



CITLAB, a laboratory for combinatorial interaction testing

Paolo Vavassori

Ph.D. course in Mechatronics, Information Technology,
New Technologies and Mathematical Methods
XXVIII Cycle

Advisor: Prof. Angelo Gargantini

Università degli Studi di Bergamo, Italy
Department of Management, Information and
Production Engineering

February, 2016

I dedicated my thesis to Professor Angelo Michele Gargantini, my Advisor who believed I could do a Ph.D.. I would like to acknowledge the great support and respect I received during my doctoral research from my family and from my colleagues of "Office 305". I also thank Anna, who tries every day, to mitigate my bad temper helping me to achieve new goals.

Paolo Vavassori

“On a given day, a given circumstance, you think you have a limit. And you then go for this limit and you touch this limit, and you think, ‘Okay, this is the limit.’ As soon as you touch this limit, something happens and you suddenly can go a little bit further. With your mind power, your determination, your instinct, and the experience as well, you can fly very high.”

— Ayrton Senna

Abstract

Although the research community around combinatorial interaction testing has been very active for several years, it has failed to find common solutions on some issues. First of all, there is not a common abstract nor concrete language to express combinatorial problems. Combinatorial testing generator tools are strongly decoupled making difficult their interoperability and the exchange of models and data. We propose a new framework for Combinatorial Interaction Testing. Our project starts from the definition of a specific language for combinatorial problems. It features and formally defines the concepts of parameters and types, constraints, seeds, and test goals. The language is defined by means of XTEXT, a framework for the definition of domain-specific languages. XTEXT is used to derive a powerful editor integrated with eclipse and with all the expected features of a modern editor. Eclipse plugins architecture is used to make our framework extensible. A practitioner can easily add test generators, importers, and exporters as plugins. This new framework, CITLAB, aims to be a Laboratory for Combinatorial Interaction Testing tailored for the practitioners and the researchers who want to devise new solutions and compare them with existing ones. We present the effectiveness of this framework through the description of its use in several research projects. The research activities presented in this thesis are focused on the improvement of the state of art of Combinatorial and on the novel uses of this testing technique. Beside the description of the development of our framework, this thesis deals with many aspect of CIT from the research about new algorithm for Test Suite generation to its application for testing of the SPLs.

Contents

Abstract	iv
1 Introduction	1
1.1 Publications arising from this thesis	4
1.2 Acknowledgments	4
2 Combinatorial Interaction Testing	5
2.1 Combinatorial testing time-line	6
2.2 A small example	6
2.3 Research area	8
3 CITLAB Language	11
3.1 Language definition	11
3.1.1 Parameters and their types	11
3.1.2 Constraints	15
3.1.3 Seeds	18
3.1.4 Test Goals	18
4 Xtext implementation	21
4.1 Language and its editor with Xtext	22
4.1.1 DSL development	22
5 Framework architecture	29
5.1 Defining the extension points	29
5.1.1 Test generation plugins	30
5.1.2 Translation to and from other notations and tools	34
5.1.3 Test suite exporters	35
5.1.4 Validators	35
6 MDD generator	37
6.1 Introduction	37
6.2 Multivalued Decision Diagram	39

6.3	Using MDD for CIT	40
6.4	An MDD-based algorithm for CCIT	43
6.4.1	Optimization: Weighting compatibility	44
6.4.2	Optimization: Repetitions	46
6.5	Experiments	47
6.5.1	Optimal threshold value	48
6.5.2	Using compatibility	48
6.5.3	Number of repetitions	50
6.5.4	Comparison with other tools in CITLAB	51
6.5.5	Threats to validity	53
6.6	Related work	53
6.7	Future work and Conclusions	55
7	CIT Validation process	57
7.1	Background work	58
7.2	Notation	59
7.2.1	Using Logics and SAT/SMT solvers for CIT problems	59
7.2.2	Desired properties of combinatorial models and tests	61
7.3	Validation of CIT models	62
7.3.1	Inconsistent constraints	62
7.3.2	Constraints Vacuity	64
7.3.3	Useless parameter values and useless parameters	66
7.4	Validation of CIT test suites	67
7.4.1	Test Suite Correctness	67
7.4.2	Test Suite Minimality	69
7.5	Experimental results	72
7.5.1	Consistency of constraints	72
7.5.2	Vacuity detection	74
7.5.3	Useless parameter values and useless parameters	74
7.5.4	Test suite validation	75
7.6	Implementation in CITLAB	76
7.7	Conclusions	77
8	Feature models testing in CITLAB	79
8.1	Background	79
8.1.1	Feature Modeling frameworks	81
8.1.2	Feature Model semantics	81
8.2	Translation from FM to CitLab	83
8.2.1	Representation of every feature	83
8.2.2	Adding implicit constraints	84
8.2.3	Cross-tree constraints	84

8.2.4	Extra testing requirements	88
8.3	Simplification Process	89
8.3.1	Constraint simplification	90
8.3.2	Parameter simplification	90
8.4	Experiments	92
8.4.1	Testing the correctness of the transformation	92
8.4.2	Effect of the simplification over the parameters and the constraints	93
8.4.3	Comparison with approaches using Boolean variables	93
8.4.4	Test generation	96
8.5	Related Work	101
8.6	Conclusion and Future Work	103
9	Using Decision Trees for Algorithm Selection	105
9.1	Algorithm Selection problem	106
9.2	The importance of the " <i>right</i> " algorithm	107
9.2.1	Aiding the selection	107
9.3	How to solve algorithm selection problem	108
9.4	Background	108
9.4.1	Defining the "best" combinatorial generation algorithm	108
9.4.2	CITLAB as a framework for benchmark comparison .	109
9.4.3	Decision Tree	110
9.4.4	Tools for Data Mining	111
9.5	Process of Building the Decision Tree	111
9.5.1	Problem Understanding	112
9.5.2	Data Understanding and Preparation	113
9.5.3	Modeling	115
9.5.4	Evaluation	118
9.5.5	Deployment	120
9.6	Experiments	120
9.7	Threats to validity	127
9.8	Conclusions	128
	10 Results achieved by CITLAB	131

List of Figures

2.1	History of Combinatorial testing	7
2.2	Testing procedure for CIT	10
3.1	A smartphone example	12
4.1	XTEXT fragments and artifacts	23
4.2	Validation of seeds	24
4.3	Content assist	24
4.4	A meta model for Tests Suites	25
4.5	UML Class Diagram of the grammar of CITLAB	26
4.6	UML Class Diagram of CITLAB logic constructs	27
5.1	TestGenerator extension point and corresponding extension	30
5.2	UML Class Diagram of some implemented extensions	32
5.3	Test Suite Ecore Class Diagram	33
5.4	A fragment of the XTEXT exporter to CASA	34
6.1	MDD for the combinatorial problem of Listing 6.1	41
6.2	The MDD representing the pair (eV = true, fC = 2MP)	42
6.3	M_{VS} for the phone with the constraints	43
6.4	an MDD representing a single test case	43
6.5	CITLAB plugin classes	47
6.6	Test suite <i>size</i> and <i>time</i> depending on the threshold	48
6.7	Greedy vs Compatibility comparison with optimization	49
6.8	Tuple coverage rate for b_12 with optimization	49
6.9	Test suite <i>size</i> and <i>time</i> depending on the repetitions settings ($repeat_{\min}$ $repeat_{\max}$ $repeat_{\text{better}}$).	50
6.10	Test suite size <i>size</i> and time <i>time</i> depending on the number of repetitions	51
6.11	Comparison with ACTS and CASA	51
6.12	Number of models that present the minimum cost for each generator for $time_{\text{test}}$ from 0.01 to 5000 secs.	52

7.1	Example of useless parameter and useless parameter value . . .	67
7.2	TestSuiteValidator UML	76
7.3	Model Validator UML	77
8.1	Examples of Feature model notations	80
8.2	A two alternative model	83
8.3	A small complete example	88
8.4	FeatureIDE importer plugin	92
8.5	Reduction of the number of parameters	94
8.6	Reduction of the number of constraints	95
8.7	Comparison of number of CitLab Parameters vs Boolean variables	97
8.8	Comparison of CitLab constraints vs SPLOT constraints	98
8.9	Variability variation due to simplification process	99
8.10	Variability comparison with SPLOT	100
8.11	Implementation in CITLAB	103
9.1	A decision tree representing the exit selection in a crowded parking area.	110
9.2	Training set attributes and characteristics	114
9.3	Pearson chart	116
9.4	FBT: feature-based decision tree	119
9.5	CBT: cost-based decision tree	119
9.6	Test generator distribution of the optimum predictor	121
9.7	Generation tools distribution using FBT	123
9.8	Generation tools distribution using CBT	124
9.9	Performance comparison of Predictors versus the use of a fixed tool using the optimum as measure of comparison	126
9.10	Performance comparison of predictors versus random selection of generators using the optimum as measure of comparison	127

List of Tables

2.1	A minimal SmartPhone product line	7
2.2	A test suite for pairwise coverage of the SmartPhone example	8
5.1	Extension points defined by CITLAB	31
6.1	Characteristics of the CCIT benchmarks.	47
7.1	Vacuous constraints	73
7.2	Useless parameter values and useless parameters	75
7.3	Test suite validation	76
8.1	Conventional translation	82
8.2	Representing features and setting <i>isChosen</i> function	85
8.3	Constraints to be added	86
8.4	Translation to CITLAB model for model of Fig. 8.3	87
8.5	Parameter simplification	91
8.6	Pairwise with ACTS (time in seconds)	101
8.7	Pairwise with CASA (time in seconds)	101
9.1	FBT confusion matrix for the test set	122
9.2	CBT confusion matrix for the validation set	123
9.3	Total Mean Cost varying single test cost	125

1

Introduction

Combinatorial interaction testing (CIT) has been an active area of research for many years. In a recent survey [63] Nie and Leung count more than 12 research groups that actively work on CIT area and many other groups and tools are missing in the count. In a previous survey, Grindal et al. [40] presented 16 different combination strategies, covering more than 40 papers. There are several web sites listing tools and approaches (like [66]), and publishing benchmarks and evaluations of tools and algorithms (like [27]). Being each of these tools the outcome of independent research and development processes, every one has its own user interface (some graphical, other textual), its own syntax, its own algorithms, and its own benchmarks (if any). Despite they all are designed to tackle the very same tasks, as a matter of fact, there is not a common *abstract meta-model* to represent combinatorial problems with a precise semantics for parameters, values, their constraints, and related concepts, nor exists a common *modeling syntax* or model exchange format between tools.

The lack of a common syntax and semantic framework for CIT makes harder the research in this area w.r.t. the following issues:

- The *comparison* among tools and approaches, an activity very useful and used in research literature, is instead quite unreliable since every user must remodel in its own language and tool the case studies taken from another tool or from the literature, with possible errors and misunderstandings. If the researchers could exchange examples using such

common syntax (like XML, or textual, or graphical), the comparison of techniques and approaches would be facilitated and more technically sound.

- While conceptually modeling an abstract CIT task, a research group may use a term with a meaning, while other groups use the same term with a slightly different meaning (for example *seed* or *partial test*).
- Limited assistance in writing the models: very often generation tools do not offer any editing capabilities (only a grammar and a parser) and are rarely integrated in any IDE for programming or design. Very often the formats accepted by algorithms and tools are quite hard to understand¹.
- All the CIT generation tools are strongly decoupled making difficult to switch the use from one tool to another and also difficult the reuse of information (e.g. custom settings) and data already inserted and available in one tool.

This situation is also an obstacle for practitioners from fully advantage from such many different CIT generation techniques available, other than slowing down the research in this area.

Sometimes, designers may prefer a tool or a technique because it provides an usable graphical or a web interface instead of searching for the best tool that suites their needs. For instance, one the most used tools ACTS [1], has a very nice graphical interface regardless the fact that the generation methods it supports (IPOG [57] and variants) may be not suitable for the design of particular combinatorial test suites since, for instance, its support for constraints is not as powerful as in others.

Similar difficulties rise for researchers willing to devise a new CIT technique and compare it with existing ones. In order to experiment a new test generation algorithm, a researcher should define a proper grammar and a parser, develop the libraries to manipulate the model data, and translate the benchmarks found in literature into the newly defined language. These activities can be error-prone and quite time consuming without adding any actual contribution to the real problem of generating “better” combinatorial tests.

In this thesis, we present CITLAB, a laboratory for combinatorial testing that tries to address all the aforementioned issues. CITLAB features:

¹Consider for instance one of the best tools for Constrained CIT, CASA [21]. CASA accepts only constraints written as a conjunction of disjunctions over the symbols (CNF), in a quite difficult format to write for humans.

- A rich abstract language with a precise *formal semantics* for specifying combinatorial problems.
- A concrete syntax with a well-defined *grammar* that allows practitioners to write models and researchers to share examples and benchmarks written in a "common" notation. Besides the concrete syntax given in XTEXT, CITLAB provides also an ANTLR grammar and an XMI interchange format for CIT models.
- A framework based on the Eclipse Modeling Framework (EMF) which provides tools and run-time support to (automatically) produce a set of Java classes for combinatorial models, along with a set of adapter classes and utility libraries that enable manipulating combinatorial problems in Java application using simple APIs. This allows developers to access combinatorial models inside their programs and tools.
- An editor integrated in the Eclipse IDE for editing combinatorial problems. The editor provides users with all the expected features in a modern programming environment like syntax highlighting, code completion, run-time error checking, quick fixes, and outline view.
- A simple EMF meta-model also for combinatorial test suites.
- A rich collection of Java utility classes and methods, specifically developed for combinatorial problems in CITLAB, which can be reused for manipulating combinatorial models and test suites. For instance, CITLAB provides utility methods for generating all the test requirements for a combinatorial coverage of strength t , a set of methods to check if a test suite satisfies all the requirements, and a set of methods for semantic validation of models and test suites.
- A framework for introducing new test generation algorithms which can be added to CITLAB as plugins. This allows researchers to develop new generation techniques and plug them in the framework without the burden of defining a grammar, a parser, an abstract syntax tree visitor, and so on.
- A framework for introducing code translators for importing and exporting models and tests to other notations based on Model to Text (M2T) or Model to Model (M2M) transformations. This could facilitate the use of CITLAB language as language for exchanging models and benchmarks.

1.1 Publications arising from this thesis

This Section presents a list of the published and submitted papers that have arisen from the work in the thesis.

Papers in Proceedings of International Conferences and Workshops:

- Gargantini, A.; Vavassori, Paolo, "CITLAB: A Laboratory for Combinatorial Interaction Testing," Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference, pp.559,568, 17-21 April 2012 doi: 10.1109/ICST.2012.141
- Calvagna, A.; Gargantini, A.; Vavassori, P., "Combinatorial Interaction Testing with CITLAB," Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference, pp.376,382, 18-22 March 2013 doi: 10.1109/ICST.2013.53
- Calvagna, A.; Gargantini, A.; Vavassori, P., "Combinatorial Testing for Feature Models Using CitLab," Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference, pp.338,347, 18-22 March 2013 doi: 10.1109/ICSTW.2013.45
- Arcaini, P.; Gargantini, A.; Vavassori, P., "Validation of Models and Tests for Constrained Combinatorial Interaction Testing," Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference, pp.98,107, March 31 2014-April 4 2014 doi: 10.1109/ICSTW.2014.5
- Gargantini, Angelo and Vavassori, Paolo "Efficient Combinatorial Test Generation Based on Multivalued Decision Diagrams," Hardware and Software: Verification and Testing Springer International Publishing, pp.220,235, 2014 doi: 10.1007/978-3-319-13338-6_1
- Gargantini, A.; Vavassori, P., "Using decision trees to aid algorithm selection in combinatorial interaction tests generation," Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference, pp.1,10, 13-17 April 2015 doi: 10.1109/ICSTW.2015.7107442

1.2 Acknowledgments

The research leading to the results documented in this thesis was supervised by the Prof. Angelo Gargantini (Università degli Studi di Bergamo) and Prof. Paolo Arcaini (Charles University in Prague).

2

Combinatorial Interaction Testing

Combinatorial Interaction Testing (CIT) systematically 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-constraint the costs of testing while still having a testing process driven by an effective and exhaustive coverage metric [23, 53]. 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 [29, 78]. Dunietz *et al.* [31] 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 [14] reported 93% code coverage from applying pairwise testing of a large commercial software system, and many CIT tools (see [66] for an up to date listing) and techniques have already been developed [30, 40, 53] and are currently applied in practice [10, 52, 77]. 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.

2.1 Combinatorial testing time-line

Combinatorial (t-way) testing may be regarded as an adaptation of design of experiment methods for testing software systems because in both cases, information about the SUT is gained by exercising it and the test plan satisfies relevant combinatorial properties. In the 1930s, Ronald Fisher developed the Design of Experiments. Rao introduced orthogonal arrays in the 1940s.

From the 1950s, Genichi Taguchi developed a methodology for applying statistics to improve quality in manufacturing and established the Quality Engineering. His methodology got a great reputation and was named "Taguchi method" in the United States. Shortly after, software QA engineers at Fujitsu started to apply the Design of Experiments to *Software Testing* in 1983, and they published some papers in 1984 and 1987. In the late 1980s, a few case studies were reported from NEC and IBM Japan, however, in 1989, Fujitsu introduced this technique to AT&T. After that, OATS, CATS and AETG algorithms were developed by the companies of AT&T and Bellcore constellation. From the mid 1990s, combinatorial testing began to spread in the US.

In 1995, Dr. Phadke's paper was translated into Japanese and printed in the journal of Quality engineering. Some case studies were reported in the community of quality engineering in the late 1990s. However, combinatorial testing got much attention again after HAYST method was announced by Fuji Xerox in 2004.

The algorithms reported in this thesis have been developed after 2004. This evolution process is described in Fig. 2.1.

2.2 A small example

There are basically two approaches to combinatorial testing use combinations of configuration parameter values, or combinations of input parameter values. In the first case, we select combinations of values of configurable parameters. For example, a server might be tested by setting up all 4-way combinations of configuration parameters such as number of simultaneous connections allowed, memory, OS, database size, etc., with the same test suite run against each configuration. The tests may have been constructed using any methodology, not necessarily combinatorial coverage. The combinatorial aspect of this approach is in achieving combinatorial coverage of configuration parameter values. (Note, the term variable is often used interchangeably with parameter to refer to inputs to a function.) In the second approach, we select combinations of input data values, which then become

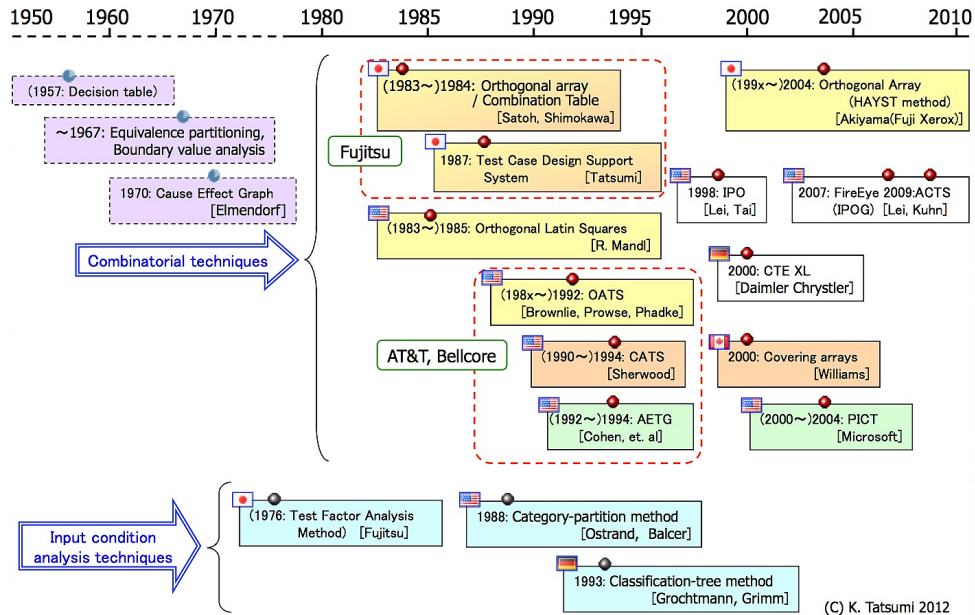


Figure 2.1: History of Combinatorial testing

part of complete test cases, creating a test suite for the application. In this case combinatorial coverage of input data values is required for tests constructed.

Combinatorial testing can be applied to a wide variety of problems: highly-configurable software systems, software product lines which define a family of software, hardware systems, and so on.

As an example, Table 2.1 reports the input domain model of a simple smart-phone product line containing only three parameters: the `display` can have 16 or 8 million colors or be in black and white (BW), the `frontCamera` can have 1 or 2 mega-pixels (1MP and 2MP) or not be present (NOC). The phone can also have an `emailViewer`. We will use this simple example throughout the chapter and to explain our approach.

Table 2.1: A minimal SmartPhone product line

display	frontCamera	emailViewer
16MC	2MP	true
8MC	1MP	false
BW	NOC	

While testing of all the possible configurations for the phone would require $3 \cdot 3 \cdot 2 = 18$ tests, pairwise coverage can be obtained by the optimal test

Table 2.2: A test suite for pairwise coverage of the SmartPhone example

	display	frontCamera	emailViewer
1	16MC	1MP	false
2	16MC	2MP	true
3	16MC	NOC	false
4	8MC	2MP	false
5	8MC	1MP	true
6	8MC	NOC	true
7	BW	2MP	false
8	BW	1MP	true
9	BW	NOC	false

suite containing only 9 tests shown in Table 2.2.

2.3 Research area

Changai Nie and Hareton Leung [63], in a recent survey propose eight categories for the classification of the CIT research projects:

1. Modeling (Model): Studies on identifying the parameters, values, and the interrelations of parameters of SUT.
2. Test case generation (Gen): Studies on generating a small test suite effectively.
3. Constraints (Constr.): Studies on avoiding invalid test cases in the test suite generation.
4. Failure characterization and diagnosis (Fault): Studies on fixing the detected faults.
5. Improvement of testing procedures and the application of CT (App.): Studies on practical testing procedure for CT and reporting the results of the CT application.
6. Priority of test cases (Prior.): Studies on the order of test execution to detect faults as early as possible in the most economical way.
7. Metric (Metric): Studies on measuring the combination coverage of CT and the effectiveness of fault detection.

-
8. Evaluation (Eval.): Studies on the degree to which CT contributes to the improvement of software quality

This thesis tries to improve some aspect of CIT that involve the categories 1, 2, 3 and 8.

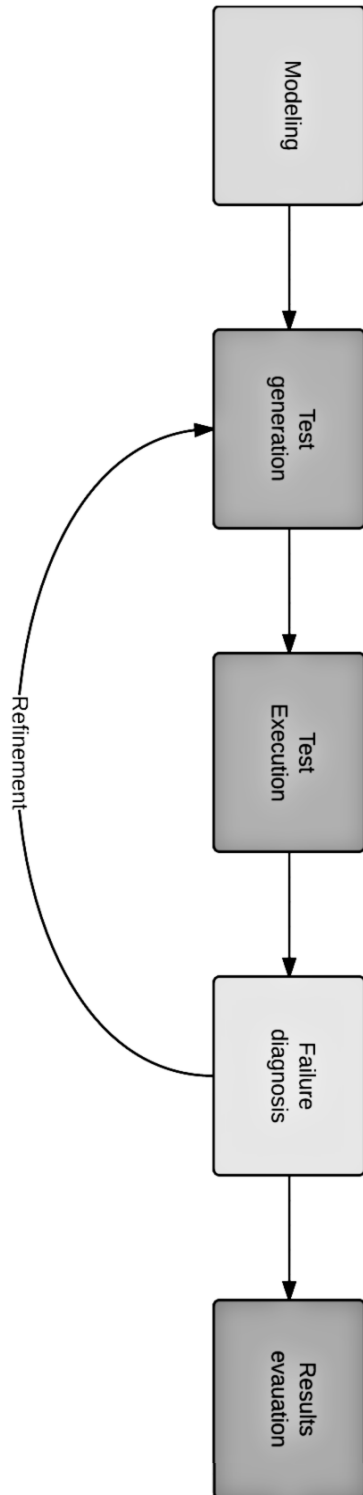


Figure 2.2: Testing procedure for CIT

3

CITLAB Language

The first goal of our project is the definition of a domain specific language for combinatorial problems. We identify four parts we want to model: the parameters and their types, the constraints among them, the seeds to be included in the final test suites, and further test goals defined by the user. Fig. 3.1 shows an example of combinatorial model, called **Phone**, a modified version of an example given in [25]. Constraints, seeds, and test goals are removed by the example for simplicity and they will be discussed later in the section.

