversion and its use with sat. *SAT*, 2, 2004.

[182] George B Sherwood. Embedded functions for constraints and variable strength in combinatorial testing. In *Software Testing, Verification and Validation Workshops (ICSTW), 2016 IEEE Ninth International Conference on*, pages 65–74. IEEE, 2016.

[183] Jiangfan Shi, Myra B. Cohen, and Matthew B. Dwyer. Integration testing of software product lines using compositional symbolic execution. In *Fundamental Approaches to Software Engineering - 15th International Conference, FASE 2012*, pages 270–284, 2012.

[184] Liang Shi, Changhai Nie, and Baowen Xu. A software debugging method based on pairwise testing. In *Proc. of the 5th Int. Conference on Computational Science*, ICCS'05, pages 1088–1091, Berlin, Heidelberg, 2005. Springer-Verlag.

[185] Daisuke Shimbara and Øystein Haugen. *Generating Configurations for System Testing with Common Variability Language*, pages 221–237. Springer International Publishing, Cham, 2015.

[186] D. E. Simos, R. Kuhn, A. G. Voyiatzis, and R. Kacker. Combinatorial methods in security testing. *IEEE Computer*, 49:40–43, 2016.

[187] Dimitris E. Simos, Kristoffer Kleine, Laleh Shikh Gholamhossein Ghandehari, Bernhard Garn, and Yu Lei. A Combinatorial Approach to Analyzing Cross-Site Scripting (XSS) Vulnerabilities in Web Application Security Testing. In *Testing Software and Systems*. Springer, 2016.

[188] B. D. Smith, M. S. Feather, and N. Muscettola. Challenges and methods in validating the remote agent planner. In CO Breckenridge, editor, *Proceedings of the Fifth International conference on Artificial Intelligence Planning Systems (AIPS)*, 2000.

[189] Kenneth Sörensen. Metaheuristics – the metaphor exposed. *International Transactions in Operational Research*, 22(1):3–18, feb 2013.

[190] Jan Springintveld, Frits Vaandrager, and Pedro R. D'Argenio. Testing timed automata. *Theoretical Computer Science*, 254(1-2):225–257, March 2001.

[191] Matt Staats, Michael W. Whalen, and Mats P.E. Heimdahl. Programs, tests, and oracles: the foundations of testing revisited. In *Proceeding of the 33rd international conference on Software engineering - ICSE '11*, page 391, Waikiki, Honolulu, HI, USA, 2011. ACM Press.

[192] K. C. Tai and Y. Lie. A test generation strategy for pairwise testing. *IEEE Trans. Softw. Eng.*, 28(1):109–111, 2002.

[193] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. Feature consistency in compile-time-configurable system software: Facing the linux 10,000 feature problem. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 47–60, New York, NY, USA, 2011. ACM.

[194] Paul Temple, Mathieu Acher, Battista Biggio, Jean-Marc Jézéquel, and Fabio Roli. Towards adversarial configurations for software product lines. *CoRR*, abs/1805.12021, 2018.

[195] Paul Temple, Mathieu Acher, Jean-Marc Jezequel, and Olivier Barais. Learning Contextual-Variability Models. *IEEE Software*, 34(6):64–70, November 2017.

[196] Paul Temple, José Angel Galindo Duarte, Mathieu Acher, and Jean-Marc Jézéquel. Using machine learning to infer constraints for product lines. In *Proceedings of the 20th International Systems and Software Product Line Conference*, SPLC '16, pages 209–218, New York, NY, USA, 2016. ACM.

[197] TestCover tool. http://www.testcover.com/.

[198] Thomas Thüm, Leopoldo Teixeira, Klaus Schmid, Eric Walkingshaw, Mukelabai Mukelabai, Mahsa Varshosaz, Goetz Botterweck, Ina Schaefer, and Timo Kehrer. Towards efficient analysis of variation in time and space. In *Proceedings of the 23rd International Systems and Software Product Line Conference volume B - SPLC '19*, pages 1–8, Paris, France, 2019. ACM Press.

[199] Thomas Thüm, Don Batory, and Christian Kastner. Reasoning about edits to feature models. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 254–264, Washington, DC, USA, 2009. IEEE Computer Society.