3.1 Language definition

3.1.1 Parameters and their types

A model for a combinatorial problem consists in several parameters (at least 2) which can take values in their domain. To describe a combinatorial problem would be sufficient to specify the *number* of variables and their *cardinality*. Many papers, like [46], simply use the exponential symbolic notation x^y to model y parameters each of which can take x values. For heterogeneous alphabets, the notation is extended as $x_1^{y_1} x_2^{y_2} \dots x_n^{y_n}$ to model y_i parameters that take x_i values with $i = 1 \dots n$. A CITLAB model consists in six parts: *Definitions*, *Types*, *Parameters*, *Constraints*, *Seeds*, and *TestGoals*.

In the *Definitions* section, the user can define numerical constants. For

Model Phone

Definitions:

`Number threshold = 27;``end`

Types:

`EnumerativeType cameraType { 2MP 1MP NOC };``end`

Parameters:

`Boolean emailViewer;``Range textLines [25 .. 30];``Enumerative display { 16MC 8MC BW };``Enumerative rearCamera : cameraType;``Enumerative frontCamera : cameraType;``end``Constraints: ... end``Seeds: ... end``TestGoals: ... end`

Figure 3.1: A smartphone example

instance, the following statement introduces a constant with its value. Constants can be used in constraints, test goals, and seeds.

```
Number threshold = 27;
```

In the *Parameters* section the designer specifies the parameters (inputs) of the system. CITLAB language forces the designer to name parameters and to specify their types by listing all the values in their domain. Four kinds of parameter type are introduced:

- **Enumerative** for parameters that can take a value in a set of symbolic constants. Enumerative parameters are declared in the following way. For instance if the display of the cell phone can be colored (with 16 or 8 millions colors) or black and white, we introduce the following parameter.

```
Enumerative display {16MC 8MC BW};
```

- **Boolean** for parameters that can be either true or false, which can be declared as follows. For instance if the phone can have an email viewer, the designer can introduce the following parameter.

```
Boolean emailViewer;
```

- Numerical values in a **range** for parameters that take any value in an integer range. The user can also specify an integer step. For instance, if the phone has a number of lines between 10 and 30, but the designer want to test only this parameter every 5 values, he/she can write:

```
Range textLines [ 10 .. 30 ] step 5;
```

Note that the step can be omitted and in that case its value is 1, otherwise it must be a divisor of the difference between the two extreme values.

- A list of **Numbers** for parameters that take any value in a set of integers. It is like an enumerative, but mathematical comparisons and operations are allowed over these parameters.

```
Numbers year {2012 2013};
```

Types can be implicitly introduced directly when declaring a parameter belonging to an *anonymous* type or they can be defined with their name in the `Types` section to be used in parameters declaration. For instance, a type can be defined as follows and this allows two parameters to share the same domain.

```
Types:
  EnumerativeType
    cameraType { 2MP 1MP NOC};
end
Parameters:
  Enumerative rearCamera : cameraType;
  Enumerative frontCamera : cameraType;
end
```

Definition 1. *t*-way coverage A test suite achieves the *t*-way combination coverage if for all the *t*-way combinations of the *n* parameters in *P*, every variable-values configuration is covered by at least a test.

Comparison with other approaches

It is apparent that the exponential notation $x_1^{y_1} x_2^{y_2} \dots x_n^{y_n}$ can be easily translated in the proposed notation without any loss of information. It suffices to introduce $\sum y_i$ parameters p_{ij} with range domain $[1 .. x_i]$ with $i = 1 \dots n$ and $j = 1 \dots y_i$.

Note that sometimes parameters are referred as *factors* and their cardinality as *level* [11].

In AETG [59], parameters and their values are defined in the basic construct *relation* that is a table with columns for each input item, and rows for the values of each input item. *Input* items are called fields and represent the parameters. Each field can have a different number of values, each of which is a symbolic string. The translation to the CITLAB language is straightforward. For instance, in AETG, the problem of Fig. 3.1 would be represented as the following table:

emailViewer	textLines	display	rearCamera	frontCamera
true	25	16MC	2MP	2MP
false	26	8MC	1MP	1MP
	27	BW	NOC	NOC
	28			
	29			
	30			

Sometimes (like in [57]) parameters are named P_i and their domains are number starting from 1 to the domain cardinality. Also in this case the translation is straightforward.

3.1.2 Constraints

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 [26]. Constraints were first described as being important to combinatorial testing in [23] and were introduced in the AETG system. In our approach, tests that do not satisfy the 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 [26]). 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.

Definition 2. Constraints A constraint is a predicate P over the parameters, i.e., $P(p_1, \dots, p_n)$. We say that a test t satisfies a predicate P or it is a model of P and we write $t \models P$ iff P holds by substituting each parameter p_i in P with the value of p_i in t .

In CITLAB, we adopt the language of propositional logic 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.

In the CITLAB language all the constraints must be listed in a section called **Constraints** (and included between two $\#$ symbols). For instance, assume that the user wants to model the following two constraints

- if the display is black and white, then the phone cannot have any camera
- if the phone has a email viewer and front camera, the display is colored and the lines greater than the given threshold (defined as constant).

The following declarations can be added in the model in the Constraints section:

Constraints:

```
# display==display.BW => rearCamera==cameraType.NOC #

# emailViewer==true or frontCamera!=cameraType.NOC =>
  display!=display.BW and textLines>=threshold #
end
```

We assume that all the constraints must be satisfied by any test case, i.e., the constraints are conjoint with an implicit \wedge operator. For a precise semantics of constraints and some formal definitions, see [34].

Definition 3. Constraints A constraint is a predicate P over the parameters, i.e., $P(p_1, \dots, p_n)$. We say that a test t satisfies a predicate P or it is a model of P and we write $t \models P$ iff P holds by substituting each parameter p_i in P with the value of p_i in t .

In the CITLAB language all the constraints must be listed in a section called Constraints (and included between two $\#$ symbols). For instance, the following constraint may be listed in the model of Fig. 3.1,

```
# emailViewer or frontCamera != NOC =>
  display != BW and textLines >= threshold #
```

It models the requirement that, if the phone has an email viewer or a front camera, then the display cannot be black and white and the lines of the email viewer must be greater than or equals to the threshold.

We assume that all the constraints must be satisfied by any test case, i.e., the constraints are conjoint with an implicit \wedge operator.

Definition 4. valid tests Let C_j be the constraints of a combinatorial problem with $j = 1 \dots m$. A test t is *valid* if it is a model of the constraints, i.e., $\forall j t \models C_j$ or equivalently $t \models \bigwedge_j C_j$.

Due to the presence of constraints, not all the t -way combinations are coverable. We say that a combination is *not coverable* if there is no valid test that can cover it. A test suite still achieves the full coverage if it contains only valid tests and it covers all the coverable combinations.

Finding if there exists a model that satisfies the constraints, and building that model, is an NP-complete problem, since it can be translated to a SATisfiability problem. However, checking if a test is valid is linear with the dimension of the constraints and can be performed easily.

Note that the constraints may be *inconsistent*, i.e., it is impossible to find any test that satisfies them. Consistency checking of constraints is again a SATisfiability problem.

Definition 5. consistent constraints Let C_j be the constraints of a combinatorial problem with $j = 1 \dots m$. The constraints are *consistent* if there exists at least a test that satisfies them, i.e., $\exists t \models \bigwedge_j C_j$.

Comparison with other approaches The CITLAB language is expressive enough to represent most constraints we found in examples and case studies presented in many papers. Note that, in order to deal with constraints, some methods require to remodel the original specification, while very few directly support constraints in CIT. Cohen et al. [26] found that just one tool, PICT [28], was able to handle *full* constraints specification, that is, without requiring remodeling of inputs or explicit expansion of each forbidden test case. Test generation in the presence of (general) constraints has become a very active research topic in the CIT arena [16, 25].

Most tools provide a limited support for constraints. For instance, AETG [23, 59] requires to separate the inputs in a way that they become unconstrained, and only simple constraints of type **if then else** (or **requires** in [26]) can be directly modeled in the specification. The translation of those templates into our logic is straightforward. For example the **requires** constraint is translated by an *implication*; the **not supported** to a *not*, and so on.

Many approaches [26, 44] allow constraints only in the form of forbidden configurations [45]. A forbidden combination would be translated in our model as a *not* statement. For instance, a forbidden pair $x = a, y = b$ would be represented by the following constraint:

```
# not (x = a and y = b) #
```

Requiring to explicitly list all the forbidden combinations can soon become impractical. As the number of input grows, the explicit list may explode and it may become practically unfeasible and error-prone to build it.

3.1.3 Seeds

The testers can also force the inclusion of their favorite test cases by specifying them as *seed* tests [11]. The seed tests must be included in the generated test set without modification. Since seeds represent tests the user has already executed or will execute in any case, the generation algorithm should take advantage of the seeds and avoid redundant coverage of interactions.

In CITLAB, seeds can be added in the **Seeds** section and can be expressed as a sequence of assignments as follows. For instance, the user wants to force the inclusion of the following combinatorial test, by writing in the model:

```
Seeds:
# emailViewer=false , display=display.16MC,
  frontCamera=cameraType.NOC, year = 2012,
  rearCamera=cameraType.2MP, textLines=30 #
end
```

Definition 6. *seed* Given a combinatorial problem with parameters p_i with $i = 1, \dots, n$, a *seed* assigns to each parameter p_i a value in D_i , except if D_i has cardinality 1.

Note that some seeds may be invalid if they violate the constraints. Checking if a seed violates a constraint is easy and has linear complexity.

Comparison with other approaches Many approaches and tools support seeds, like AETG [11, 23], PICT [28], NSS [64], and t-tuples and IPOs [18]. Some approaches support also *partial* seeds, i.e., tests that have some parameters with unassigned values.

3.1.4 Test Goals

CITLAB allows the tester to introduce extra testing requirements by means of *test goals*. They must be considered in addition to the desired t -wise coverage. In fact, the user may be interested to test some particular critical situations or input combinations, for instance simple incomplete combinations, or more generic relations than simple combinations among parameters. For instance,

if the user wants to be sure that the test suite contains at least a test in which at least one camera is missing and the display has at least threshold lines, he can write the following test goal:

```
# (rearCamera == cameraType.NOC or frontCamera ==
   cameraType.NOC) and textLines >= threshold #
```

Note that most tools do not support test goals and seeds, however we decided to include them in the language because one of the CITLAB aims is to provide a standard common language capable to represent a rich variety of combinatorial testing concepts.

Definition 7. test goals A test goal is a predicate tg over the parameters p_i . A test goal tg is *consistent* iff there exists a test that can achieve it, i.e., formally $\exists t t \models tg$. A test goal tg is *feasible* iff there exists a valid test that can achieve it, i.e., formally $\exists t t \models \bigwedge_j C_j \wedge tg$.

Inconsistent test goals are never feasible. All the feasible test goals should be covered.

Definition 8. covered test goals Given a set of feasible test goals TG , a test suite covers them if for each test goal $tg \in TG$ there exists at least test that makes tg true.

A test generation method may support or not extra test goals. Checking feasibility of test goals is an NP-complete problem, since it can be reduced to a SATisfiability problem (assuming the all the parameters have finite domain). In CITLAB we assume that a generator method, if it supports test goals, it will also be able to check if they are feasible. We may add in the future a feasibility checker of test goals based on the use of SAT or SMT solving as done in [15]. However, checking if a test suite covers a test goal is easy.

Comparison with other approaches Test goals can be used to represent partial seeds, used for instance by AETG [23]. Partial seeds do not specify the value of each parameter, however they must be covered by the final test suite. Most tools, like AETG, complete the partial test cases by filling in random values for the missing fields and adding the completed seeds to the test suite.

4

Xtext implementation

XTEXT [82] is a framework for development of programming languages and domain specific languages. The developer can describe his very own DSL using XTEXT simple EBNF grammar language and the generator will create a parser, an AST-meta model (implemented in EMF) as well as a full-featured eclipse text editor from that. The framework integrates with technology from eclipse modeling such as EMF, GMF, M2T and parts of EMFT. Development with XTEXT is optimized for short turn-arounds, so that adding new features to an existing DSL is a matter of minutes. Still sophisticated programming languages can be implemented. The APIs of the DSL and its specific editor are generated as eclipse plugins; this important feature permits the development of tools, related to the DSL, that are fully based on the eclipse-IDE integration. XTEXT generator uses the special modeling workflow engine (MWE2) to configure the generator.

XTEXT uses the lightweight dependency injection (DI) framework Google Guice to wire up the whole language as well as the IDE components created by its generator. Most parts of XTEXT are implemented as services. A service is an object which implements a certain interface and which is instantiated and provided by Guice. When Guice instantiates an object, it also supplies this instance with all its dependent services. This architecture makes XTEXT one of the most interesting framework for development of DSL.

The DSLs developed using XTEXT can be translated to other languages or notations. XTEXT *2.0* provides the possibility to integrate a Model to Text (M2T) code generator. The code generation is done with Xtend2 that

substitutes the old template language XPand. Xtend2 is a statically-typed programming language which is tightly integrated with and runs on the Java Virtual Machine. It is the natural successor to Xtend which allows to have XPand template syntax as an expression, in fact it is more Java like than XPand, but also has some similarities. The introduction of Xtend2 makes the developer able to choose the model to model transformations approach in the code generation. It offers in fact a polymorphic dispatch, extension methods or the template syntax. Xtend files are directly compiled to Java source code, which means that the developer does not have to manage byte-code when he wants to debug a generation process.

4.1 Language and its editor with Xtext

4.1.1 DSL development

The development of a DSL and its corresponding editor, using XTEXT, passes through the following five stages from the definition of the DSL grammar to the creation of a fully eclipse integrated text editor:

1. Grammar definition.
2. Configuration of the artifacts generator.
3. Generation of the DSL APIs and of the editor plugin.
4. Implementation of the scope and the validation rules.
5. Refinement of the text formatting and the content proposal provider.

In order to develop a new DSL it is necessary to create a new XTEXT project where the practitioner describes his very own DSL using the XTEXT simple EBNF grammar language. The next operation consists of defining the language configurations of the code generator. The whole generator is composed by fragments (as listed in Fig. 4.1) for generating parsers, the serializer, the EMF code, the outline view, etc. One of the most important features that XTEXT offers is the translation of a DSL grammar to an EMF meta-model. XTEXT generator uses a special DSL called MWE2 - the modeling workflow engine.

Fig. 4.5, 4.6 show how the *Ecore-model* produced by the XTEXT generator after the parsing of the EBNF grammar language.

For the development of the CITLAB language was necessary to customize many generated artifacts:

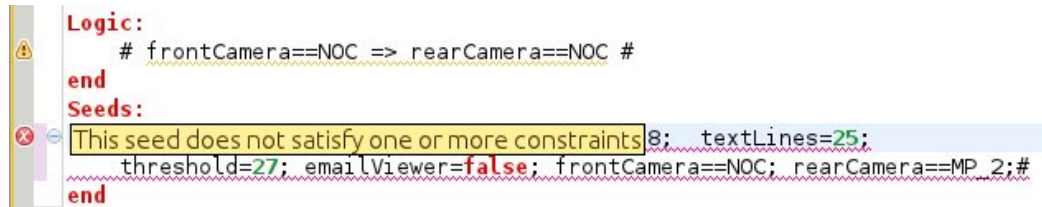
Class	Generated Artifacts
EcoreGeneratorFragment	EMF code
XtextAntlrGeneratorFragment	ANTLR
GrammarAccessFragment	Grammar access
ResourceFactoryFragment	EMF resource
ParseTreeConstructorFragment	Serializer
JavaScopingFragment	Scoping
JavaValidatorFragment	Model validation
CheckFragment	Model validation
FormatterFragment	Code formatter
LabelProviderFragment	Label provider
OutlineNodeAdapterFactoryFragment	Outline node
TransformerFragment	Outline
JavaBasedContentAssistFragment	Content assist
XtextAntlrUiGeneratorFragment	Content Helper
SimpleProjectWizardFragment	Project wizard

Figure 4.1: XTEXT fragments and artifacts

JavaScopingFragment - Scoping The auto generated scope-provider results fully operating for all the semantic and syntactical expression except for the assignments that present cross-reference. The CITLAB language ensures the correctness of its cross-reference domains using a custom `AbstractDeclarativeScopeProvider`, without this customization it would be possible to write inconsistent assignments.

CheckFragment - Model validation XTEXT provides several levels of validation for the defined language. The first level regards the syntactical validation done by the lexer and the parser, a cross link validation done by a linker and a concrete syntax validation done by the serializer that validates all constraints that are implied by a grammar. Besides these first three kinds of validation that are automatically introduced by XTEXT, the user can specify additional constraints for the model by providing generator fragments. We have introduced the validation fragments for the following rules:

1. In each expression of kind $x = y$, where x is a parameter and y is a value, y must belong to the domain of x .
2. A seed must assign a value to each parameter.
3. No seed can violate any constraint.



```

Logic:
  # frontCamera==NOC => rearCamera==NOC #
end
Seeds:
This seed does not satisfy one or more constraints: textLines=25;
threshold=27; emailViewer=false; frontCamera==NOC; rearCamera==MP_2;#
end

```

Figure 4.2: Validation of seeds

The validation of the last requirement (3), requires the evaluation of constraints. This is performed by two classes:

- Logic Evaluator: evaluates Boolean expression starting from the value of its operands.
- Arithmetic Evaluator: computes the integer value of a numeric expression.

The validation is performed run-time while the user types the model. If the validator finds an error in the model it generates an error message. The nature of the error is indicated in the error-log view of eclipse and the point in which the error occurs is marked in the editor. The editor is able to check the presence of inconsistencies between seeds and constraints while the user writes them.

Editor-Contributes Once the grammar, the meta-model of the language, and the validation rules are defined, XTEXT generates an eclipse-based development environment providing editing experience known from modern IDEs. The editor provides the developer with an IDE integrated in eclipse for writing combinatorial models in the CITLAB language. The editor features a content assist, quick fixes, a project wizard, template proposal, outline view, hyperlinking, and syntax coloring. For instance, the content assist shown in Fig. 4.3 helps the user to write the seeds and constraints while he types.

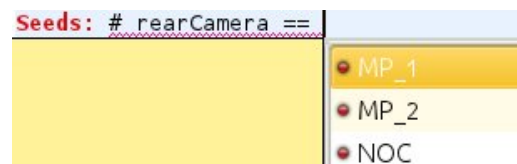


Figure 4.3: Content assist

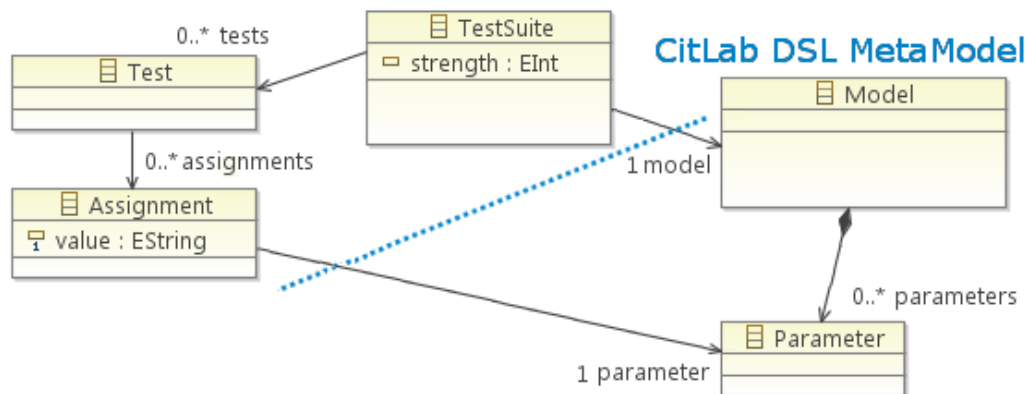


Figure 4.4: A meta model for Tests Suites

Test Suite meta-model and Java utils CITLAB introduces also a simple meta-model for tests and test suites. The meta-model is shown in Fig. 4.4, which reports also a fragment of the language meta-model. A **TestSuite**, which refers to a **Model**, contains several **Tests** which can contain several **Assignments**. Each assignment gives a String value to a **Parameter**. We have added also the code to check the validity of the assignments and of the tests. We have developed several classes and methods capable to perform routine tasks like generating all the test requirements for a given strength, checking if a test or a seed violates some constraints, and checking if a test goal is covered by a given test suite.

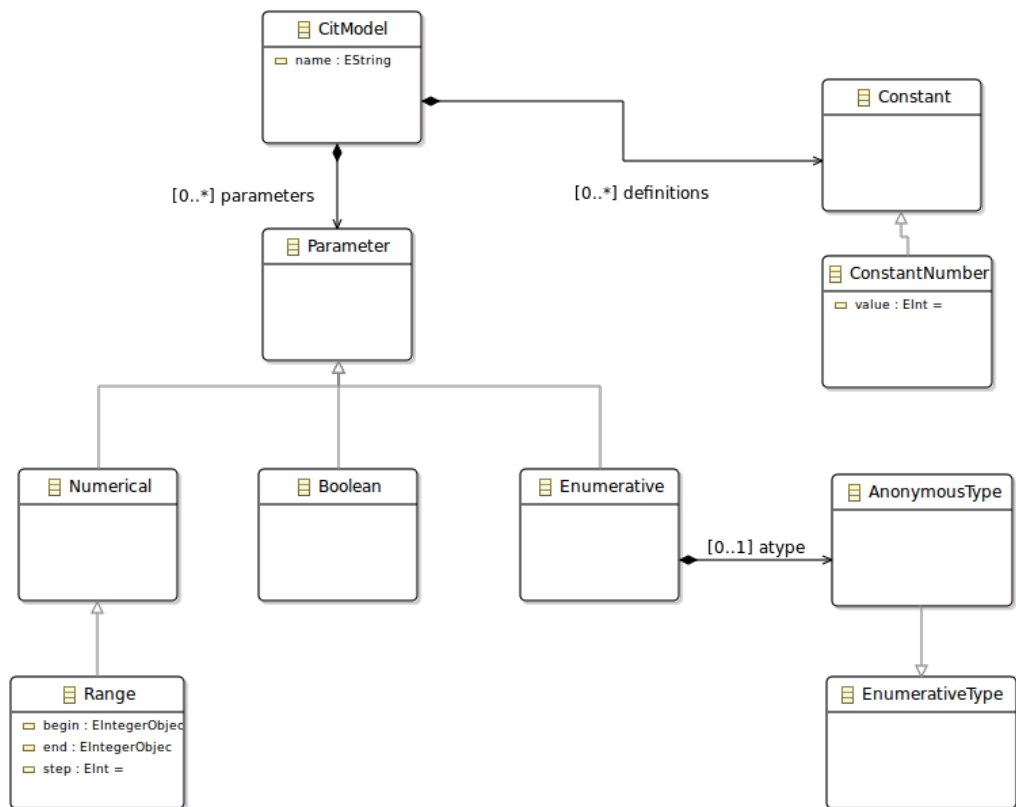


Figure 4.5: UML Class Diagram of the grammar of CITLAB

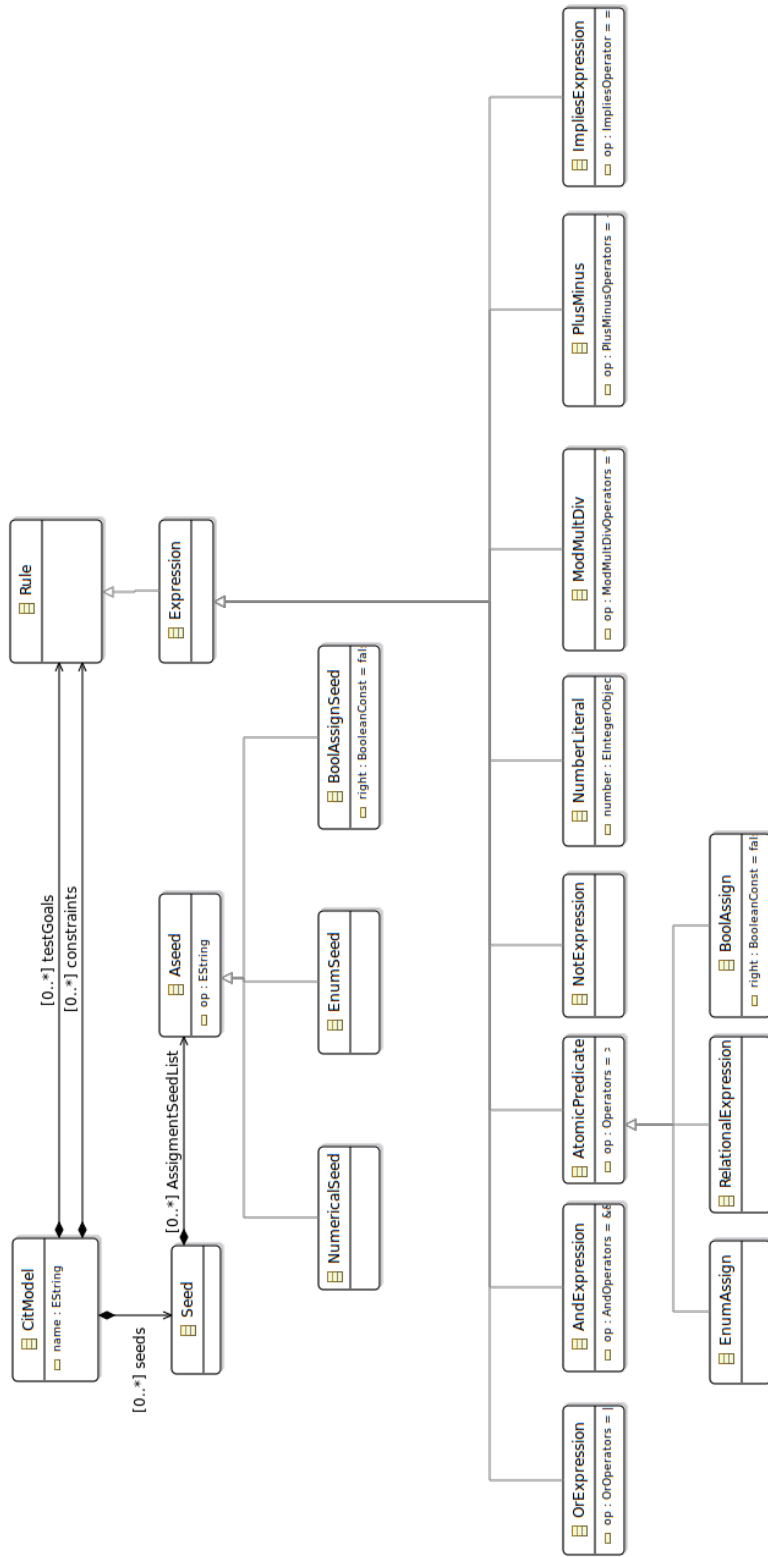


Figure 4.6: UML Class Diagram of CitLAB logic constructs

5

Framework architecture

Besides the definition of a language for combinatorial problems, together with its editor, meta-model, and Java API to manipulate combinatorial models, a further goal of CITLAB is to introduce a framework for the definition and implementation of actual test generators and a set of exporters/importers to and from other languages to foster tools interoperability. In order to ease the development and deployment of such components that can extend its capabilities, CITLAB relies on the extension techniques as defined by the eclipse framework. In eclipse, a framework or a platform can accept new contributions as plugins by defining *extension points*. External contributors can add to the framework new plugins by implementing *extensions*. One can think of an extension point as a port – an entry point for other plugins to offer services.

5.1 Defining the extension points

An extension is a plug that connects to the right port. An extension point defines a contract between the platform and the service provider introduced as plugin. The extension implementation is the actual service which will be added to the platform by using the plugin mechanism of eclipse.

There are several benefits from this architecture. New plugins can be dynamically added and removed from the platform without recompiling them. Third-party tools can be easily added to the platform by registering them as

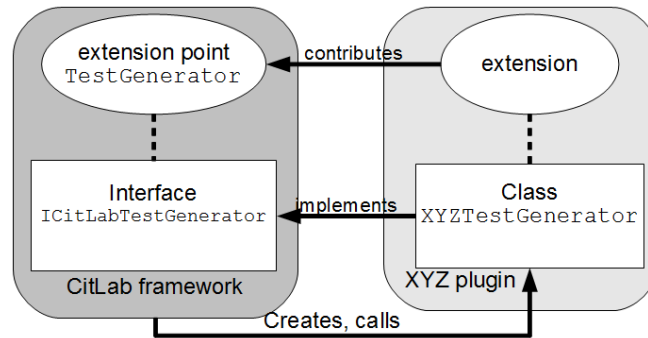


Figure 5.1: **TestGenerator** extension point and corresponding extension

extensions. A plugin includes some descriptive information and the platform extension point can decide how to use it. For instance, a plugin can declare to support a feature and the platform can decide if it is worth loading the extension or not. The development of a plugin is strongly decoupled with the development of the platform, making easy for third-parties to contribute to the framework.

Very often an extension point introduces a Java interface or an abstract class that must be implemented by the extensions. As depicted in Fig. 5.1, CITLAB introduces an extension point called **TestGenerator** which defines the abstract class **ICitlabTestGenerator** declaring the methods necessary for any test generator. A plugin (XYZ in Fig. 5.1) must define a class (**XYZTestGenerator**) extending the required abstract class in order to extend the platform with new test generators and register this extension into the platform. The platform will become aware of new generator and will be able to create and call instances of that class when needed.

CITLAB introduces the six extension points listed in Tab. 5.1 together with the abstract classes and the required methods, while some implemented extension are repotederted in Fig.5.2.

All the code for CITLAB is available under the Eclipse Public License¹.

5.1.1 Test generation plugins

Any test generator plugin must introduce a class that implements the abstract class **ICitLabTestGenerator**. A test generator must declare if it supports the constraints, the seeds and the test goals (*accept* methods). Moreover, it must define a method that actually computes the test suite for

¹All the source code is available at <http://code.google.com/a/eclipselabs.org/p/citlab/>.

Extension Point	Abstract Methods
TestGenerator	ICitLabTestGenerator TestSuite generateTests(Model m, int nWise) boolean acceptConstraints(List<Constraint> c) boolean acceptSeeds(List<Seed> s) boolean acceptTestGoals(List<TestGoal> tg)
Exporter	ICitLabExporter void export(Model m) boolean acceptConstraints(List<Constraint> c) boolean acceptSeeds(List<Seed> s) boolean acceptTestGoals(List<TestGoal> tg)
Importer	ICitLabImporter Model import(Reader r)
TestSuiteExporter	ICitLabTestSuiteExporter void save(TestSuite ts, String file)
TestSuiteValidator	ICitLabTestSuiteValidator boolean isValid() boolean isComplete() boolean isMinimal()
ModelValidator	ICitLabModelValidator String Analyze(CitModel model)

Table 5.1: Extension points defined by CITLAB

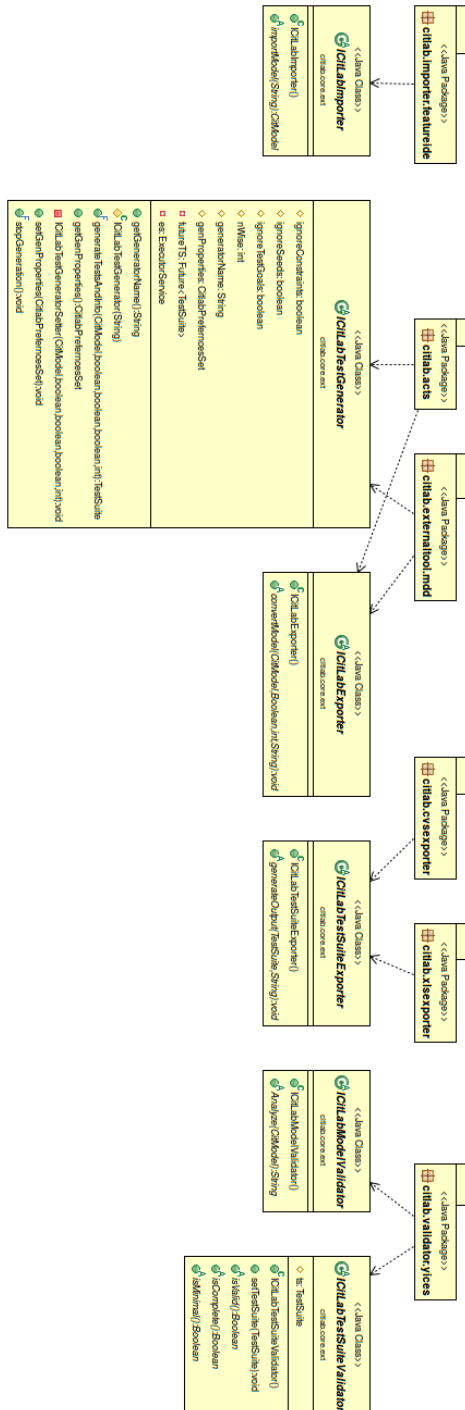


Figure 5.2: UML Class Diagram of some implemented extensions

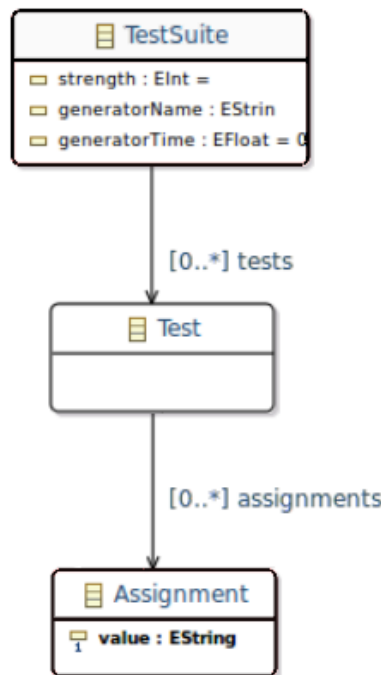


Figure 5.3: Test Suite Ecore Class Diagram

the n-wise coverage of a given model (*generateTests* method). The method will be invoked by the CITLAB user interface upon an user request. If an extension does not support some model features, like constraints for instance, the CITLAB user interface will prompt the user and will suggest to strip the model from that feature in order to continue with the test generation. Any test generator can decide to partially support a feature. For instance, a test generator may support only constraints having a particular pattern (like forbidden tuples or CNF). In this case, its method *acceptConstraints* will examine the constraints and it will decide if they conform to the requested pattern. In order to standardize the format of the test suite, produced by the different generator plugins, we have designed a *Test Suite Ecore Model*. The output of each generators must be translated to these *Java* objects in order to respect the CITLAB standard and to preserve every links to the original CITLAB model. The Class Diagram of the Test Suite is reported in Fig.5.3. These classes are embedded in an Eclipse plugin that also exposes the utilities described in 4.1.1.

```

...
def getSize(Parameter param){
  switch (param) {
    Enumerative:
      '''«(param as Enumerative).type.elements.size»'''
    Boolean :'''2'''
    Number :'''1'''
    Range:
      '''«(((param as Range).end—
        (param as Range).begin) as Integer).toString»'''
  }
}

```

Figure 5.4: A fragment of the XTEXT exporter to CASA

5.1.2 Translation to and from other notations and tools

CITLAB introduces also two extension points for importing and exporting models from and to other notations and tools (as reported in Tab. 5.1) By defining exporters and importers, the researchers could use the CITLAB language as a sort of *pivot* language. In DSLs, a pivot language can be used for exchanging models in several notations employed by different tools.

A possible way to define importers and exporters is to use Model to Text (M2T) or Model to Model (M2M) transformations. To this purpose, Xtend 2 can be used. For instance, we have defined an exporter to the CASA [21] language by defining a model to text transformer in Xtend. The transformer is translated to a Java class which extends abstract class required by the exporter extension point. Inside the transformer, the designer can use polymorphic dispatching of template definitions and easily navigate through models. For instance, a fragment of the transformer for CASA is reported in Fig. 5.4 in which the size of a domain is converted to a string.

Transformations could also be defined at higher level, between meta-models (if provided by the target notations). In this case, the meta-model provided by CITLAB for combinatorial problems, could be used as *pivot meta-model* by providing suitable model to model transformations in order to allow tools interoperability. According to the view presented in [9], a pivot meta-model of a given formalism or language L is intended as a platform independent modeling language which abstracts a certain number of general concepts about L . The integration among tools supporting L can be

achieved by providing, for the notation L' (a dialect of L) of each tool $T_{L'}$, a meta-model - seen as a platform specific modeling language - and model transformations to the pivot and from the pivot to the L' -meta-model. Hence, the meta-model of the notation L^i of a tool T_{L^i} can be linked to the meta-model of the notation L^j of another tool T_{L^j} by the composition of the two transformations from L^i -meta-model to the pivot and from the pivot to the L^j -meta-model. In this way, the interoperability between tools T_{L^i} and T_{L^j} is achieved by translating PSM (Platform-specific Model) models written in L^i to L^j and vice versa.

5.1.3 Test suite exporters

The third extension point is *TestSuiteExporter* that allows designers to add new plugins for exporting the test suites generated into files in specific formats. The plugin must introduce a class that extends the abstract class **ICitLabTestSuiteExporter**. In CITLAB we have already implemented an exported towards the Microsoft Excel format.

5.1.4 Validators

The last two extension points are *TestSuiteValidator* and *ModelValidator*. They allows designers to add new plugins for validating the test suites produced using the generators or for validating a combinatorial model. The plugins must introduce a class that extend the abstract class **ICitLabTestSuiteValidator** or the abstract class **ICitLabModelValidator**. In CITLAB we have implemented two validators as described in Chapter 7.

6

MDD generator

Efficient test generation for CIT is still an open problem especially when applied to real models having meaningful size and containing many constraints among inputs and components. In this chapter we present a novel technique for the automatic generation of compact test suites starting from models containing constraints given in general form. It is based on the use of Multi-valued Decision Diagrams (MDDs) which prove to be suitable to efficiently support CIT. We devise and experiment several optimizations including a novel variation of the classical greedy policy normally used in similar algorithms. The results of a thorough comparison with others similar techniques are presented and show that our approach can provide several advantages in terms of applicability, test suite size, and generation time.

6.1 Introduction

Combinatorial Interaction Testing (CIT) helps tester to find defects due to the interaction of components or inputs. It is based on the assumption that not every parameter or parameter value contributes to every fault and many faults are caused by *interactions* among parameters. CIT tests the interaction in a systematic way. For instance, *pairwise* testing requires that every pair of parameter value is tested at least once. It can be generalized by the *t-way* testing requiring that every *t*-tuple is included in the test suite. CIT has been proved to be very effective in finding faults [53, 54].

A major problem in CIT is the generation of compact test suites, especially when the cost of executing each test case is high. Suitable tools can produce very compact test suites. For instance [54], consider a manufacturing automation system that has 20 controls, each with 10 possible settings—a total of 10^{20} combinations, which is far more than a software tester would be able to test in a lifetime. A good CIT tool can produce a test suite for the pairwise testing with only 180 tests in it. Note that the general problem of finding a minimal set of test cases that satisfies t -wise coverage is NP-complete [75, 80].

Applying CIT to highly-configurable software systems is complicated by the fact that, in many such systems, the parameters are rarely independent from each other. There exist constraints that model dependencies among parameters that render certain combinations invalid or some combinations mandatory [25]. The presence of constraints increases the complexity of the test generation task: if constraints on the input domain are to be taken into account, even finding a single test or configuration that satisfies the constraints is NP-complete [11], since it can be reduced in the most general case to a satisfiability problem. The presence of constraints may either reduce or increase the size of the final test suite, since some combinations may become infeasible while others may become more difficult to cover [11]. For this reason, several works, like this, target explicitly the test generation for CIT in the presence of constraints. In this chapter we focus on reaching a good trade off between the size of the generated test-suite and its time of generation.

Our algorithm is a classical greedy algorithm which produces a test at the time, similar to AETG [23] (as opposed to algorithms proceeding by one parameter at the time, like IPO [78] and its variants [18, 57]). When building a single test, it chooses the *best* parameter and assigns the *best* value to it until a test is complete. However, we advance with respect to the state of the art by adopting the following original approaches:

- we employ a data structure, called Multivalued Decision Diagram which is particularly suitable to combinatorial problems in order to represent inputs, their domains, and constraints over those inputs;
- we soften the classical greedy algorithm by reducing the importance of the number of tuples covered by the test currently built, by weighting parameters and tuples depending on the constraints in order to reduce the final size of the test suite;
- we introduce several optimizations, including constraints partitioning, in order to speed up the test generation.

6.2 Multivalued Decision Diagram

Binary Decision Diagrams (BDD) and Multi Valued Decisions Diagrams (MDD) are commonly used structures for representing Boolean functions and multi-valued functions, respectively. Such decision diagrams are widely used within the domain of system design verification. A Multi-Valued Decision Diagram (MDD) is a directed acyclic graph used to represent/encode some multi-valued function $f(X)$ $X = \{x_0, x_1, \dots, x_{n-1}\}$, where the x_i are p -valued, i.e. each x_i can have value $0, \dots, p-1$. The function takes values from 0 to $q-1$. Such a function can be represented by a MDD which is a directed acyclic graph (DAG) with up to q terminal nodes each labeled by a distinct logic value $0, 1, \dots, q-1$. Every non-terminal node is labeled by an input variable and has p outgoing labeled edges; one corresponding to each logic value. The diagram is ordered if the variables adhere to a single ordering on every path in the graph, and no variable appears more than once on any path from the root to a terminal node. an MDD is a natural extension of the reduced ordered binary decision diagrams (ROBDDs) to the multiple-valued case. It is easy to see that BDDs may be considered a special variant of MDDs, where the value of the range p for representing the input and output values for the function is equal to 2. Due to the nature of combinatorial problems, we focus on MDDs representing two-valued logic functions and we allow the inputs in X to have domains of different size. If we define the set of possible values for variable x_k as $\mathcal{D}_k = 0, 1, \dots, |\mathcal{D}_k| - 1$ and $\mathcal{D} = \mathcal{D}_1 \times \mathcal{D}_2 \times \mathcal{D}_3 \times \dots \times \mathcal{D}_n$, a two-valued logic function is a mapping $\mathcal{D} \rightarrow B$, where B is the Boolean constant set $\{F, T\}$. This mapping, $f : \mathcal{D} \rightarrow B$, can be represented by an MDD and can be used to select the values in \mathcal{D} that can be accepted: if the values X for the variables in \mathcal{D} are valid, then $f(X) = T$, otherwise $f(X) = F$. MDDs can represent logic function using less memory and shorter path than BDDs. From a theoretical point of view, Nagayama [62] demonstrated that the amount of memory used by mapping Boolean function with Boolean variables to heterogeneous MDD is lesser than using OBDD directly. O’Sullivan et al. [41] have experimentally compared BDDs and MDDs over a set of real-world and artificially constructed instances with non Boolean domains for compiling constraint satisfaction problems (CSPs). They have demonstrated that MDDs are often smaller than log-BDDs on configuration instances. In all the considered product configuration instances the MDD representation uses roughly four times fewer edges than log-BDDs. Furthermore, direct BDDs are two to five times larger than the log-encoded BDDs. In order to achieve this performance improvement, it is very important the use of techniques that can reduce the size of MDDs. To our knowledge, Meddly [5] is the only opensource C/C++ library that natively supports these

```

Model phone
Parameters:
  Enumerative display { 16MC 8MC BW };
  Enumerative frontCamera { 2MP 1MP NOC };
  Boolean emailViewer;
end
Constraints: # emailViewer => display != BW # end

```

Listing 6.1: A mobile phone example

DDs. According to our opinion, Meddly native support for MDDs and their variants, along with its performance makes it a good candidate for applications in areas where these DDs make sense. Meddly provides a breadth of builtin operations for all supported DDs:

- Unary: Complement, Cardinality.
- Binary on booleans: Union, Intersection, Difference.
- Binary on integers and reals: $+$, $-$, \times , \div .
- Relational: $=$, $<$, $>$, \leq , \geq , \min , \max .
- Symbolic: Given an initial set of states and a next state function, Meddly can symbolically generate the reachable states in one step.

Meddly supports several operations between MDDs, including the complement, the union, and the intersection. These operations can be mapped to logic operation between the Boolean functions represented by our Boolean-valued MDDs. Given an MDD m with function f , its complement m^c represents the function $\neg f$. The union between two MDDs $m_1 \cup m_2$ represents the function $f_1 \vee f_2$. The intersection between two MDDs $m_1 \cap m_2$ represents the function $f_1 \wedge f_2$. Given the MDD m , its cardinality $|m|$ is the number of all the possible paths to the terminal node \mathbf{T} . The cardinality can be used to check consistency among Boolean functions: if f_1 and f_2 are inconsistent, the intersection between their MDDs is empty.

6.3 Using MDD for CIT

A combinatorial model with n parameters each with cardinality p_i can be easily represented by an MDD that has n non-terminal nodes labeled by the name of every parameter and each node for parameter P_i has p_i outgoing

labeled edges to the node for P_{i+1} for $i < n$ and to the \mathbf{T} terminal node for P_n . We call this MDD M_{TS} . For instance, the MDD in Fig. 6.1 represents the M_{TS} for the phone given in Listing 6.1 ignoring its constraints. In the following figures, edges sharing the same starting and final node, are shown with a unique arch and the list of labels.

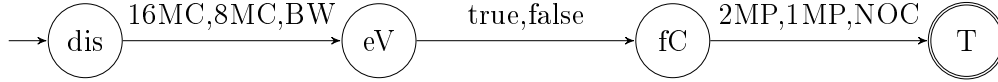


Figure 6.1: MDD for the combinatorial problem of Listing 6.1

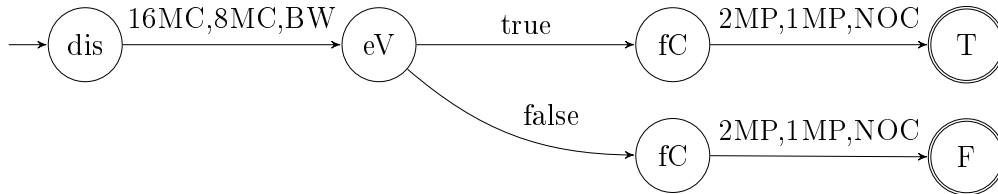
Every path from root to the terminal \mathbf{T} is a valid configuration. The M_{TS} represents all the tests, i.e., all the possible paths from the start to the terminal node. The cardinality of M_{TS} is equal to $\prod_{i=1}^n p_i$ which is equal to the total number of possible tests.