[200] Thomas Thüm, Christian Kastner, Sebastian Erdweg, and Norbert Siegmund. Abstract features in feature modeling. In *Proceedings of the 2011 15th International Software Product Line Conference*, SPLC '11, pages 191–200, Washington, DC, USA, 2011. IEEE Computer Society.

[201] John B. Tran and Richard C. Holt. Forward and reverse repair of software architecture. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 12–. IBM Press, 1999.

[202] S. Urli, A. Bergel, M. Blay-Fornarino, P. Collet, and S. Mosser. A visual support for decomposing complex feature models. In *2015 IEEE 3rd Working Conference on Software Visualization (VISSOFT)*, pages 76–85, Sept 2015.

[203] Macario Polo Usaola and Beatriz Pérez Lamancha. A framework and a web implementation for combinatorial testing. Technical report, Informe técnico, University of Castilla-La Mancha, 2010.

[204] Markus Voelter. Using domain specific languages for product line engineering. In *Proceedings of the 13th International Software Product Line Conference*, SPLC '09, pages 329–329, Pittsburgh, PA, USA, 2009. Carnegie Mellon University.

[205] Alexander von Rhein, Alexander Grebhahn, Sven Apel, Norbert Siegmund, Dirk Beyer, and Thorsten Berger. Presence-condition simplification in highly configurable systems. In *Pro-*

ceedings of the 37th International Conference on Software Engineering - Volume 1, ICSE '15, pages 178–188, Piscataway, NJ, USA, 2015. IEEE Press.

[206] Vytautas Štuikys, Renata Burbaitė, Kristina Bespalova, and Giedrius Ziberkas. Model-driven processes and tools to design robot-based generative learning objects for computer science education. *Science of Computer Programming*, 129:48–71, 2016. Special issue on eLearning Software Architectures.

[207] Ting Wang, Jun Sun, Yang Liu, Xinyu Wang, and Shanping Li. Are timed automata bad for a specification language? Language inclusion checking for timed automata. In Erika Ábrahám and Klaus Havelund, editors, *TACAS*, volume 8413 of *Lecture Notes in Computer Science*, pages 310–325. Springer, 2014.

[208] Ziyuan Wang, Baowen Xu, Lin Chen, and Lei Xu. Adaptive Interaction Fault Location Based on Combinatorial Testing. In *2010 10th International Conference on Quality Software*, pages 495–502. IEEE, July 2010.

[209] E. J. Weyuker. On Testing Non-Testable Programs. *The Computer Journal*, 25(4):465–470, November 1982.

[210] Claes Wohlin, Per Runeson, Martin Hst, Magnus C. Ohlsson, Bjrn Regnell, and Anders Wessln. *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated, 2012.

[211] Xtext. http://www.eclipse.org/xtext/.

[212] Jifeng Xuan, Matias Martinez, Favio DeMarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering*, 43(1):34–55, jan 2017.

[213] Akihisa Yamada, Armin Biere, Cyrille Artho, Takashi Kitamura, and Eun-Hye Choi. Greedy combinatorial test case generation using unsatisfiable cores. pages 614–624. ACM Press, 2016.

[214] Cemal Yilmaz, Myra B. Cohen, and Adam A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *IEEE Trans. Software Eng*, 32(1):20–34, 2006.

[215] Cemal Yilmaz, Emine Dumlu, Myra B. Cohen, and Adam A. Porter. Reducing masking effects in combinatorial interaction testing: A feedback driven adaptive approach. *IEEE Trans. Software Eng.*, 40(1):43–66, 2014.

[216] Linbin Yu, Yu Lei, Raghu N Kacker, and D Richard Kuhn. Acts: A combinatorial test generation tool. In *IEEE Int. Conf. on Software Testing, Verification and Validation*, 2013.

[217] Ruediger Zarnekow and Walter Brenner. Distribution of cost over the application lifecycle-a multi-case study. *ECIS 2005 Proceedings*, page 26, 2005.

[218] Jian Zhang, Feifei Ma, and Zhiqiang Zhang. Faulty interaction identification via constraint solving and optimization. In Alessandro Cimatti and Roberto Sebastiani, editors, *Theory and Applications of Satisfiability Testing – SAT 2012*, pages 186–199, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[219] Zhiqiang Zhang and Jian Zhang. Characterizing failure-causing parameter interactions by adaptive testing. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 331–341, New York, NY, USA, 2011. ACM.