A test case is a set of u values, one for each parameter. The set of all possible tests is $TSA = P_1 \times P_2 \times P_3 \times \dots \times P_u$ and has $\prod_{i=1}^u p_i$ elements. The coverage criterion Z of a CIT problem is defined by its strength t . A test set $TSZ \subseteq TSA$ fulfills the criterion if every t -wise combination of values from different parameters is covered with at least one test case in TSZ .

The equality formula that associates parameter P_i to one of its values v , i.e., $P_i = v$ can be easily represented by the following function.

$$f(p_1, \dots, p_n) = \begin{cases} T & \text{if } P_i = v \\ F & \text{if } P_i \neq v \end{cases}$$

Such function can be represented by an MDD in which all the paths, traversing the edge outgoing the node P_i with label v , terminate to the terminal \mathbf{T} while all the other ones terminate in \mathbf{F} . For instance, the assignment $eV = \text{true}$ is represented by the following MDD.



A similar MDD representation can be given for every tuple, which assigns values to a list of parameters. A path terminates to the node \mathbf{T} if and only if it contains an assignment contained also in the tuple. For instance, the tuple $(eV = \text{true}, fC = 2MP)$ is represented by the MDD shown in Fig. 6.2.

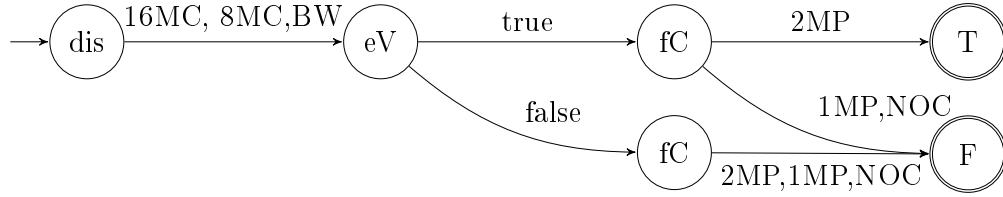


Figure 6.2: The MDD representing the pair ($eV = \text{true}$, $fC = 2MP$)

In most configurable systems, constraints or dependencies exist between parameters. Since we assume that the constraints corresponding to a CIT problem can be described by propositional logic with equality, we can describe every model constraint c_i using a Boolean general formula containing operators \neg , \vee , \wedge over equalities among parameters and their values. Every constraint can be represented by an MDD modeling its truth function. The MDD for any constraint can be built using the representation of equality formulas and the operations between MDDs presented in Sect. 6.2.

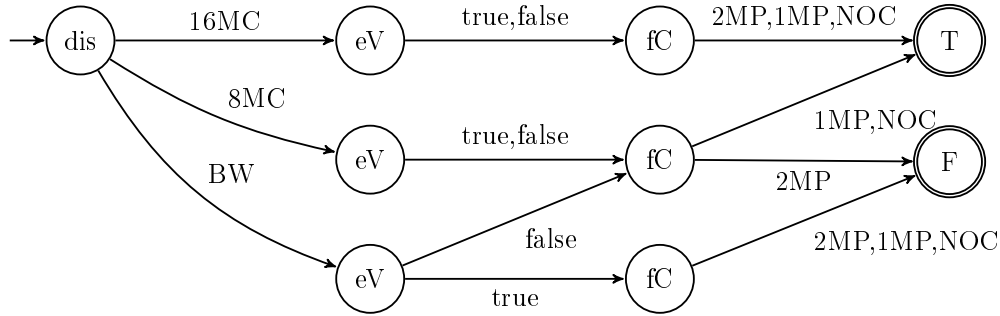
Let $U = \bigcup_{i=1}^u P_i$ be the set of all possible input values. We define an assignment for each values in U so every value is associated to its own parameter. Our approach is based on the modeling of the combinatorial interaction test problem with constraints as a single propositional logic formula.

In order to include the constraints in the MDD M_{TS} representing the unconstrained model, we can use the operations between MDDs. The conjunction of the MDD M_{TS} with all the constraints c_i restricts the set of satisfying interpretations of the function associated to M_{TS} such that it contains exactly those interpretations that correspond to valid test cases. Let m_{c_i} be the MDD for the constraint c_i , and the MDD M_{VS} be defined by the following formula: $M_{VS} = \bigcap_{i=1}^n m_{c_i} \cap M_{TS}$.

The MDD M_{VS} can be obtained by integrating the constraints c_i into the MDD M_{TS} by changing its original topology. A constraint changes the state of one or more paths from valid to not-valid. In the original MDD there are n levels and n not-terminal nodes, where n is the number of parameter. In order to model not-valid paths it is necessary to duplicate some nodes. The MDD M_{VS} preserves the number n of levels but has some more not-terminal nodes. The M_{VS} represents all the *valid tests*, i.e. all the possible paths from the start to the terminal **T** node.

An example of the MDD M_{VS} representing the model and the constraints for the phone problem is shown in Fig. 6.3. For instance, the combination ($dis = BW$, $eV = \text{true}$) is not valid, since the requirement prohibits a BW display with the emailViewer.

On the contrary, the test ($dis = 16MP$, $eV = \text{true}$, $fC = 2MP$) is a valid

Figure 6.3: M_{VS} for the phone with the constraints

test, as shown by the corresponding path leading to the terminal **T** node in the MDD.

an MDD with cardinality 1, i.e. with only one path to the **T** terminal node, represents one valid test. An example is shown in Fig. 6.4.

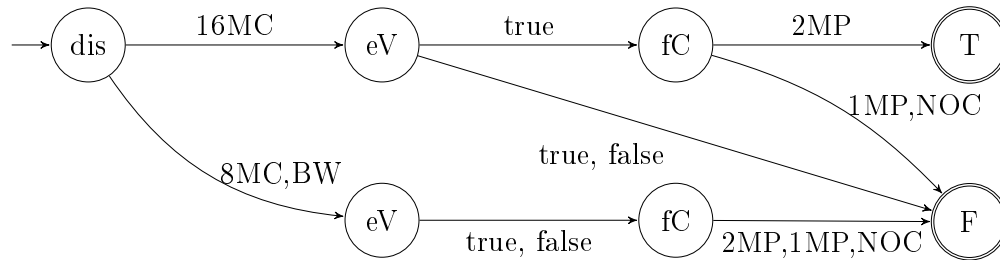


Figure 6.4: an MDD representing a single test case

6.4 An MDD-based algorithm for CCIT

We have devised an automatic algorithm for the generation of combinatorial test suites based on the use of MDDs. The algorithm takes as input the MDD M_{VS} representing the intersection between the model domain and the constraints and produces as output the desired test suite R . It builds one test at the time until all the testing requirements are achieved. When building a single test, it proceeds in a greedy manner: it chooses one *optimal* parameter, that is not already set in the test, and its *optimal* value, according to our weighting criteria, and it adds this assignment to the test to be built. In the following we explain in details the algorithm which is reported in Alg. 1.

Firstly, we populate a list of tuples T_{TC} including all the combinations to cover based on a given coverage criterion C , usually *t-wise* coverage. Some

Algorithm 1 Generation of the test suite R

Input: M_{VS} : MDD for the model with the constraints**Output:** R : set of MDDs representing the test set $T_{TC} \leftarrow feasibleTuples(M_{VS})$ $R \leftarrow \emptyset$ **while** $T_{TC} \neq \emptyset$ **do**▷ Build single test M_{nc} $M_{nc} \leftarrow M_{VS}$ $P \leftarrow sortParamList(T_{TC})$ **for all** $P_i \in P$ **do**▷ Fix every parameter in P $value \leftarrow chooseBest(P_i, M_{nc}, T_{TC})$ $M_{nc} \leftarrow M_{nc} \sqcap P_i = value$ **if** $|M_{nc}| = 1$ **then****break****end if****end for** $T_{TC} \leftarrow removeCoveredTuples(M_{nc})$ $R \leftarrow R \cup M_{nc}$ **end while**

tuples may be infeasible because of the constraints. In order to filter all the valid tuples, we use MDDs as well: the function *feasibleTuples* returns all the tuples required by the criterion C that have a non-empty intersection with the MDD M_{VS} .

We then start the iteration part where we generate, for each iteration, a test case M_{nc} represented by an MDD with final cardinality equal to 1. At the end of each iteration we update T_{TC} removing the tuples covered by the generated M_{nc} until T_{TC} is empty.

In the single iteration we initialize M_{nc} to the valid set M_{VS} , we then sort all the parameters (*sortParamList*) by simply counting for every parameter p the number of tuples in T_{TC} that contain p . We then start assigning every P_i to the best value for it, by taking the *value* producing an assignment that is compatible with M_{nc} and that maximizes the coverage of tuples in T_{TC} .

This basic algorithm is a classical greedy algorithm that generates a test at the time and tries to cover as many uncovered tuples as possible. It can be improved in several directions, as explained in the following sections.

6.4.1 Optimization: Weighting compatibility

Although most greedy algorithms consider only the number of remaining tuples that will be covered in order to determine the best choice [13], it is

Algorithm 2 Computation of weights

```

function ASSIGNWEIGHT( $T_{TC}, M_{VS}$ )
  for all  $T \in T_{TC}$  do  $weight(T) \leftarrow 0$  end for
  for all  $(T_i, T_j) \in T_{TC} \times T_{TC}$  with  $i < j$  do
    if  $M_{VS} \cap T_i \cap T_j = \emptyset$  then
       $weight(T_i) \leftarrow weight(T_i) + 1$ 
       $weight(T_j) \leftarrow weight(T_j) + 1$ 
    end if
  end for
end function

```

well known that such greedy policy can lead to bigger test suites, even for unconstrained models¹. In the presence of constraints, this greedy policy can be even more inefficient. One reason is that, in the presence of constraints, this policy leaves at the end all the tuples that are “more” difficult to cover, because the constraints limit the number of valid test cases that can cover them. In this way, the last generated tests cover only a few tuples not covered yet, leading to bigger test suites.

We propose to *weight* every tuple depending on its compatibility degree with respect to the other tuples not covered yet considering also the constraints. Heavy tuples are more difficult to cover and they should be fixed sooner than light tuples. To weight tuples, we introduce a dynamic function *weight* that measures the weight of every tuple and we modify the Alg. 1 by calling the function in Alg. 2 that assigns the weights before ordering the parameters. We modify the functions *sortParamList* and *chooseBest* accordingly in order to consider tuple weights.

The function ASSIGNWEIGHT increases the **weight** (initially set to 0) for all the tuple pairs (T_i, T_j) with T_i and T_j in T_{TC} that are mutually exclusive by considering also the constraints. Checking if two tuples are compatible can be performed by using the usual intersection operator among MDDs. For instance the tuples $(dis = BW, fC = 2MP)$ and $(fC = 2MP, eV = true)$ would have their *weight* increased because they are incompatible due to the constraints and this can be easily computed using the MDD of Fig. 6.3.

Although we can rely on the efficiency of MDDs for the computation of weights, Alg. 2 has complexity $N^2/2$ where N is the number of remaining tuples to cover (T_{TC}) and this can increase the computation time. For this reason we define a simplified algorithm (Alg. 3) that is less precise but it

¹Bryce and Colbourn report in [12] the example in which a simple greedy algorithm provides a solution of 1,222 tests. Relaxing the greedy behavior or other algorithms can provide much smaller test suites till 910 tests

Algorithm 3 Approximate and faster computation of weights

```

function ASSIGNWEIGHTFROMPARAMS( $T_{TC}$ )
  for all  $T \in T_{TC}$  do  $weight(T) \leftarrow 0$  end for
  for all  $P_i$  and  $T_i \in T_{TC}$  with  $P_i \in T_i$  do
     $weight(P_i) \leftarrow weight(P_i) + 1$ 
  end for
  for all  $T_i \in T_{TC}$  and  $P_i \in T_i$  do
     $weight(T_i) \leftarrow weight(T_i) + weight(P_i)$ 
  end for
end function

```

is much faster than Alg. 2. This algorithm 3 first assigns a weight to every parameter depending on the number of remaining tuples to cover (T_{TC}) that contain it. Then, every tuple gets a weight that is the sum of the weights of the parameters in it. It does not consider the model and its constraints (M_{VS}), it does not need to perform any operation among MDDs, and for this reason is much faster.

We devised the following policy. If the number of tuples to be covered ($|T_{TC}|$) is greater than a **threshold**, the weighting is performed by Alg. 3 otherwise the more precise Alg. 2 is used instead.

6.4.2 Optimization: Repetitions

Our algorithm produces non deterministic results, since when ordering the parameters and when identifying the best value for the chosen parameter, it may occur that two or more choices are equally valid. In this case the algorithm randomly chooses one possibility. The choice may affect the behavior of the test generation only much later (typically only in the last steps). One possibility is to repeat with a different random seed the entire algorithm (except the evaluation of tuple feasibility) in order to see if by chance a better solution is found. We call this optimization *repetition*, as defined in [13]. We manage the repetition policy by setting the following three parameters $repeat_{\min}$, $repeat_{\max}$, and $repeat_{\text{better}}$. When repetition is activated, the algorithm generates at least $repeat_{\min}$ times a new test suite. It keeps generating new test suites unless for $repeat_{\text{better}}$ the test suite is not smaller than the best found so far. In any case no more than $repeat_{\max}$ generation runs will be executed. The smallest test suite found is returned.

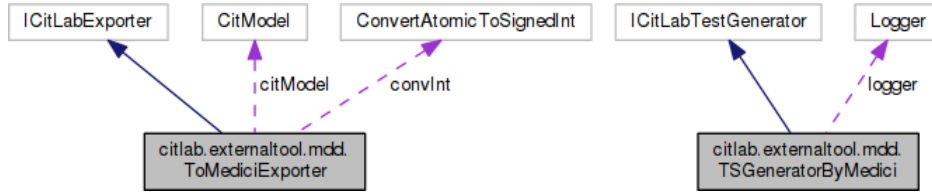


Figure 6.5: CITLAB plugin classes

6.5 Experiments

We have implemented the algorithm presented in the previous section in a prototype tool called MEDICI (Multivalued Decision diagrams for Combinatorial Interaction testing). We have integrated MEDICI in CITLAB [19], as a generator plugin. MEDICI is written in C++ and is based on Meddly [5] for the MDDs. It has been embedded in CITLAB and it is freely available². CITLAB simply exports the necessary input file for MEDICI and executes it as reported in Fig. 6.5. Note that MEDICI accepts constraints in general form and thanks to the fact that it uses MDDs, it avoids the time consuming conversion to CNF.

As benchmarks for CCIT problems we have gathered a wide set of 117 models with constraints taken from the literature (Casa [21, 25, 38], FoCuS [74], ACTS [1], and IPO-S [18]) and from SPLOT SPLs repository, and used (in subsets) also by many other papers. The benchmarks can be found on the CITLAB web site and can be used for further comparisons. For the sake of brevity, we show, in Tab. 6.1, only some useful statistical summary of the data reported on the web site. We run the experiments on a PC with two Intel(R) Xeon(R) CPU E5-2630 @ 2.30GHz and 64 GByte of RAM. We exploit the multi-core architecture by running 20 threads in parallel and we run all the experiments with the pairwise coverage and 50 runs.

Table 6.1: Characteristics of the CCIT benchmarks.

	#Variable	#Constraints	Domain size	#Valid confs	Ratio ³
Minimum	3.00	0	8.00	1.00	2.44×10^{-29}
Maximum	259.00	388	9.26×10^{77}	2.44×10^{62}	1.00
Mean	44.85	27.46	1.16×10^{76}	5.89×10^{60}	0.25
Median	15.00	15	8.35×10^{04}	2.60×10^{04}	7.86×10^{-02}

²CitLab and its MEDICI plugin can be found at <http://sourceforge.net/projects/citlab/>

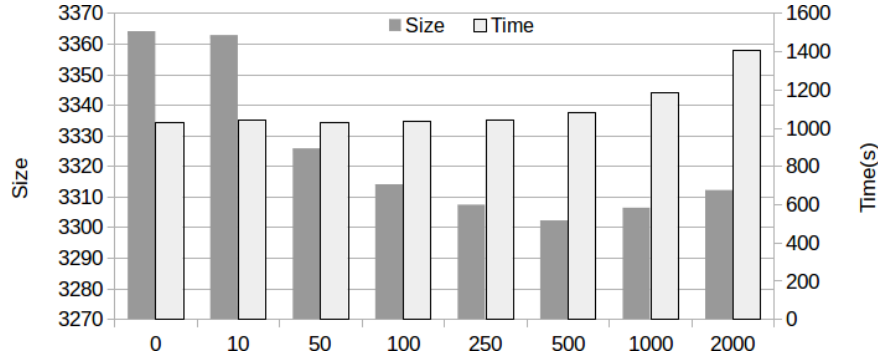


Figure 6.6: Test suite *size* and *time* depending on the threshold

Let \overline{size}_m be the average of the test suite size for model m over all the runs and \overline{time}_m be the average of the time for model m , we introduce *size* and *time* defined as: $size = \Sigma \overline{size}_m$ which is the sum of the averages of the test suite sizes and $time = \Sigma \overline{time}_m$ which is the sum of the averages of the executions times (in seconds). We will use *size* and *time* as performance indexes.

6.5.1 Optimal threshold value

We perform an experiment in order to discover the impact of the threshold introduced in Sect. 6.4.1 over the test generation size and time (with 1 repetition). Fig. 6.6 reports how the test suite size and time changes depending on the value of the threshold⁴. As the graph shows, the test suite size has a minimum for a threshold around 500, while it becomes sensibly greater with thresholds smaller than 250. The time becomes significantly greater for threshold greater than 250. From now on, we chose as optimal threshold the default value of 250.

6.5.2 Using compatibility

We experiment the efficacy of the use of the compatibility and weights in order to choose the optimal parameter and value w.r.t. the classical greedy algorithm as explained in Sect. 6.4.1 by performing a comparison with a version of MEDICI that avoids this optimization and uses a greedy algorithm over the number of covered combinations. The results are shown in the chart of Fig. 6.7.

³Ratio=(#Valid configurations / Domain Size)

⁴Threshold values are in the set {0,10,50,100,250,500,1000,2000}.

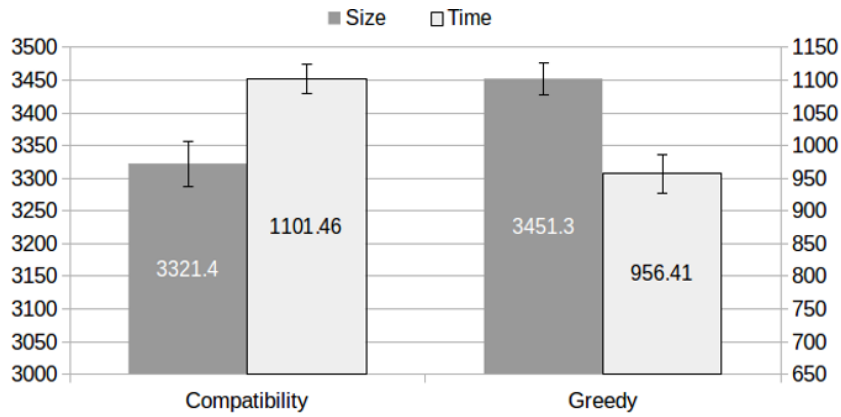
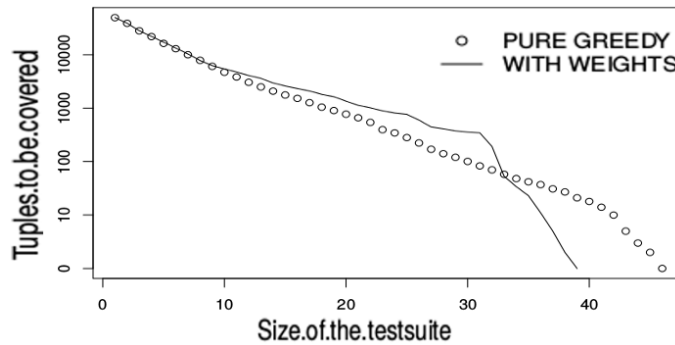


Figure 6.7: Greedy vs Compatibility comparison with optimization

Figure 6.8: Tuple coverage rate for b_{12} with optimization

We observe that using the compatibility leads to smaller test suites (*size* is around 4% smaller on average) with an increase of the time (*time*) of around 15%. Using the proposed technique slows the rate in which uncovered tuples are covered but reduces the final test suite size. For instance, Fig. 6.8 reports the size of still uncovered tuples (y-axis) while generating tests for one model (the number of tests already generated is on the x-axis). By maximizing the coverage of tuples (dotted line), the test generation covers more tuples at the beginning but at the end it needs new tests to cover the residual uncovered tuples. By using compatibility and by weighting the tuples (continuous line), the algorithm covers fewer tuples at the beginning but at the end all the residual tuples are easily covered with few tests. The figure shows that the problem of finding minimal test suites is not easily solvable by using pure greedy algorithms, since only near the end the our proposed approach outperforms the classical greedy approach.

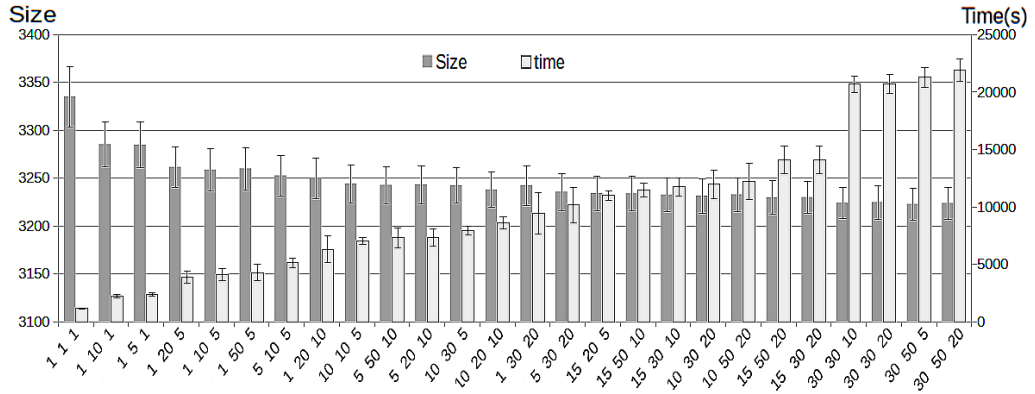


Figure 6.9: Test suite *size* and *time* depending on the repetitions settings ($repeat_{\min}$ $repeat_{\max}$ $repeat_{\text{better}}$).

6.5.3 Number of repetitions

Regarding the number of repetitions (options $repeat_{\min}$, $repeat_{\max}$, $repeat_{\text{better}}$ introduced Sect. 6.4.2), the situation is more clear, since the use of these options is purely incremental and increasing the number of tries will always increase the time and decrease (or keep equal) the number of tests. The choice of the optimal values for these options, is however a typical multi-objective optimization, in which we try to optimize the two conflicting objectives of a small test suite size *and* a small generation time. We test for the *repeat* options the values $\{1, 5, 10, 15, 20, 30, 50\}$ which give rise to 27 valid configurations. The data for the execution of all the configurations is shown in Fig. 6.9 (and later in Tab. 6.11). The graph confirms that the two objectives of minimizing both *size* and *time* are conflicting: it is possible to obtain smaller test suite but at the expense of the test generation time. Our technique allows the tester to decide of spending more time in order to have smaller test suites. To give users guidance in the choice of the best values for these options, we try to model the behavior of our algorithm by hypothesizing the following law:

$$size = minSize + \frac{k}{\log(time - minTime)}$$

saying that the value of *size* is a constant, *minSize* which is the minimum size that MEDICI could find in an unlimited amount of time, plus a quantity that decreases by increasing *time*. *minTime* is the minimum amount of time that MEDICI would take to generate a test suite without any retry and *k* is a constant representing the loss of efficiency in size in this case ($time =$

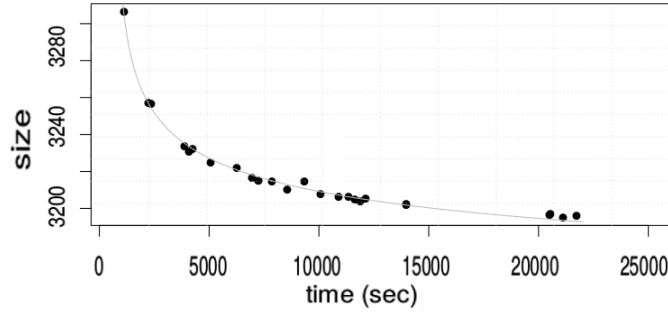


Figure 6.10: Test suite size *size* and time *time* depending on the number of repetitions

	<i>size</i>	$\Sigma\sigma S$	<i>time</i>	$\Sigma\sigma T$
ACTS	3387.5	0.5	73.7	2.4
CASA	3185.4	4391.2	14781.2	14305.9
MEDICI	3214.4	6633.5	7871	965.4

Figure 6.11: Comparison with ACTS and CASA

minTime). To find the constant values we have applied nonlinear regression analysis using the statistical tool R by minimizing the sum of squares of the distances. R finds the values $minTime = 740$, $minSize = 3024$ and $k = 1693$, which originate the gray line drawn in Fig. 6.10. Since the line approximates sufficiently well the behavior of the algorithm, a tester can use the proposed law, once he/she has executed some tests and estimated the constants, the costs and the benefits of further test generation time.

From all the configurations, we select one configuration with $(repeat_{min}, repeat_{max}, repeat_{better})$ equal to $(10, 30, 5)$ which represents a good compromise between time and speed and it can be considered as a good candidate for a default use of MEDICI. From now on, we will use this version for further comparison.

6.5.4 Comparison with other tools in CITLAB

We perform a comparison with the other external tools supported by CITLAB, namely ACTS [1,57] and CASA [21,38] to check if MEDICI actually improves over the current state of combinatorial tools. ACTS is a tool developed by

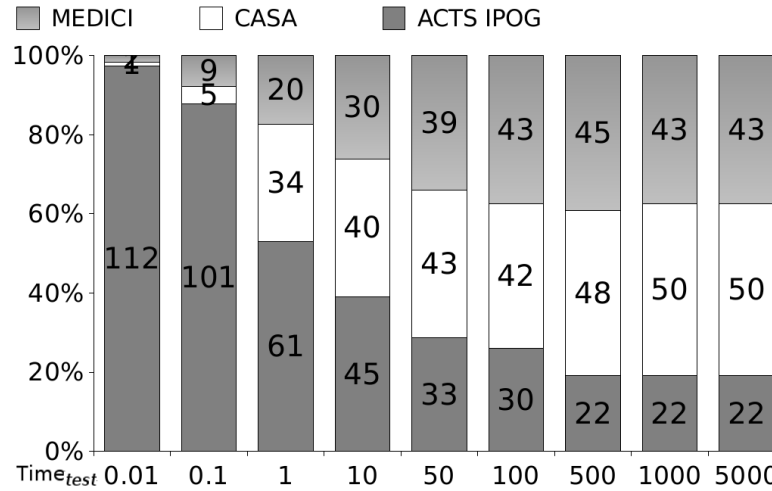


Figure 6.12: Number of models that present the minimum cost for each generator for $time_{test}$ from 0.01 to 5000 secs.

the NIST and implements several variants of the In Parameter Order (IPO) strategy. CASA is a tool developed at the University of Nebraska and it is based on simulated annealing, a well-studied meta-heuristic algorithm. Both support constraints, are freely available, have a large user base, and are very often used in comparison studies. Using CITLAB allows us to perform all the experiments in a very controlled environment on the same computer and using exactly the same examples.

Due to the high number of models and experiments we can give only some cumulative results. Table 6.11 reports the results of the comparison: we have computed the mean, and the standard deviation (σ) of the size and time (in secs) among all the 115 runs for every model. Besides the sum of averages (*size* and *time*), the table displays the sum of the standard deviations.

Table 6.11 shows that ACTS is the fastest but it produces also the biggest test suites. ACTS has a deterministic algorithm and hence the standard deviation of its sizes is null. MEDICI is always slower than ACTS but it produces smaller test suites. MEDICI is around 200 times slower than ACTS, but it produces a test suite on the average 5.4% smaller than ACTS. On the other hand, CASA is the slowest of all, but it produces rather small test suites. CASA has a very high standard deviation both in time and in size (running CASA only once may not lead to the best solution of its). MEDICI is faster than CASA and it has a smaller standard deviation. CASA produces a test suite on the average 1% smaller than MEDICI, but its generation time is, on average, double that of MEDICI.

Overall, we can say that MEDICI performances are between CASA and

ACTS. To better guide the user in the choice of the best test generator tool, we can roughly estimate the cost of testing (*cost*) as the total time for test generation ($time_{gen}$) plus test execution, which depends on the size of the test suite (*size*) and time necessary to execute every single test ($time_{test}$): $cost = time_{total} = time_{gen} + size \times time_{test}$. Using the data previously computed, we have also calculated the *cost* for each model and for each generator selecting a meaningful set of $time_{test}$. Fig. 6.12 shows the number of models that present the minimum average *cost* for each generator varying the $time_{test}$. ACTS outperforms both CASA and MEDICI if each test takes on average less than 10 seconds. This is in line with what was found by Garvin et al. [37]. If the time for executing a single test increases, CASA and MEDICI cost less than ACTS in most models. Even for very costly test execution (e.g. tests that require some human intervention), MEDICI can still compete with CASA in a meaningful number of models.

6.5.5 Threats to validity

We have identified some threats to validity of the proposed study and we present some countermeasures we have employed. First, the benchmark data may be not representative. We have tried to collect models from many sources: to the best of our knowledge this is one of the biggest benchmark set of constrained combinatorial models used for test generation. The models represent a wide heterogeneous range of real life and academic models. Second, we are aware that our tool, MEDICI, may produce incomplete and incorrect test-suites that allow it to perform better than the other tools. To avoid this, besides performing unit testing we have used the CITLAB “validator” [4] that checks that the resulting test suite actually cover all the required tuples (except those infeasible). We use this program for debugging MEDICI. In order to have confidence of the data obtained in the experiments, we have executed 50 runs for every configuration. Using multi-threads allows us to reduce the experimental time, but it may alter the running time, since an ordinary user will generally launch only one execution at the time. However, we believe that the comparison is still fair because we have treated all the generators in the same way.

6.6 Related work

Combinatorial interaction testing has been an active area of research for many years. In a recent survey [63] Nie and Leung count more than 12 research groups that actively work on CIT area and many other groups and tools

are missing in the count. In a previous survey, Grindal et al. [40] presented 16 different combination strategies, covering more than 40 papers. There are several web sites listing tools and approaches (like [66]), and publishing benchmarks and evaluations of tools and algorithms. The most studied area in CIT is the test suite generation, where several research groups continuously challenge existing algorithms and tools in order to provide better approaches in terms of execution times, supported features, and minimality of the produced test suites. Finding an algorithm that improves over the current state of the art has become a hard research task.

There are several families of CIT test generation tools, including bio-inspired, algebraic, logic-based [16], and greedy. In [13], Bryce et al. presented a general framework of *greedy* construction algorithms, in order to study the impact of each *type* of decision on the effectiveness of the resulting heuristic construction process. To this aim, they designed the framework as a nested structure of four decision layers, regarding respectively: (1) the number of instances of the process to be run, (2) the size of the pool of candidate rows from which select each new row, (3) the factor ordering selection criteria and (4) the level ordering selection criteria. The approach presented in this work fits exactly in the *greedy* category of algorithms modeled by that framework, and it is structured in order to be parametric with respect to the desired number of repetitions and the factor and level ordering strategies. Note that their study concluded that factor ordering is predominant on the resulting test suite size, and that *density*-based level ordering selection criteria was the best performing one out of those tested. In the present work, we explored original ways of redefining the *density* concept. In fact, while Bryce et al. compute it as the expected number of uncovered pairs, we weight tuple compatibility and we order parameters accordingly.

Comparison with BDD-based tools. Regarding the data structure we use, a comparison can be done with works using for CCIT binary decision diagrams (BDDs) which are similar to MDDs. Salecker et al. [72] developed a test set calculation algorithm which uses BDDs as efficient data structure to represent the combinatorial interaction testing problem with constraints. Both their and our approach are based on the modeling of the combinatorial interaction test problem with constraints as a single propositional logic formula. MDDs are a more efficient data structure for CCIT than BDDs: while modeling CCIT using BDDs requires a logic subformula corresponding to all possible alternatives for selecting values from each parameter P_i , MDDs permit to avoid the representation of these subformulas for single parameters; the benefit produced by this technique is the absence of the implicit

constraints introduced to represent value selection. Unfortunately the tool presented in [72] is not available and a fair comparison is difficult. For sanity check, we found that on the same models presented in [72], MEDICI without repetitions was able to produce a smaller test suite (486.2 vs 547) and the time required in [72] was 2.3 times the time for MEDICI (687 vs 1606 secs), although our PC is only 1.8 times faster (considering the SPECint of around 42.6 vs 23.5).

Segall et al. [74] developed FoCuS, another BDD-based CCIT tool. In their approach each parameter is represented by one or more binary variables in the BDD. In order to build the BDD of valid tests, they first built for each constraint (called *restriction*) the BDD representing the set of tests allowed by it. A test is valid if and only if it is valid according to all restrictions, therefore the set of valid tests is exactly the intersection of the sets of tests allowed by the restrictions. This is computed by the conjunction of the BDDs representing these sets. Their approach is therefore very similar to ours in terms of problem representation, and we believe that also their approach would benefit from the use of MDDs instead of BDDs. Unfortunately FoCuS is not publicly available. However, again for sanity check, we found that on the same models presented in [74] MEDICI produced smaller test suites (923.5 vs 934) while published data for FoCuS do not include generation time.

6.7 Future work and Conclusions

We plan to work in several directions in order to improve our approach and the tool. MEDICI (as most other test generation tools, with the notable exception of ACTS) does not support constraints containing arithmetic expressions. CITLAB already adopts the language of propositional logic with equality and arithmetic to express constraints. To be more precise, it uses propositional calculus, enriched by the arithmetic over the integers and enumerative symbols. Although arithmetic expressions are quite rare in models published in the literature, we plan to extend MEDICI in order to deal with the arithmetic constraints expressed in CITLAB, since we believe that industrial studies often use them.

Moreover, we have experimented only pairwise coverage, even if MEDICI, ACTS, and CASA support n-wise coverage. Initial experiments shows that MEDICI performs well also with n-wise coverage. Overall, we believe that the technique presented in this chapter and implemented in a prototype tool is a viable alternative to other commonly used tools for tests generation of combinatorial tests in the presence of constraints. Our techniques exploits an efficient data structure (MDDs) that proved to be suitable to represent

and solve constrained combinatorial models and promise to scale better than BDDs [41]. We have also devised several optimizations, like *weighting*, that combined with a classic greedy approach allow us to obtain very good results, as demonstrated by our experiments. The use of the framework CITLAB has allowed us to define a wide body of benchmarks and to perform the comparison with other tools in a simple and fair way.

7

CIT Validation process

In Combinatorial Interaction Testing, models specify a set of parameters with associated domains, and some constraints over the parameters. Test generation tools produce, starting from these models, test suites for achieving some given coverage criteria. The validation of both the models and the produced test suites is a worthwhile activity. Validating the models permits to early discover possible defects in them, to feed the test generations tools with *good* inputs and, possibly, to improve the quality of the testing process. Validating the produced test suites, instead, permits to check if the test generation tools are correct and to judge their quality. This section proposes to validate the models by checking that the constraints are consistent, that there is no constraint implied by the other constraints, and that the parameters and their values are really necessary. The proposed test suite validation, instead, consists in checking that the tests respect the type definitions and the constraints, that all the test requirements are covered, and that all the tests in the test suite are valid and necessary. For every error we propose a possible technique able to identify the potential causes and to suggest fixes for those problems. Experiments show that the targeted defects are widespread both in benchmark and real-life models.

7.1 Background work

For different programming languages, several tools automatically look for common errors as, for example, FindBugs, PMD and Checkstyle for Java, or Splint for C¹. These tools look for erroneous code but also for *stylistic conventions* violations that may indicate a possible problem. For example, the pattern *Unwritten field* of FindBugs signals if a field has never been written and always returns its default value: the violation of this pattern could show that the field is not necessary or that it must be updated somewhere.

A model review technique has been developed for the Software Cost Reduction (SCR) method [47], a requirements specification method that uses a tabular notation to define mathematical functions. There are different tables: *condition*, *event*, and *mode transition* tables. Each table describes a *variable* or a *mode* as a function of modes and/or events and/or conditions. The authors defined a *formal requirements model* specifying the properties that any SCR specification must satisfy, and developed a tool, the *consistency checker*, for checking these properties. They identified eight categories of properties: *Proper Syntax*, *Type Correctnesses*, *Completeness of Variable and Mode Class Definitions*, *Initial Values*, *Reachability*, *Disjointness*, *Coverage* and *Lack of Circularity*. Some properties are similar to ours. For example, *Coverage* requires that at least one condition in each row of a condition table must be true; this is similar to our consistency check.

The UML state machines are an object-based variant of Harel statecharts. In [69] the authors present a set of rules that seek to avoid common types of errors by ruling out certain modeling constructs for UML state machines or Statecharts. The authors state that the first rules that must be respected are the UML *well-formedness rules*. These rules are expressed as OCL constraints over UML models; the satisfaction of these constraints assures the syntactical correctness, which is a prerequisite for executing more complex checks. An example of *well-formedness rule* is the rule *CompositeState-1* that states that *a composite state can have at most one initial vertex*. The authors then reviewed different style guides proposed for statecharts and their dialects. They devised two categories of rules. *Syntactical Robustness Rules* identify syntactical constructions that, although syntactically correct according to the *well-formedness rules*, should be avoided because they could produce misleading models. *Semantic Robustness Rules* try to detect incorrect model behaviours, e.g., race conditions. Some syntactical robustness rules map to our checks. For instance, rules *MiracleState* and *Connectivity*,

¹<http://findbugs.sourceforge.net/>, <http://pmd.sourceforge.net/>,
<http://checkstyle.sourceforge.net/>, <http://www.splint.org/>

requiring that each state is reachable, are similar to our useless parameter and value check since they all require model minimality.

In [2] a model review technique is proposed for Abstract State Machines (ASMs), an extension of Finite State Machines. Seven meta-properties have been devised for checking the consistency, the completeness and the minimality of ASM models. Some of these meta-properties inspired our current work. For example, a meta-property requires that *every controlled function can take any value in its co-domain*: this meta-property is similar to the control we do to check that every parameter value is useful (see Section 7.3.3).

A model review technique has been developed also for models of the NuSMV model checker [3]; the authors identified ten meta-properties for checking the consistency, the completeness and the minimality of NuSMV models. We have been inspired also by some of these meta-properties. One meta-property checks that the temporal properties of the NuSMV specification are not vacuously satisfied: in a similar way, we check that the constraints are not totally/partially vacuous. Note that, however, our definition of vacuity is slightly different from the classical definitions [7, 55].

Test suite reduction has been extensively used in regression testing. Chavatal [22] proposes the use of a greedy heuristic that selects at a time a test case that covers most yet to-be-covered requirements, until all requirements are satisfied. Our algorithm in Alg. 8 is an instantiation of that proposed by Chavatal. Harrold and colleagues [43] propose a similar, but improved heuristic that generates solutions that are always as good or better than the ones computed by Chavatal. Their technique selects a representative set of test cases from a test suite that provides the same coverage as the entire test suite. This selection is performed by identifying, and then eliminating, the redundant and obsolete test cases. We plan to translate Harrold’s algorithm also for combinatorial test suites.

7.2 Notation

7.2.1 Using Logics and SAT/SMT solvers for CIT problems

In this section, in order to analyze combinatorial models and tests, we adopt a logic-based approach.

Definition 1. Test A test $t = (v_1, \dots, v_m)$ is an assignment of values to all the parameters of the combinatorial problem that respects the type definitions, i.e., $\forall i \in \{1, \dots, m\}: v_i \in D_i$. Let $D = D_1 \times \dots \times D_m$ be the domain of the tests, i.e., $t \in D$.

Definition 2. A test t is a model for a formula φ if it makes the formula φ true, formally $t \models \varphi$.

Definition 3. Constraint A constraint c_i is a formula over some parameters of the combinatorial problem.

Definition 4. Test validity A test t is *valid*, if t is a model of the constraints, i.e., $t \models \bigwedge_{i=1}^n c_i$.

Given a formula φ over the parameters in P , there are several decision problems regarding the truth evaluation of φ .

Definition 5. Satisfiability A formula φ is satisfiable if there is model for it. Formally, $\exists t \in D: t \models \varphi$.

Satisfiability can be proved with a SAT/SMT solver. There are some formulas that are always true, regardless the test.

Definition 6. Validity A formula φ is valid if and only if it is true under every interpretation, i.e., every possible assignment to the parameters makes φ true. Formally, $\forall t \in D: t \models \varphi$, or briefly $\models \varphi$.

To prove validity one can use a satisfiability solver, thanks to the following theorem.

Theorem 7. *A formula φ is valid if $\neg\varphi$ is not satisfiable.*

In combinatorial testing, test requirements are given as a set of tuples which are to be covered by the tests. Each tuple can be represented as a formula in a straightforward way by a conjunction of equalities. For instance, the pair $(p_i = v_j, p_k = v_h)$ can be represented as $p_i = v_j \wedge p_k = v_h$. A test *covers* a test requirement, thus a tuple tp , if it makes tp true. A satisfiability solver can be used also to check if a test requirement is feasible.

Definition 8. A test requirement tp is feasible if $tp \wedge \bigwedge_{i=1}^n c_i$ is satisfiable.

We exploit, whenever necessary, a Satisfiability Modulo Theories (SMT) solver, namely Yices [32], for representing and solving the formulas derived from combinatorial problems and tests. An SMT problem is a decision problem for logical formulas with respect to combinations of background theories expressed in classical first-order logic with equality. Yices can easily deal with the CITLAB models. An SMT instance is a generalization of a boolean SAT instance in which various sets of variables are replaced by predicates from a variety of underlying theories. Obviously, SMT solvers provide a much richer modeling language than that provided by SAT solvers. We have embedded Yices in CITLAB using JNA (Java Native Access) and we exploit the following commands of the Yices APIs (besides those for creating domains and variables):

- `mk_context` creates the logical context.
- `assert` asserts a constraint in the logical context. After one assertion, the logical context may become inconsistent.
- `push` creates a backtracking point. The logical context can be viewed as a stack of contexts.
- `pop` backtracks, i.e., it restores the context from the top of the stack, and pops it off the stack.
- `check` checks if the logical context is satisfiable.
- `del_context` deletes the logical context.

In order to check the satisfiability of a formula in Yices, we add the formula to the logical context by the command `assert` and then we execute the command `check`. Whenever we need to check the satisfiability of a set of formulas having a common subset, we can do, in order to decrease the computation time, an incremental checking by exploiting the commands `push` and `pop`.

We use the SMT solver not only as satisfiability checker, but also as an equivalence prover, thanks to Thm. 7.

7.2.2 Desired properties of combinatorial models and tests

In this section we introduce some properties that should be proved in order to assure that a combinatorial model and tests have some quality attributes. These properties refer to attributes that are defined independently from the particular combinatorial specification to be analyzed and they should be true in order to guarantee a certain degree of quality for the combinatorial model and tests. For this reason, we call them *meta-properties*. The violation of a meta-property always means that a quality attribute is not met and may indicate a potential/actual fault in the model or in the tests. Although we will actually define the meta-properties in the following sections, we introduce now three generic categories of quality attributes.

- **Consistency** requires that there are no elements that conflict with each other. For instance, the constraints should not be contradictory (see Sect. 7.3.1).
- **Completeness** requires that every feasible requirement must be covered by at least one test (see Sect. 7.4.1).
- **Minimality** guarantees that the specification does not contain elements defined or declared in the model but never used. These defects are also known as *over-specification*. For example, if a parameter value is never used, it could be removed from the parameter domain (see

Sect. 7.3.3). Another minimality meta-property checks that the test suite is *minimal* (see Sect. 7.4.2).

7.3 Validation of CIT models

7.3.1 Inconsistent constraints

In classical deductive propositional logic, a theory is consistent if it does not contain a contradiction. A simple syntactic contradiction happens when the theory contains both the formula φ and its contradiction $\neg\varphi$. In general, we consider a theory (semantically) consistent if and only if it has a model, i.e., there exists an interpretation under which all formulas in the theory are true. In brief, the theory is satisfiable.

We can extend the concept of consistency to combinatorial models with constraints. Given a set of parameters together with their domains, we can check if the constraints over these parameters are consistent, i.e., if they actually allow at least a possible valid assignment to every parameter of the model. An inconsistent set of constraints restricts too much the problem space to the point that no solution is possible.

Definition 9. A set of constraints $C = \{c_1, \dots, c_n\}$ is consistent if $\bigwedge_{i=1}^n c_i$ is satisfiable. A model is consistent if its constraints are consistent.

Example 1. An example of inconsistent set of constraints is $\{a \wedge \neg b, a \rightarrow b\}$.

In order to discover if a model is consistent, we use the SMT solver Yices by simply checking the satisfiability of the conjunction of the constraints.

How to deal with inconsistent constraints

Once the model has been proved inconsistent, the designer is interested in identifying which constraints are responsible for such inconsistency. In the simplest case a *single* constraint c_i is inconsistent by itself, i.e., it is a contradiction ($\models \neg c_i$). Single inconsistencies must be identified and removed (or corrected).

Example 2. For example, the constraint reported below is inconsistent.

```
# a=5 and a=6 #
```

Algorithm 4 Algorithm for finding a maximum consistent subset

Require: an inconsistent set of constraints C

Ensure: it returns a consistent subset of C

```

for  $i = (|C| - 1), \dots, 1$  do
  for all  $\{C' \subset C : |C'| = i\}$  do
    if  $isConsistent(C')$  then
      return  $C'$ 
    end if
  end for
end for
return  $\emptyset$ 

```

Identifying single inconsistencies using Yices is easy: it is sufficient to check each constraint for satisfiability.

However, in most cases there is not a single inconsistent constraint, but the inconsistency derives from the interaction of the constraints. In this case, the designer may be interested in finding a maximum subset of consistent constraints.

Definition 10. Given an inconsistent set Γ of constraints, we say that Ω is a *maximum consistent* (or *satisfiable*) subset (MCSS) of Γ , if $\Omega \subset \Gamma$ and every subset Δ (such that $\Omega \subset \Delta \subseteq \Gamma$) is inconsistent.

Finding the MCSS can be done by a greedy algorithm, as the one shown in Alg. 4. It checks for consistency all the proper subsets of C , going from the biggest ones to the singletons. As soon as a consistent subset C' is found, it is returned. The proposed algorithm has the advantage of returning a consistent subset, but does not precisely identify the causes of the original inconsistency: we can only know that the constraints in $C \setminus C'$ are inconsistent with at least one of the constraints in C' . In order to exactly discover the constraints responsible for the inconsistency, one should use an algorithm for finding the *minimum unsatisfiable core* [60], i.e., the smallest set of constraints that is still unsatisfiable.

Example 3. Let's consider the inconsistent set of constraints $C = \{a \wedge \neg b, a \rightarrow b, a \vee b\}$. The algorithm first selects the subsets of size $|C| - 1$; the subset $C_1 = \{a \wedge \neg b, a \rightarrow b\}$ is still inconsistent, subsets $C_2 = \{a \wedge \neg b, a \vee b\}$ and $C_3 = \{a \rightarrow b, a \vee b\}$, instead, are consistent. The algorithm returns the first subset found consistent (C_2 or C_3); note that the two subsets are not equivalent: C_2 admits only $\{a = true, b = false\}$, while C_3 admits two tests different from the one admitted by C_2 . The modeler should check if the returned set actually captures the intended requirements of the system.

7.3.2 Constraints Vacuity

In this section we extend the notion of vacuity to combinatorial constraints. Generally, vacuity has been applied to properties of behavioral models in formal verification. A property is vacuously satisfied if that property is satisfied and proved true regardless of whether the model really fulfills what the specifier originally had in mind or not. For example, the property $a \rightarrow b$ is vacuously satisfied by any model where a is never true. Vacuity is an indication of a problem in either the model or the property. Several techniques to detect vacuity have been proposed (e.g., [7, 55]) and also tools that perform vacuity detection have been developed (e.g., [39]). In classical formal verification, to detect vacuity as defined in [7, 55], it is enough to replace parts of the property and see if the replacement has any effect on the result of the verification. A common technique to detect vacuity [55] consists in replacing a subformula ϕ of property φ with *true* or *false* (depending on the polarity of ϕ in φ) and checking for its satisfiability.

In our case, however, the constraints do not represent properties that can be derived from the model, but the model itself, i.e., the whole and only formal specification, and the classical definitions do not apply. We still borrow the term *vacuity* to indicate a constraint or one of its subformulas which is useless. Intuitively, a constraint, or a part of it, is vacuous if it can be removed, either because it is always true or because it is implied by the other constraints. We distinguish between total vacuity and partial vacuity.

Definition 11. A constraint c_i is *totally vacuous* iff $\models \left(\bigwedge_{k \in \{1, \dots, n\} - \{i\}} c_k \right) \rightarrow (c_i \equiv \text{true})$.

Intuitively, a constraint is totally vacuous if it is implied by the other constraints which make the constraint useless, since it does not add any further restriction to the model.

Example 4. Let's consider the set of constraints $C = \{\neg a, a \rightarrow b\}$. The constraint $a \rightarrow b$ is totally vacuous. Indeed, $\neg a \rightarrow (a \rightarrow b)$ is valid.

Tautologies used as constraints are a special case of total vacuity.

Example 5. Let's consider the set of constraints $C = \{a \wedge b, c \vee \neg c\}$. The constraint $c \vee \neg c$ is totally vacuous, since it is a tautology.

A constraint could be partially vacuous, when it contains an occurrence of a subformula ϕ which is useless, i.e., it is implied by the other constraints and by the other part of the same constraint. Therefore ϕ could be removed in that occurrence.

Algorithm 5 Given a predicate R , REDUCE returns the set of all the formulas obtained from R by removing one occurrence of a subformula in R

```

function REDUCE( $R$ )
  if  $R$  is atomic then
    return  $\emptyset$ 
  else if  $R = A \circ B$  then
     $ra \leftarrow$  REDUCE( $A$ )
     $rb \leftarrow$  REDUCE( $B$ )
    return  $ra \circ B \cup rb \circ A \cup \{A, B\}$ 
  else if  $R = \neg A$  then
    return  $\neg$ REDUCE( $A$ )
  end if
end function

```

where

\circ is \vee or \wedge

$\{x_1, \dots, x_n\} \circ R = \{x_1 \circ R, \dots, x_n \circ R\}$, $\emptyset \circ R = \emptyset$

$\neg\{x_1, \dots, x_n\} = \{\neg x_1, \dots, \neg x_n\}$, $\neg\emptyset = \emptyset$

In order to discover if a constraint c_i is partially vacuous, we generate, by the function REDUCE reported in Alg. 5, all the possible formulas that can be obtained from c_i by removing only an occurrence of one of its subformulas². Partial vacuity is checked using the following definition.

Definition 12. A constraint c_i is partially vacuous only if there is $\varphi \in \text{REDUCE}(c_i)$ such that $\models \left(\bigwedge_{k \in \{1, \dots, n\} - \{i\}} c_k \right) \rightarrow (c_i \equiv \varphi)$.

If formula in Def. 12 is true, then φ is equivalent to c_i (assuming all the other constraints). Using φ instead of c_i would give an equivalent simpler model.

Example 6. Let's consider the set of constraints $C = \{c_1, c_2\}$ with $c_1 = a \wedge b$ and $c_2 = (a \vee b) \wedge d$. $\text{REDUCE}(c_2) = \{a \wedge d, b \wedge d, a \vee b, d\}$. The constraint c_2 is partially vacuous because it is equivalent to d : indeed, $c_1 \rightarrow (c_2 \equiv d)$ is valid. Note that the whole constraint is not vacuous: indeed, $c_1 \rightarrow (c_2 \equiv \text{true})$ is not valid.

In order to check a constraint for total vacuity in Yices, we check the formula introduced in Def. 11 for validity. For checking partial vacuity of constraint c_i , we check, for every formula in $\text{REDUCE}(c_i)$, the validity of the formula introduced in Def. 12. Note that one may stop as soon as she/he finds a φ in $\text{REDUCE}(c_i)$ which makes the formula of Def. 12 true.

²The algorithm can be easily extended to deal with other boolean operators as \rightarrow , \leftrightarrow , and \oplus .)

How to deal with vacuous constraints

The vacuity of a constraint may actually be caused by an error in it. The user may have mistyped an element or an operator and this may cause the vacuity. Instead, if the constraint is correct, it can be eliminated or simplified in order to make the model simpler (and possibly the test generation faster). When a single constraint is totally vacuous, it can be eliminated without problems; note that, after its removal, the vacuity should be checked again, since some of the other vacuous constraints may have become not vacuous. When a constraint c_i is partially vacuous several times, i.e., there exist φ_1 and φ_2 in $\text{REDUCE}(c_i)$ which are equivalent to c_i , c_i can be substituted by either φ_1 or φ_2 and its vacuity must be checked again.

In some cases, however, the user may be interested in keeping also totally vacuous constraints as further properties of the system. It is common in formal modeling having, stated in the model, properties which are implied by the assumptions or axioms asserted in the model. Indeed, a vacuous constraint represents a property of the system under test which is implied by other constraints. So, it should be classified as **property** and not as constraint. We plan to add to the CITLAB language also a notation for properties, together with the support for proving them.

7.3.3 Useless parameter values and useless parameters

A parameter p can contain in its domain some values which are never taken by p .

Definition 13. The value v_k^j of a parameter p_j is *useless* if, due to the constraints, p_j can never assume value v_k^j .

Definition 14. If the parameter p can assume only a value, then the whole parameter is *useless*.

We consider such elements useless, since they can be ignored during the test generation process.

In order to discover if the value $v_k^j \in D_j$ of parameter p_j is useless, we check with Yices if $p_j = v_k^j \wedge \bigwedge_{i=1}^n c_i$ is unsatisfiable. Totally, we must check $\sum_{j=1}^m |D_j|$ values. Once we know which parameter values are useless, we can also identify the useless parameters. Indeed, given a parameter p , if all its values except one are useless, then p can be classified as useless. Note that, if the model is consistent, each parameter can take at least one value. Uselessness checking should be done only on consistent models.

```
Model uselessModel
Parameters:
  Enumerative a {a1 a2 a3};
  Enumerative b {b1 b2 b3};
end
Constraints:
  # a == a.a1 #
  # b != b.b1 #
end
```

Figure 7.1: Example of useless parameter and useless parameter value

How to deal with useless elements and parameters

Uselessness of parameters and values can be caused by errors in the constraints: the test designer may have inadvertently introduced a restriction not present in the real system under test. In this case, the constraints should be revised. If this is not the case, the useless parameters and values can be removed from the model. However, if they are contained in some constraints, also the constraints must be modified accordingly.

Parameters removal can ease the test suite generation process, since size reduction of a combinatorial model domain decreases significantly the generation time. If a useless parameter is removed, it may be reintroduced in the tests with its unique value.

Example 7. Consider, for instance, the CITLAB model in Fig. 7.1. Due to the first constraint ($a == a.a1$), parameter a can take only value $a1$. In this case, parameter a could be removed during the test generation process and, if necessary, inserted again in the test suite. Due to the second constraint ($b != b.b1$), instead, the domain of b can be reduced, excluding value $b1$. In both cases the corresponding constraint must be removed as well.

7.4 Validation of CIT test suites

Test suite validation checks that the test suites produced by a generation tool are correct (see Sect. 7.4.1) and minimal (see Sect. 7.4.2).

7.4.1 Test Suite Correctness

We introduce the following definitions.

1. A test suite is *sound* if every test is syntactically correct and valid:
 - (a) an assignment of values to the parameters is a *syntactically correct* test if it satisfies the type definitions;
 - (b) a test is *valid* if it does not violate any constraint (see Def. 4).
2. A test suite is *complete* if every feasible test requirement is covered.

Definition 15. Test suite correctness A test suite is *correct* if it is *sound* and *complete*.

Checking if a test suite is sound only requires syntax checking and the tests validity assessment. In order to assess if a test t is valid, it is enough to substitute the values of the parameters in t in each constraint c_i and check that every c_i evaluates to true.

Example 8. Let's consider the model with a parameter x defined in the domain $\{1, \dots, 100\}$ and the constraint $x > 10$. The following test suites are both not sound:

- $\{x = 101\}$: Although the test suite satisfies the constraint, it does not respect the type definition of x (the test is not syntactically correct).
- $\{x = 9\}$: The test suite does not satisfy the constraint (the test is not valid).

Checking the completeness of a test suite requires a satisfiability solver, since it is not possible to judge if a test requirement is feasible or not by syntax checking. The completeness check can be performed by Alg. 6. It checks if every tuple is covered by at least one test in TS . If a tuple tp is not covered, it checks its feasibility using the SMT solver and returns false if tp is feasible and not covered.

How to deal with incorrect test suites

An unsound test suite must be fixed before it can be used. There are two main ways: either discard any invalid test or modify it in order to make it valid. Removing an invalid test is easier than fixing it, but it may reduce the testing coverage. On the other hand, fixing a test requires a greater effort, since it is not clear in general which assignments in the test are responsible for its invalidity.

An incomplete test suite can still be useful, although it may not exercise the system under test as well as required. The tester may use a test generator tool that accepts an existing possibly incomplete test suite (often called *seeds*) and tries to generate the missed test cases. With a slight modification, Alg. 6

Algorithm 6 Test suite completeness check

Require: test suite TS to be checked**Require:** the domain D of the parameters**Require:** the required n -wise coverage**Ensure:** it returns *true* if TS is complete, *false* otherwise $TP \leftarrow$ all the n -tuples from D **for all** $tp \in TP$ **do** $covered \leftarrow false$ **for all** $t \in TS$ **do** **if** t covers tp **then** $covered \leftarrow true$ **break** **end if** **end for** \triangleright if not covered, check feasibility **if** $\neg covered$ **then** **if** $tp \wedge \bigwedge_{i=1}^n c_i$ is satisfiable **then** **return** *false* **end if** $\triangleright tp$ is infeasible **end if****end for****return** *true*

can also be used to measure the incompleteness of a test suite by counting the number of feasible tuples that are not covered. As in classical testing, coverage measures can be used to assess the quality of the test suite and of the tool that has generated it. Our approach would correctly count only the feasible test requirements.

7.4.2 Test Suite Minimality

Combinatorial test generators can produce very compact test suites, which, however, could still contain redundant tests. For example, tools generating at every step a test that covers still uncovered tuples, may generate at some point a test which might also cover several other tuples previously covered by tests and these previously generated tests may become useless. An optimum test case should satisfy two objectives simultaneously. First, it must satisfy the maximum number of uncovered requirements (tuples). Second, it must have the minimum overlap in requirements coverage with other test cases. The smallest ideal test suite is that in which each tuple is covered by exactly one test case, but this very seldom can happen: in most cases, a tuple will be covered by many tests, creating possible redundancies. A certain level

of redundancy is in general unavoidable. However, some redundancies are useless: some tests that overlap may be eliminated without reducing the total coverage of the test suite. This problem is also known as test suite reduction or minimization [43].

First of all, we want to discover if a test suite could be reduced without losing coverage. We introduce the following definition.

Definition 16. A test suite TS is minimal if there exists no subset $TS' \subset TS$ such TS' satisfies all the testing requirements as the original set TS does, i.e., that all the tuples covered by TS are also covered by TS' .

How to recognize a non-minimal test suite? We can postulate that a test t is *redundant* if all the tuples covered by t are also covered by other tests. On the contrary, a test case can be defined *essential*, i.e., it cannot be removed from the test suite, as follows.

Definition 17. A test case t_i is *essential* if it covers at least one tuple tp in TP (the set of all the tuples for a given n -wise coverage) not covered by other test cases of the test suite TS . Formally, $\exists tp \in TP: (t_i \models tp \wedge (\neg \exists t_j \in TS: (i \neq j \wedge t_j \models tp)))$.

Theorem 18. A test suite TS is non-minimal iff TS contains at least a not essential test.

Alg. 7 reports the algorithm we use to check if a test suite is minimal. It checks if each test t of the test suite is essential, i.e., for every tuple tp it collects (in cov) the tests that cover tp . If cov contains only one test, then that test is essential and collected in *EssentialTests*. If all the tests are essential, then the test suite TS is minimal.

How to deal with non-minimal test suites

Test suite reduction (also known as test suite minimization) is often applied in the context of regression testing, when one wants to find a minimal subset of test cases which satisfy all the testing requirements as the original set does. The problem of finding the minimal test suite that satisfies a set of test goals can be reduced, in polynomial time, to the minimum set covering problem which is NP-hard. A simple greedy heuristic for the minimum set covering problem defined in [22] can be adapted to the test suite minimization.

In order to obtain a final test suite with fewer test cases, we try to build a reduced test suite in which the requirements coverage is preserved and all the test cases are essential. Note that, however, if the test suite is non-minimal, then one cannot simply remove all the not essential test cases, since

Algorithm 7 Test suite minimality check

Require: test suite TS to be checked**Require:** set of tuples TP **Ensure:** it returns *true* if TS is minimal, *false* otherwise $EssentialTests \leftarrow \emptyset$

```

for all  $tp \in TP$  do
   $cov \leftarrow \emptyset$ 
  for all  $t \in TS$  do
    if  $t$  covers  $tp$  then
       $cov \leftarrow cov \cup \{t\}$ 
    end if
  end for
  if  $|cov| = 1$  then
     $EssentialTests \leftarrow EssentialTests \cup cov$ 
  end if
end for
return  $|EssentialTests| = |TS|$ 

```

a not essential test case may become essential after another not essential test case is removed from the test suite. The choice of which tests to include in the final test suite is critical. We have implemented a greedy algorithm, reported in Alg. 8, which can reduce the original test suite, still covering all the requirements. At every step it chooses the test that covers most uncovered tuples. When there is a tie between multiple test cases, one test case is randomly selected.

Example 9. Let's consider the test suite $TS = \{t_1, t_2, t_3\}$ whose tests cover, respectively, requirements $\{a, b\}$, $\{a, c\}$ and $\{b, d\}$. If in the first iteration the greedy algorithm collects test t_2 , in the second iteration it must collect the test t_3 since it covers most uncovered tuples. At this point, all the tuples have been covered. So, the greedy algorithm reduces TS as $mTS = \{t_2, t_3\}$.

Note that the algorithm may fail to reduce the test suite, even if this test suite is non-minimal.

Example 10. Let's consider the same test suite shown in Example 9. If in the first iteration the greedy algorithm collects test t_1 , it must also collect both tests t_2 and t_3 in two following iterations. So the final test suite is not minimized, i.e., $mTS = TS$. However, as seen in Example 9, TS is non-minimal.

Algorithm 8 Reduction algorithm**Require:** test suite TS to reduce**Require:** set of tuples TP covered by TS **Ensure:** it returns a possibly minimal test suite

```

 $TS' \leftarrow TS$ 
 $mTS \leftarrow \emptyset$ 
while  $TP \neq \emptyset$  do
   $t \leftarrow \text{getMostCoveringTest}(TS', TP)$ 
   $CoveredTPs \leftarrow \text{getCoveredTPs}(TP, t)$ 
  if  $CoveredTPs = \emptyset$  then
    return  $mTS$ 
  end if
   $mTS \leftarrow mTS \cup \{t\}$ 
   $TS' \leftarrow TS' \setminus \{t\}$ 
   $TP \leftarrow TP \setminus CoveredTPs$ 
end while
return  $mTS$ 

```

7.5 Experimental results

As model set for CIT problems we have gathered a set of 64 models with constraints taken from the literature (CASA [25], FoCuS [74], ACTS [1], and IPOs [18]) and used (in subsets) also by many other papers. We have implemented the validation framework in CITLAB and we have performed experiments over models and their test suites generated with different tools. The benchmarks are available at the CITLAB web site³. We have performed the experiments on a Linux PC with an i7 processor 3770 (3.4 GHz) and 16 GB of RAM.

7.5.1 Consistency of constraints

The aim of this experiment is to determinate if some of the models under test present any inconsistent constraint. The constraints validation process performed over the 64 models took about 5.6 seconds and it proved that all the models have only consistent constraints. This result was expected since these models have been extensively used for test generation and no test can be generated from inconsistent models.

Table 7.1: Vacuous constraints

Model(# constr.)	# useless subform.	Vacuous constraints				
		partially (but not totally)		totally		Sum %
		#	%	#	%	
bench_01 (24)	7	4	16.7	2	8.3	25.0
bench_02 (22)	2	2	9.1	0	0.0	9.1
bench_03 (10)	0	0	0.0	1	10.0	10.0
bench_04 (17)	11	7	41.2	2	11.8	52.9
bench_05 (39)	9	0	0.0	1	2.6	2.6
bench_06 (30)	40	8	26.7	22	73.3	100.0
bench_07 (15)	20	3	20.0	11	73.3	93.3
bench_08 (37)	22	6	16.2	11	29.7	45.9
bench_09 (37)	51	13	35.1	24	64.9	100.0
bench_10 (47)	31	9	19.1	16	34.0	53.2
bench_11 (32)	11	5	15.6	6	18.8	34.4
bench_12 (27)	12	5	18.5	4	14.8	33.3
bench_13 (26)	17	9	34.6	4	15.4	50.0
bench_14 (15)	10	5	33.3	3	20.0	53.3
bench_15 (22)	3	3	13.6	0	0.0	13.6
bench_16 (34)	41	10	29.4	22	64.7	94.1
bench_17 (29)	0	0	0.0	1	3.4	3.4
bench_18 (28)	3	0	0.0	3	10.7	10.7
bench_19 (43)	6	2	4.7	0	0.0	4.7
bench_20 (48)	30	12	25.0	14	29.2	54.2
bench_21 (46)	65	12	26.1	32	69.6	95.7
bench_22 (22)	17	6	27.3	10	45.5	72.7
bench_23 (15)	13	4	26.7	7	46.7	73.3
bench_24 (29)	9	3	10.3	6	20.7	31.0
bench_26 (32)	16	3	9.4	11	34.4	43.8
bench_27 (20)	3	1	5.0	3	15.0	20.0
bench_28 (37)	6	1	2.7	3	8.1	10.8
bench_30 (35)	38	10	28.6	17	48.6	77.1
CommProt (128)	814	73	57.0	24	18.8	75.8
Concurr (7)	7	1	14.3	0	0.0	14.3
gcc (40)	6	2	5.0	0	0.0	5.0
HealthC2 (25)	14	2	8.0	0	0.0	8.0
ProcComm2 (125)	43	7	5.6	97	77.6	83.2
Services (388)	81	27	7.0	0	0.0	7.0
SmartHome (43)	33	16	37.2	17	39.5	76.7
Storage1 (95)	205	41	43.2	0	0.0	43.2
Telecom (21)	5	1	4.8	0	0.0	4.8

7.5.2 Vacuity detection

The aim of this experiment is to determine the presence of partially/totally vacuous constraints. The experiment took 45.5 seconds. Table 7.1 reports the 37 models that present at least a form of vacuity; for each model, we report in round brackets the number of its constraints. The second column reports the number of (occurrences of) subformulas that cause a partial vacuity in a constraint: such subformulas are *useless*. The next columns report the number and the percentage of constraints that are partially vacuous, but that are not also totally vacuous. Then, the number and the percentage of totally vacuous constraints are reported. Finally, the table shows the percentage of constraints that have any form of vacuity (partial or total). We can notice that there are more useless subformulas than partially vacuous constraints: indeed, a partially vacuous constraint can have more than one useless subformula. The number of totally vacuous constraints can be high (e.g., 77.6% for *ProcComm2*); however, not all the vacuous constraints can be removed at once. As explained in Section 7.3.2, in order to remove vacuity, one should remove a vacuous constraint at a time, and check for vacuity after each removal: indeed, after the removal of a vacuous constraint, (some of) the other vacuous constraints could become no more vacuous.

The constraints vacuity has proved to be a widespread problem. Almost all the benchmarks *bench_n*, that have been “randomly synthesized starting from real case studies” [25], manifest this problem. However, many real life models are affected too. In general, vacuity is difficult to detect by hand and a tool like that presented in this chapter is essential for discovering it.

7.5.3 Useless parameter values and useless parameters

The aim of this experiment is to determine the presence of useless parameters values and useless parameters. Table 7.2 shows the results. Checking useless parameters and values over the 64 models took about 6.2 seconds. We have found that 23 models have at least one useless parameter value and one useless parameter. As expected, we note that each useless boolean value corresponds to a useless parameter. Instead, a useless enumerative value does not necessarily imply that the corresponding parameter is useless.

The presence of so many combinatorial models with useless parameters has surprised us, since they have been extensively used in experiments in the literature. We discovered that most of them (*bench_n*) have been randomly synthesized. Nonetheless, they should be fixed in order to make them more plausible as real models and be considered suitable as benchmarks for

³<https://code.google.com/a/eclipselabs.org/p/citlab/>

Table 7.2: Useless parameter values and useless parameters

	Useless parameter values			Useless parameters			
	Bool	Enum	Total	Bool	Enum	Total	
						#	%
bench_01	2	0	2	2	0	2	2.06
bench_02	1	0	1	1	0	1	1.06
bench_04	2	1	3	2	0	2	3.45
bench_06	12	0	12	12	0	12	15.58
bench_07	4	0	4	4	0	4	13.33
bench_08	3	1	4	3	0	3	2.52
bench_09	10	4	14	10	0	10	16.39
bench_10	4	2	6	4	1	5	3.40
bench_11	3	0	3	3	0	3	3.13
bench_12	3	0	3	3	0	3	2.04
bench_13	5	1	6	5	0	5	3.76
bench_14	2	1	3	2	0	2	2.17
bench_15	1	0	1	1	0	1	1.72
bench_16	13	0	13	13	0	13	14.94
bench_20	8	1	9	8	0	8	5.06
bench_21	13	2	15	13	0	13	15.29
bench_22	7	0	7	7	0	7	8.86
bench_23	1	2	3	1	0	1	3.70
bench_24	2	0	2	2	0	2	1.68
bench_26	3	0	3	3	0	3	3.16
bench_30	8	0	8	8	0	8	10.13
ProcComm2	0	10	10	0	3	3	12.00
SmartHome	14	0	14	14	0	14	36.84

testing technique. One model (*ProcComm2*), however, claims to be a “real-life test space instance generated by or for our customers” [74]. In another case (*SmartHome*), useless parameters are present because the model has been automatically obtained from a feature model without applying any optimization [20].

7.5.4 Test suite validation

We have performed experiments over the test suite generated in 10 different runs with three different tools: ACTS [1], CASA [25] and MEDICI (an internal tool for test generation we are developing). The test suite used was the same of the previous experiment (64 models) and we chose to perform a pairwise generation. Results are shown in Table 7.3. The three tools produced complete and correct test suites for all the models. MEDICI and CASA al-

ways produced minimal test suites, while ACTS (which was the fastest in test generation) produced a non-minimal output for two benchmarks. We have applied 10 times the reduction algorithm presented in Alg. 8 to the test suites obtained in these two cases. In all the cases the reduction algorithm was able to reduce the test suite. In one case, however, the produced test suite was still non-minimal.

The time to check if all the test suites are minimal is 50 seconds, while the reduction of one test suite takes 0.8 seconds.

Table 7.3: Test suite validation

Tool	# test suites		
	Complete	Correct	Minimal
ACTS	64	64	62
MEDICI	64	64	64
CASA	64	64	64

7.6 Implementation in CITLAB

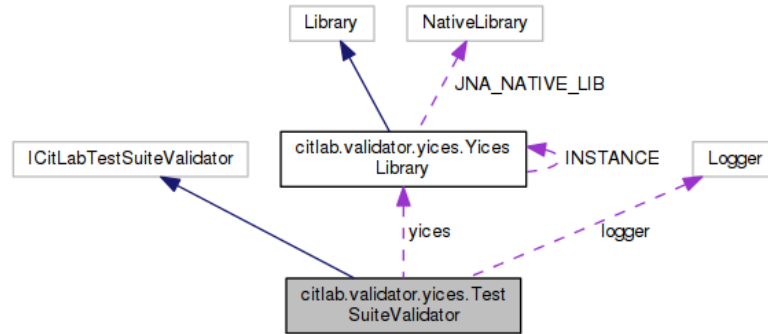


Figure 7.2: TestSuiteValidator UML

Fig. 7.2 and Fig. 7.3 show how the validation functionalities, described in the previous sections, are implemented as CITLAB plug-in. A plug-in exposes two classes in order to satisfy the requirements of the extension points describe in 5.1.4 and these classes dynamically invoke Yices native code using JNA and allowing users to validate both the models and the testsuites generated in the framework.

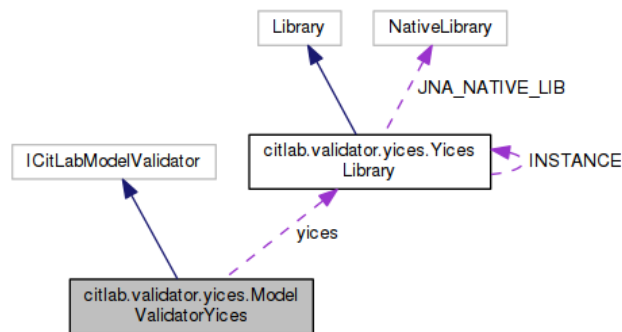


Figure 7.3: Model Validator UML

7.7 Conclusions

We have presented a set of quality checks that CIT models and test suites should pass: the constraints are consistent and do not contain useless parts, all the parameters and values are useful, test suites do not violate the constraints and cover all the testing requirements, and the final test suite does not contain tests that can be removed without loss of coverage. We have devised suitable techniques, based on an SMT solver, to perform these checks and established some policies about how to deal with violations of these properties.

We have identified the following use cases of our validation framework involving different kinds of users. The first set of users are the *clients* of combinatorial testing, i.e., those who apply combinatorial testing to real systems. Our validation framework helps *designers* to identify defects in models: an inconsistent or vacuous constraint and a useless parameter or value are often an evidence of a defect in the model. *Testers* can use our validation framework in order to assess the quality of the test suites they use. The test suite may require to be fixed and some tests may be discarded because they are wrong or useless. The measure of coverage can be used by testers to determine the quality of the testing process.

The second set of users are the *providers* of CIT tools and frameworks. By using our validation framework, researchers can check the quality of the models they use for benchmarking and experimenting their algorithms. Moreover, the validation framework can be used to validate (and eventually debug) new test generation algorithms, to see if the test suites they generate are actually correct and minimal. The presence of our validation component inside the proposed CIT framework CITLAB makes also the comparison and integration of test generation techniques more fair, since it guarantees a way to ensure that every tool embedded in CITLAB is producing correct results.

8

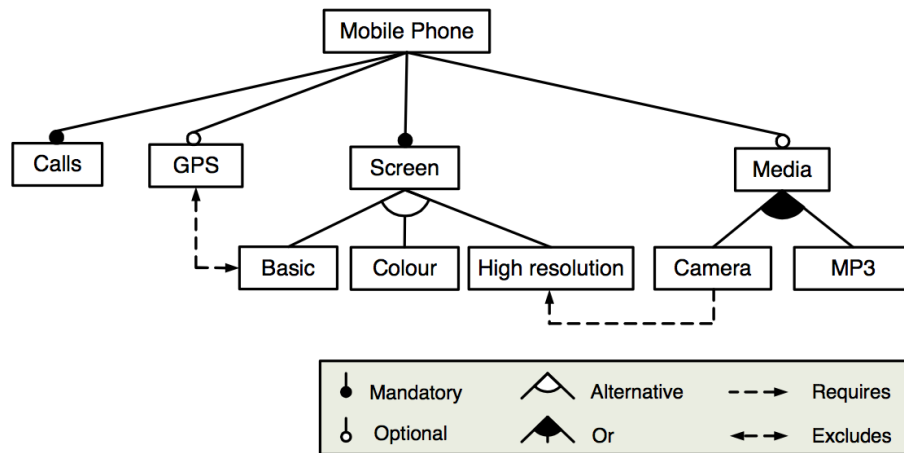
Feature models testing in CITLAB

Feature models are commonly used to represent product lines and systems with a set of features interrelated each others. Test generation from feature models, i.e. generating a valid and representative subset of all the possible product configurations, is still an open challenge. A common approach is to build combinatorial interaction test suites, for instance achieving pair-wise coverage among the features. In this chapter we show how standard feature models can be translated to combinatorial interaction models in our framework CITLAB, with all the advantages of having a combinatorial testing environment (in terms of a clear semantics, editing facilities, language for seeds and test goals, and generation algorithms). We present our translation which gives a precise semantics to feature models and it tries to minimize the number of parameter and constraints while preserving the original semantics of the feature model.

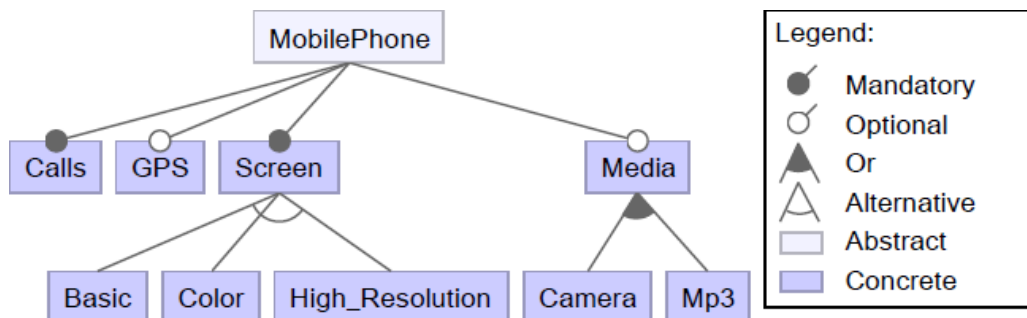
8.1 Background

In software product line engineering, feature models are a special type of information model representing all possible products of a software product line in terms of features and relationships among them. Specifically, a basic feature model is a hierarchically arranged set of features, where each parent-child relation between them may be one of the following types:

Mandatory – child feature is required.



(a) Conventional notation



GPS $\Rightarrow \neg$ Basic

Camera \Rightarrow High_Resolution

(b) FeatureIDE notation

Figure 8.1: Examples of Feature model notations

Optional – child feature is optional.

Or – at least one of the sub-features must be selected.

Alternative (xor) – exactly one of the sub-features must be selected

In addition to the parental relationships, cross-tree relations are allowed, to specify incompatibility or requirement kind of constraints between features, in the form:

A requires B – The selection of feature A in a product implies the selection of feature B.

A excludes B – A and B cannot be part of the same product.

Feature models can be visually represented by means of feature diagrams. Figure 8.1 depicts a simplified example model presented in [8] and inspired by the mobile phone industry, in order to present the visual notation commonly adopted for feature modeling. The example also shows how a model can be used to specify a product family, that is, to determine the features that will be supported (loaded) in a particular phone configuration of the considered family. According to the model, all phones must include support for calls, and displaying information in either a basic, color or high resolution screen. Furthermore, the software for mobile phones may optionally include support for GPS and multimedia devices such as camera, MP3 player or both of them.

Extensions to the basic feature model notation have been proposed in literature, e.g. allowing specifying the cardinality of the features and/or additional type of information. However, in our experiments, we consider only basic feature models.

8.1.1 Feature Modeling frameworks

Several languages/tools for specifying/analyzing feature models are currently available, some of them already mature enough to be part of a software production IDEs. In this work, FeatureIDE [79] has been used to design or import the models used in the evaluation section. FeatureIDE is an open-source framework for feature-oriented software development (FOSD) based on Eclipse. FOSD is a paradigm for the construction, customization, and synthesis of software systems. Code artifacts are mapped to features, and a customized software system can be generated given a selection of features. The set of software systems that can be generated is then a software product line (SPL).

8.1.2 Feature Model semantics

Feature models semantics can be rather simply expressed by using propositional logics as already done in [8]. Every feature becomes a propositional


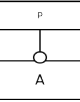
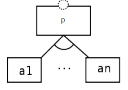
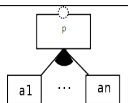
Type	Notation	Propositional formula
Mandatory		$p \rightarrow A \wedge A \rightarrow p$
Optional		$A \rightarrow p$
Alternative		$p \rightarrow alt(a1, a2 \dots an) \wedge a1 \rightarrow p \wedge a2 \rightarrow p \wedge \dots \wedge an \rightarrow p$
Or		$p \rightarrow (a1 \vee a2 \vee \dots \vee an) \wedge a1 \rightarrow p \wedge a2 \rightarrow p \wedge \dots \wedge an \rightarrow p$

Table 8.1: Conventional translation

letter or a Boolean variable, and every relationship among features becomes a propositional formula modeling the constraints about them as reported in Tab. 8.1¹.

Example 11. Note that in case of alternative features, the translation using Boolean variables introduces many variables which are mutually exclusive. For instance, the model in Fig. 8.2 introduces 8 Boolean variables for a system with just two main features (A and B), because each of them expands to four alternative leaf features $a_1 \dots a_4$ and $b_1 \dots b_4$. Several complex constraints are necessary to constraint valid products. In case of many alternative features, the complexity of the model would grow. Reducing their complexity would be a benefit to the user, which would be facilitated in the model comprehension, and would also allow the available tools to manage larger models.

Variability factor This index measures the ratio between the number of valid products and the total number of possible feature combinations, which is equal to 2^n where n is the number features. In case of Boolean variables, n is also equal to the number of variables. The feature model variability can be used to measure the flexibility of the feature model. For instance, a small factor means that the number of combinations of features is very limited compared to the total number of potential products [8].

¹Where the *alt* operator represents the exclusive or among all its arguments and it is defined as $alt(a1, a2, \dots, an) = (a1 \wedge \neg a2 \wedge \dots \wedge \neg an) \vee \dots \vee (\neg a1 \wedge \neg a2 \wedge \dots \wedge an)$.

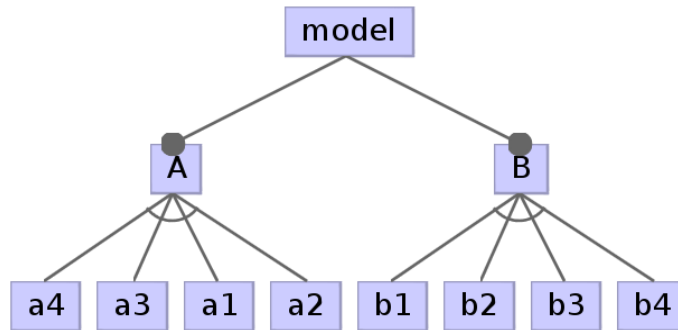


Figure 8.2: A two alternative model

8.2 Translation from FM to CitLab

In this section we present a procedure to translate feature models into combinatorial models. Our translation is performed in three steps:

- Every feature, starting from the root feature, is translated to an element (variable or literal constant) in the combinatorial problem.
- Additional constraints are added in order to represent relationships among features as specified by the hierarchies in the feature model.
- Cross-tree constraints are translated and added to the model.

Several simplifications can be applied to the final combinatorial model, but these will be presented in the next section.

8.2.1 Representation of every feature

During the first step, all the features (corresponding to all the nodes in the feature model diagram) are translated as Boolean variables, enumerative variables or enumerative constants, according to Tab. 8.2. Alternative features are translated as an enumerative variable, where the sub features, which are mutually exclusive by definition, are represented as values in a corresponding enumerative type. This represents a main difference between the proposed approach and the classical one: just a single enumerative variable is introduced to model alternative sub-features, instead of a set of Boolean variables. However, any other feature type is still translated to a Boolean variable, as in the classical approach.

During this step, the function *isChosen* is set for each encountered feature. *isChosen* defines the predicates that must hold if a feature is selected.

It associates every feature to a propositional formula (i.e., a Boolean expression):

$$isChosen : Feature \rightarrow Expression$$

Consequently, $isChosen(x)$ means that feature x is present in the considered product configuration.

8.2.2 Adding implicit constraints

During the second step of the translation, the feature model is visited again starting from the root node and constraints among the features in the original model are translated into constraints between the variables of the corresponding combinatorial task. Constraints are built depending on the node and its parents semantics mapping, as shown in Table 8.3. We refer to these constraints as *implicit*, since they are implicitly implied by the type of relationships child-parent of the nodes.

8.2.3 Cross-tree constraints

Sometimes cross-tree constraints are used to limit valid product configurations and to model relationships among features. These constraints can be translated easily in CITLAB, which accepts as constraints general form Boolean expressions. Specifically, they are translated as follows:

FM constraint	CITLAB constraint
$A \text{ requires } B$	$isChosen(A) \Rightarrow isChosen(B)$
$A \text{ excludes } B$	$isChosen(A) \Leftrightarrow \neg isChosen(B)$

Example 12. Fig. 8.3 reports a small example which, however, contains all the feature kinds. Table 8.4 reports the results obtained by visiting each node during the visit of the diagram. The columns show the CITLAB Parameter, the value of the *isChosen* function, and the implicit constraints. Listing 8.1 reports the CITLAB code before simplification.

Variability factor In our case, the variability factor can be computed as the ratio between the number of valid products and the size of Cartesian product of the parameter domains.

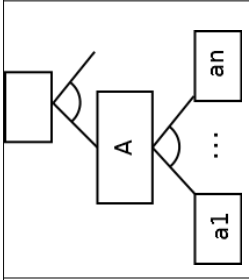
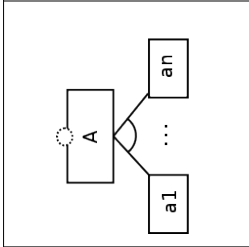
			<p>Any other node A_i whose parent is alternative P (A_i is a child in an alternative feature)</p>	<p>Any other node x whose parent is not Alternative</p>
Feature	Both A and its parent are alternative nodes	A is alternative but its parent is not alternative	skip (the father is alternative, the node is already translated into an element of an enumerative)	Boolean x
Parameter	Enumerative A {a1 ... an NONE}	$isChosen(A) \equiv A' = NONE$	$isChosen(a_i) \equiv P = A_i$	$isChosen(x) \equiv x = true$
$isChosen$	skip $isChosen(A)$	$A' = NONE$	$P = A_i$	$x = true$

Table 8.2: Representing features and setting $isChosen$ function

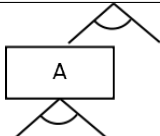
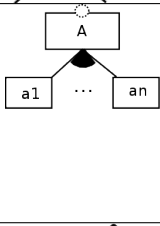
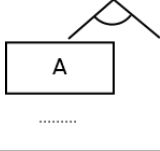
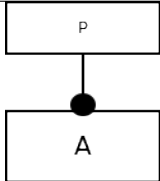
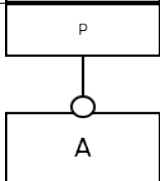
	Feature	Constraint
	A and its parent are alternative	$A \neq \text{NONE} \Leftrightarrow isChosen(A)$
	A is Or and its parent is of any kind	$isChosen(A) \Rightarrow$ $(isChosen(a1) \vee \dots \vee$ $isChosen(an)) \wedge$ $\forall i isChosen(a_i) \Rightarrow$ $isChosen(A)$
	A is different from Or and parent is alternative	<i>skip</i>
	A mandatory and parent not alternative nor or	$isChosen(A) \Leftrightarrow isChosen(P)$
	A optional and parent not alternative nor or	$isChosen(A) \rightarrow isChosen(P)$

Table 8.3: Constraints to be added

node	CITLAB Parameter	<i>isChosen</i>	CITLAB constraint
model	Enumerative model {a1 a2 a3 NONE}	model!=NONE	model!=NONE
a1	skip	model == a1	skip
a11	Boolean a11	a1 == true	a1 == true ==> model == a1
a12	Boolean a12	a12 == true	a12 == true ==> model == a1
a2	skip	model == a2	model == a2 ==> (a21 == true ∨ a22 == true)
a21	Boolean a21	a21 == true	a21 == true ==> model == a2
a22	Boolean a22	a22 == true	a22 == true ==> model == a2
a3	Enumerative a3 {a31 a32 NONE}	model == a3	a3 != NONE <=> model == a3
a31	skip	a3 == a31	skip
a32	skip	a3 == a32	skip

Table 8.4: Translation to CITLAB model for model of Fig. 8.3

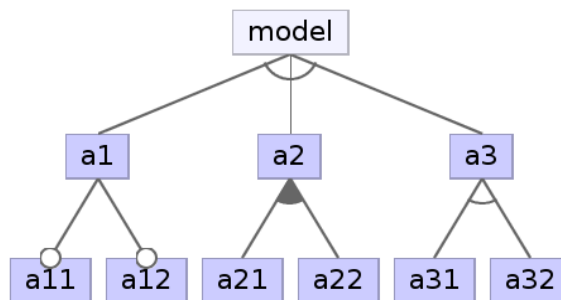


Figure 8.3: A small complete example

8.2.4 Extra testing requirements

Once the combinatorial model is derived from a feature model, the tester can apply the usual combinatorial interaction testing criteria in order to obtain set of products that cover the family of products in a desired way. Besides this standard use of CIT, our approach supports two additional features that are implemented in CITLAB.

Pre-built product configurations:

known configurations that must be included in the final suite of test configurations can be imported into the combinatorial model as a list of *seeded* tuples of feature combinations, which are already supported by some combinatorial generation algorithms integrated in CITLAB. For instance, in the FM of Fig. 8.1 if the tester wants to force the inclusion of a specific product, he/she can write the following seed in CITLAB:

Seeds:

```
# Calls = true, GPS = false, Screen = Color, Media = MP3#
```

CITLAB automatically checks that a seed sets the value to every parameter (hence to every feature) and that it satisfies the constraints (implicit and cross-tree).

Further testing goals

Testers may want to add extra requirements about the testing activity in form of further conditions over the product space that they want to cover. These test goals must be achieved either beside or by the combinatorial coverage and represent particular product configurations the tester wants to be sure

```

Model model
Parameters:
  Enumerative model { a1 a2 a3 NONE };
  Boolean a11;
  Boolean a12;
  Boolean a21;
  Boolean a22;
  Enumerative a3 { a31 a32 NONE };
end
Constraints:
  # model!=NONE #
  # a11==true => model==a1 #
  # a12==true => model==a1 #
  # model==a2 => ( a21==true || a22==true) #
  # a21==true => model==a2 #
  # a22==true => model==a2 #
  # a3!=NONE <=> model==a3 #
end

```

Listing 8.1: CITLAB code for model in Fig. 8.3

that they will be included in the final test suite. These extra constraints can be added in CITLAB by means of *test goals*. For instance, in the FM of Fig. 8.1 if the tester wants to include at least a product in which the GPS is present but Media is not MP3, he can write the following test goal:

```
TestGoals: # GPS == true and Media != MP3#
```

A test goal is *covered* if there exists at least a test that satisfies it.

For now, we assume that these extra testing requirements are manually inserted by the user after the translation is performed. We plan to study a way introduce them in the source feature models. For a precise semantics of test goals and seeds, see [34].

8.3 Simplification Process

The combinatorial model resulting from the imported feature model can be optimized in order to reduce the number of unnecessary variables and constraints. Although the proposed optimization is proposed in the context of feature model combinatorial testing, it can be applied to any combinatorial

```

Model model_smp
Parameters:
  Enumerative model { a1 a2 a3 };
  Boolean a11;
  Boolean a12;
  Boolean a21;
  Boolean a22;
  Enumerative a3 { a31 a32 NONE };
end
Constraints:
  # a11==true => model==a1 #
  # a12==true => model==a1 #
  # model==a2 => ( a21==true || a22==true ) #
  # a21==true => model==a2 #
  # a22==true => model==a2 #
  # a3!=NONE <=> model==a3 #
end

```

Listing 8.2: CITLAB code for model in Fig. 8.3 after simplification

model and it has been implemented in CITLAB as model transformation and added to its public APIs. We perform two types of syntactical simplifications:

- We simplify the constraints of the model in a semantic preserving way. The goal is to ease the test generation, which in the presence of constraints can be more difficult [25].
- We completely remove unnecessary parameters and constraints.

8.3.1 Constraint simplification

We apply the rules presented in the following table, which reports the constraint to be simplified (column A), the condition under which it can be simplified (column B), and the performed simplification (column C), where a and b can be either atomic proposition or complex propositions. It is straightforward to prove that the simplifications preserve the final semantics of the constraints.

8.3.2 Parameter simplification

Once the constraints have been simplified, we can simplify the model parameters, as reported in Tab. 8.5. In those cases, under some assumptions, the

(A) Constraint	(B) If already present	(C) Replaced by
$a \rightarrow b$	a	b
$a \rightarrow b$	b	- (remove)
$a \leftrightarrow b$	a	b
$a \leftrightarrow b$	b	a

Parameter	Constraint	Action
Boolean x	$x = true$ or $x = false$	remove x and remove the constraint (if x does not compare in other constraints).
enum $A\{a1... an\}$	$A \neq ai$	remove ai from type A and remove the constraint (if ai does not compare in other constraints).
enum $A\{a1... an\}$	$A = ai$	remove A and the constraint (if neither A nor ai compare in other constraints).

Table 8.5: Parameter simplification

parameters can be simplified and the constraints removed.

Example 13. For the example of two alternative features given in Fig . 8.2, the resulting CITLAB file is the following one:

```

Model model
  Enumerative A {a1 a2 a3 a4};
  Enumerative B {b1 b2 b3 b4};
end

```

Note that in this specific case, our simplification algorithm is able to remove all the constraints. Using unconstrained models, the tester has access to a wide set of tools, including most algebraic and bio-inspired tools for combinatorial testing (like [12]), which are very powerful in terms of generation time and test suite size, but they cannot easily deal with the constraints.

Note that since we operate only at syntactical level, we may miss some possible simplifications but the process is simple and fast. We plan to use in the future a constraint solving engine (like a SMT solver).

We propose a complete example of translation from a feature model to a CITLAB model.

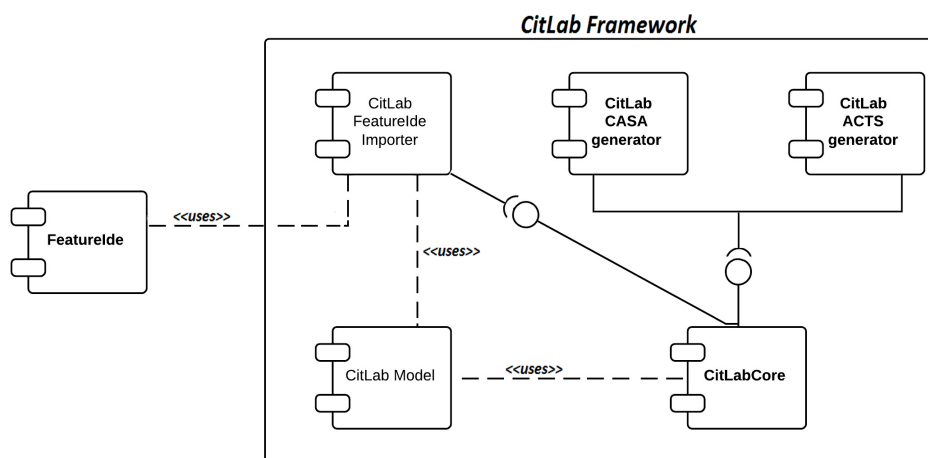


Figure 8.4: FeatureIDE importer plugin

8.4 Experiments

We have implemented in CITLAB an importer of FeatureIDE models, which can read feature models in the FeatureIDE native format and also in the SPLOT format (sxfm). We have also implemented an importer that translates feature models in SPLOT format into a combinatorial problem with only Boolean variables with the classical semantics given by [61]. These importers are CITLAB plugins and they are integrated in the framework together with other test generators as shown in Fig. 8.4.

As case studies we have taken 52 SPLOT² models which are often used as benchmarks³

8.4.1 Testing the correctness of the transformation

Formally proving the correctness of the proposed transformation would require a standard formal framework for feature models semantics, which is not available yet, although Schobbens and Benavides have made some progress in this direction [8, 73]. In this chapter we focus on *testing* the correctness of the proposed transformation by comparing it with the classical transformation using propositional logics. We want to assess the *semantic equivalence*: all the proposed transformation rules preserve the semantics of the feature

²SPLOT can be found at <http://www.splot-research.org/>.

³Note that because of a bug in FeatureIDE sxfm parser, we had to modify few SPLOT examples.

model. To systematically test the correctness of our translation we need to check the following characteristics:

- *Consistency*: our approach generates only valid products, i.e., products that satisfy the constraints.
- *Completeness*: our approach generates all the valid products.

The consistency is checked automatically within the integrated test case evaluator of CITLAB. CITLAB has an integrated logic evaluator to check for inconsistency and whether a parameter (feature) configuration is a valid test case or not. We rely on the fact that the internal evaluator works correctly.

The completeness is checked by simply verify for every case study that the number of distinct valid product of our approach is equal to the number of valid products found by the SPLOT analyzer. Note that the number of products including invalid ones may differ, but the number of valid configurations must be the same in order to preserve semantical equivalence.

The semantical equivalence ensures a biunivocal correspondence between one test case produced by CITLAB and one possible valid product.

8.4.2 Effect of the simplification over the parameters and the constraints

We want to check if the simplification process reduces the number of parameters and constraints. Fig. 8.5 reports the number of parameters and (Fig. 8.6) reports the number before (CTL) and after the simplification (CTL S). The simplification process has always reduced the number of both quantities in the models considered for experimentation. Our technique was able to remove all the constraints in 4 models.

8.4.3 Comparison with approaches using Boolean variables

We want to compare our approach with those using Boolean variables (like [8]) to check the effectiveness of our methodologies over the following quantities of the final models:

parameters: we should obtain *smaller* models

constraints: we should obtain *simpler* models

variability: we should obtain more *compact* models

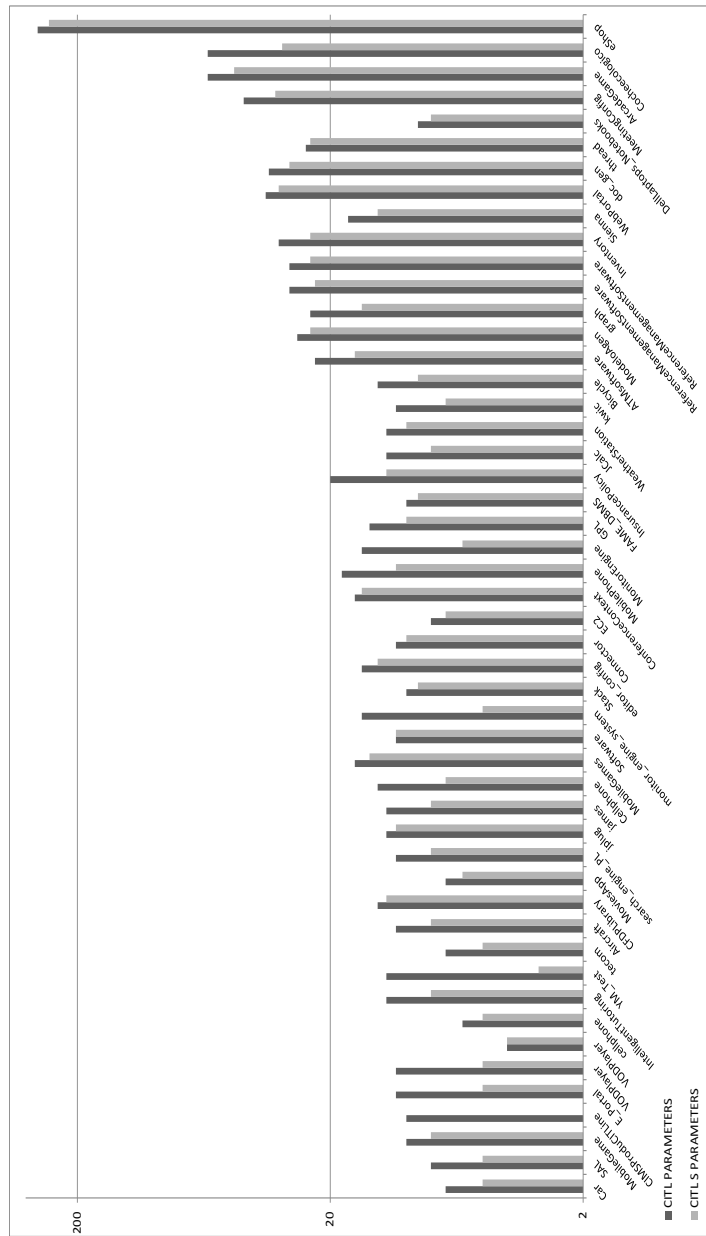


Figure 8.5: Reduction of the number of parameters

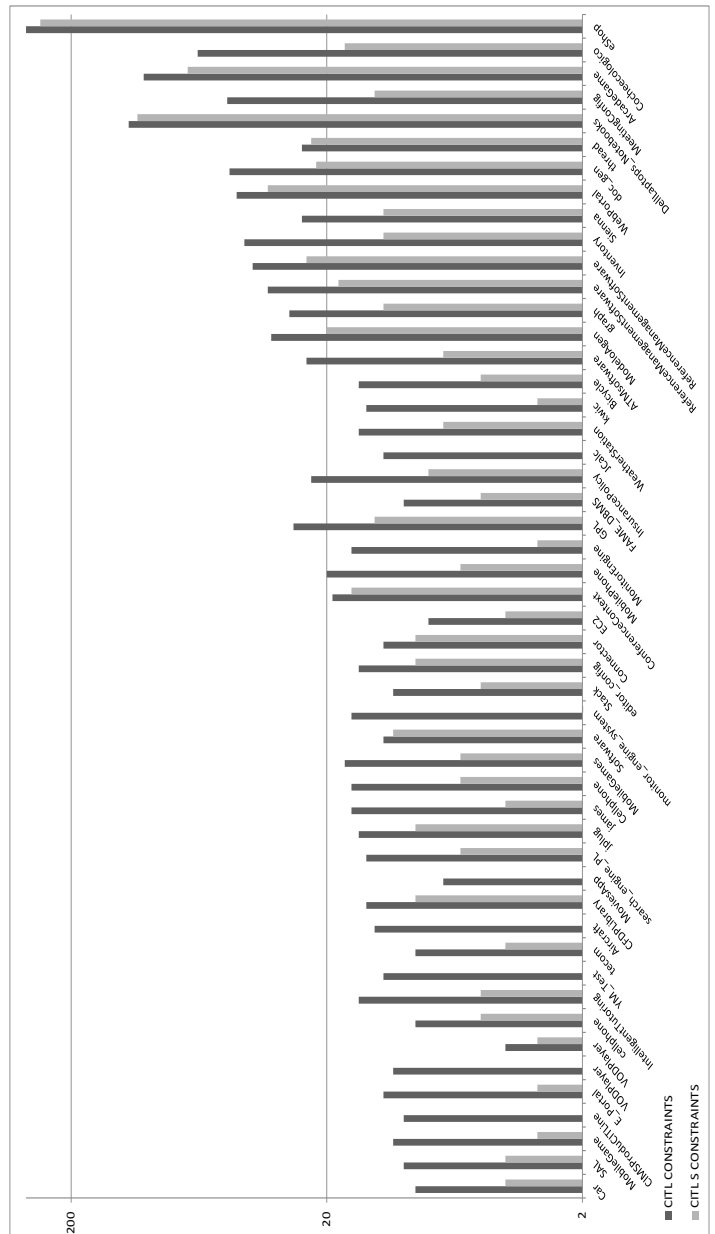


Figure 8.6: Reduction of the number of constraints

Fig. 8.7 compares the number of CITLAB parameters (after simplification) w.r.t. the number of features in the original model. It is apparent that our technique is able to reduce the number of parameters of problem. This should make the test generation faster to perform.

Fig. 8.8 compares the number of CITLAB constraints w.r.t. the number of constraints in the model obtained by using the classical translation implemented by SPLOT. Since SPLOT represents all the constraints in CNF, in order to perform a fair comparison, we have converted also the CITLAB constraints to CNF and the figure compares the total number of clauses in the CNF expressions. As the figure shows, our final models have always fewer constraints than those in SPLOT.

Fig. 8.9 shows the variability factor before and after the simplification process, while Fig. 8.10 compares the variability factor of CITLAB models with that of SPLOT models. The figures show that the simplification increases the variability factor and that our final models have a higher variability factor than the original SPLOT models. This means that in our approach valid products occur more often in the product space.

8.4.4 Test generation

A major advantage of our approach is that the tester can rely for test generation on different algorithms and tools developed for CIT. We want to evaluate the impact of our approach to the actual generation of combinatorial test suites.

Table 8.6 reports the results of pairwise test generation using ACTS⁴ which implements the IPOG algorithm [58] and which is integrated in CITLAB as generator plugin. We have run 100 test generations for 4 SPLOT models, by using the translation to only Boolean variables (BOOLEAN) and the method proposed in this chapter (CITL), where the simplification is applied (S) or not. Note that the coverage requirements for all these translation methods are the same. We generally obtain better results, i.e., both faster generation *and* smaller or equal test suites, with the proposed translation than that using only Boolean variables. However, there is one exception. For the Dell Laptop example, we have a smaller test suite but at the expenses of the test generation time. It seems that IPOG can further reduce the test suite size when using our models containing fewer parameters (8 parameters against 48 features) and simpler constraints. Per the biggest case study (Printer) and the BOOLEAN translation, ACTS did not complete the generation.

⁴<http://csrc.nist.gov/groups/SNS/acts/>

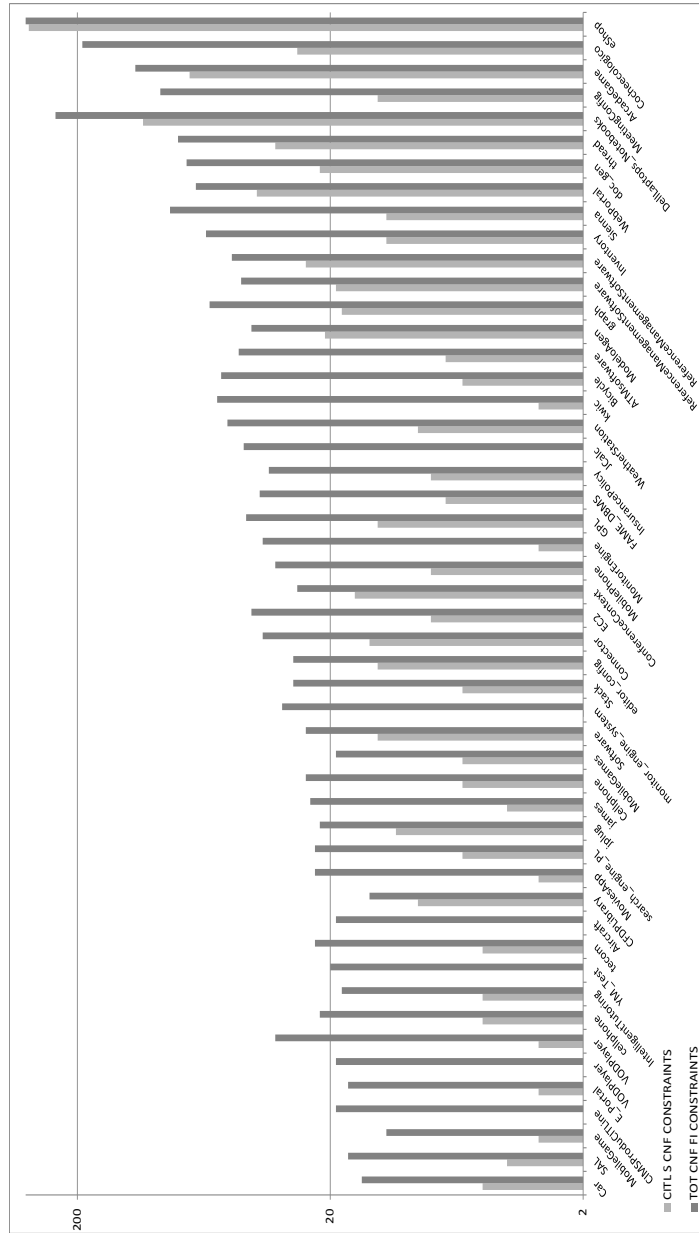


Figure 8.8: Comparison of CitLab constraints vs SPLIT constraints

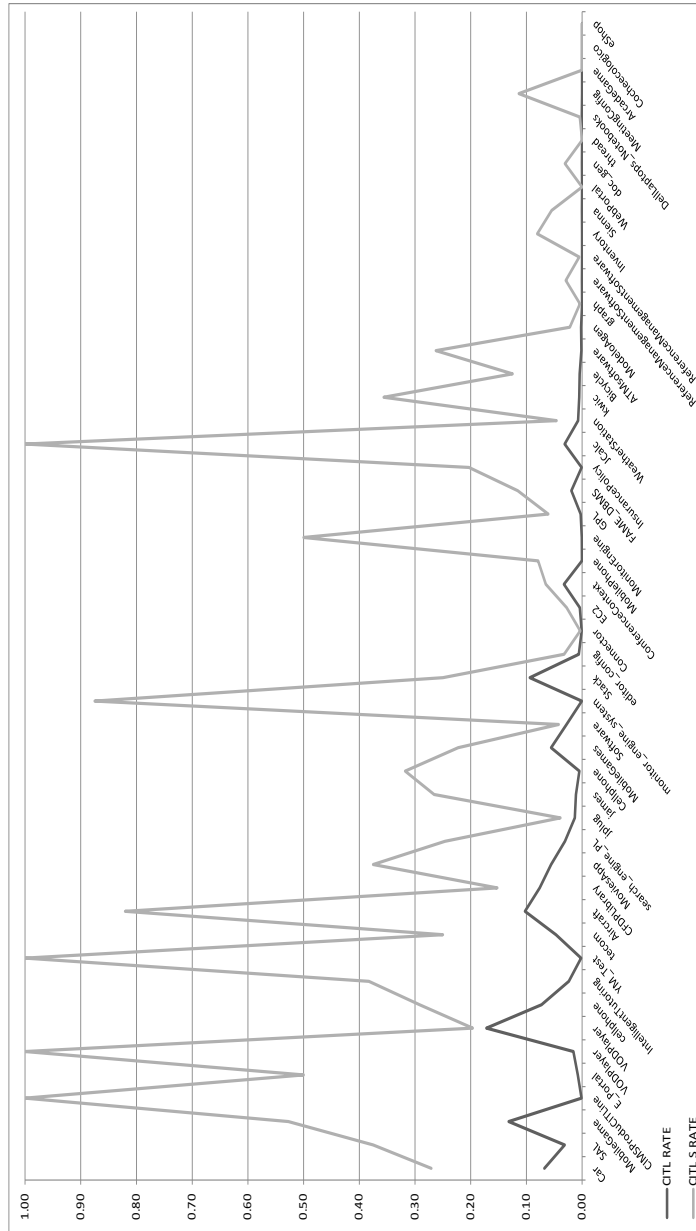


Figure 8.9: Variability variation due to simplification process

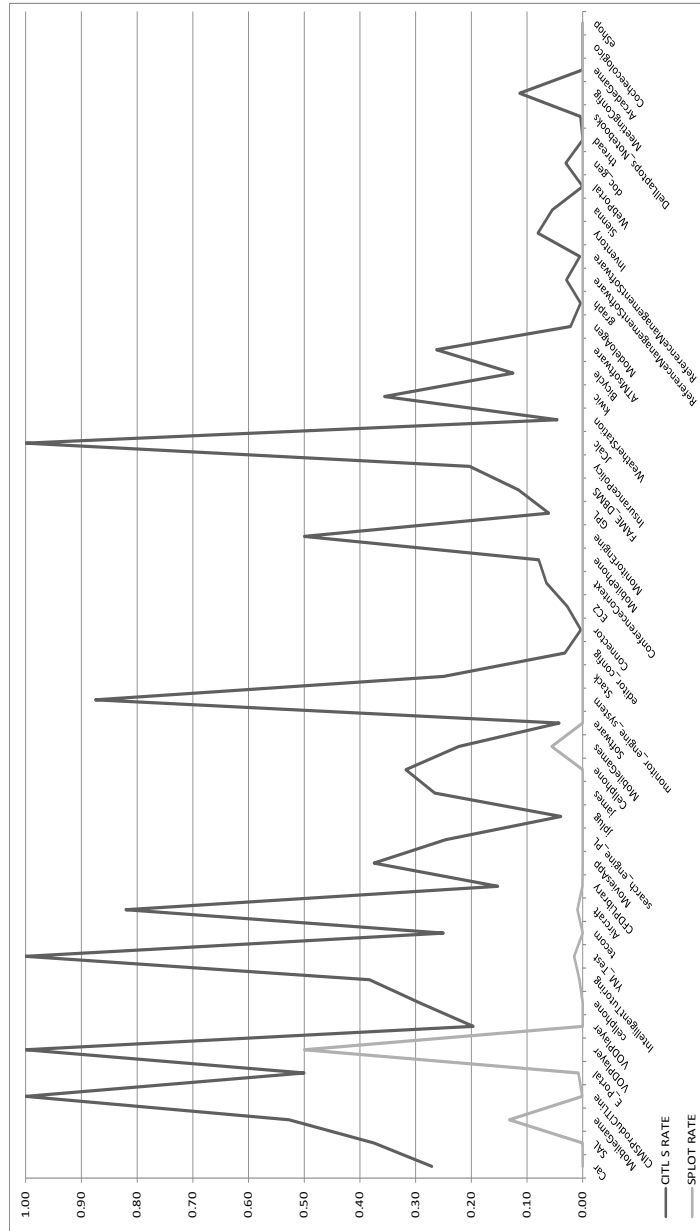


Figure 8.10: Variability comparison with SPLIT

	Aircraft		FameDBMS		Dell laptops		Printer	
	Time	Size	Time	Size	Time	Size	Time	Size
BOOLEAN	0.046	10	1.169	13	0.519	39	out of memory	
BOOLEAN S	0.014	10	0.976	13	0.785	39	out of memory	
CITL	0.017	10	0.020	13	4.296	36	20	180
CITL S	0.005	8	0.006	12	2.043	37	12	180

Table 8.6: Pairwise with ACTS (time in seconds)

	CITLAB by CASA		Oster [65]	PACOGEN [48]
	Size	Time	Size	Size
Sienna	20	1.76	24	20
Inventory	9	3.06	12	15
ArcadeGame	13	37.90	25	14
Web Portal	15	24.01	26	16
Doc generator	13	11.41	18	17

Table 8.7: Pairwise with CASA (time in seconds)

Table 8.7 reports the test generation of 5 case studies⁵ using our translation and the test generator tool CASA [36] which is integrated in CITLAB as generator plugin. In our experiments, CASA is able to produce smaller test suites than others test generation algorithms specifically designed for SPL testing.

Overall, we can say that our encoding has also advantages during test generation (in terms either of time or test suite size): some tools can take advantage of our simpler encoding and testers can benefit from having access to very powerful test generation frameworks not designed specifically for SPLs. However, further experiments are needed in order to give a final positive evaluation regarding test generation.

8.5 Related Work

There exist several attempts to give a precise semantics to feature models. Batory connects feature models, grammars, and propositional formulas [6] by giving a very simple yet clear meaning to feature models. The proposed connection allows arbitrary propositional constraints to be defined among features and enables off-the-shelf satisfiability solvers to debug feature mod-

⁵We chose these examples because data about using other tools is available in literature papers.

els. It would also allow the use of tools for generating combinatorial test suites, although this topic is not tackled in [6].

A more complete (including several variants of Feature Diagrams) and powerful semantics of feature models is presented by Shobbens [73], who emphasizes the necessity of precision and unambiguity for efficient and safe tool automation. They provide a formal semantics by a generic construction called Free Feature Diagrams (FFDs). Test generation from FFDs can be complex though.

Benavides et al. present several mappings from feature models to other formal notations (propositional logic and CSP) [8]. They focus more on automated analysis instead of testing, but our translation has been greatly influenced by their survey.

Regarding testing feature models and SPLs, a good survey can be found in [33]. A first attempt to apply CIT in the form of covering arrays to SPLs, using a simple variant of feature models called Orthogonality Variability Model (OVM), can be found in [24]. They define several testing criteria which are adaptations of combinatorial criteria to OVMs and identify some open issues like scalability, the use of constraints, and benchmarking. We believe that reducing the problem of SPL testing to a CIT problem can help to deal with all the issues mentioned in that paper.

In [67], the authors propose a scalable toolset using Alloy to automatically generate test cases satisfying T-wise from SPL models. The proposed toolset is based on the use of a SAT solver. However, an extension of the approach by using also Constraint Programming is presented in [68]. In that paper the authors present and evaluate two techniques, one focusing on generality and using high level strategies in order to improve the test generation. The other emphasizes providing efficient generation.

The tool PACOGEN is presented in [48]. PACOGEN relies on constraint programming to generate configurations that satisfy all constraints imposed by the feature model and to minimize the set of the tests configurations. Extensive experiments, based on the state-of-the-art SPLOT feature models repository, shows that PACOGEN scales well and produces reasonable small test suites also for large SPLs. A specialized algorithm (called ICPL) for generating covering arrays from feature models is presented in [51].

All these algorithms are specific to SPL testing but we believe that they could be used also for combinatorial testing, although they may need some modification in order to accept generic combinatorial problems. On the other hand, SPL testing could benefit from CIT algorithms, tools, and concepts (like seeds and test goals). Our experiments show that reusing CIT test generation tools for SPL testing can have great advantages.

8.6 Conclusion and Future Work

We have presented a mapping from feature models to combinatorial interaction problems which can leverage a combinatorial testing framework like CITLAB in order to generate test suites for software product lines or for system products. Our translation provides several advantages over classical mappings, like the reduction of the parameters and constraints, the possible use of concepts like seeds and test goals, and the exploitation of external tools developed during these years for CIT. The approach has been implemented in CITLAB in a friendly easily usable way as shown in Fig. 8.11

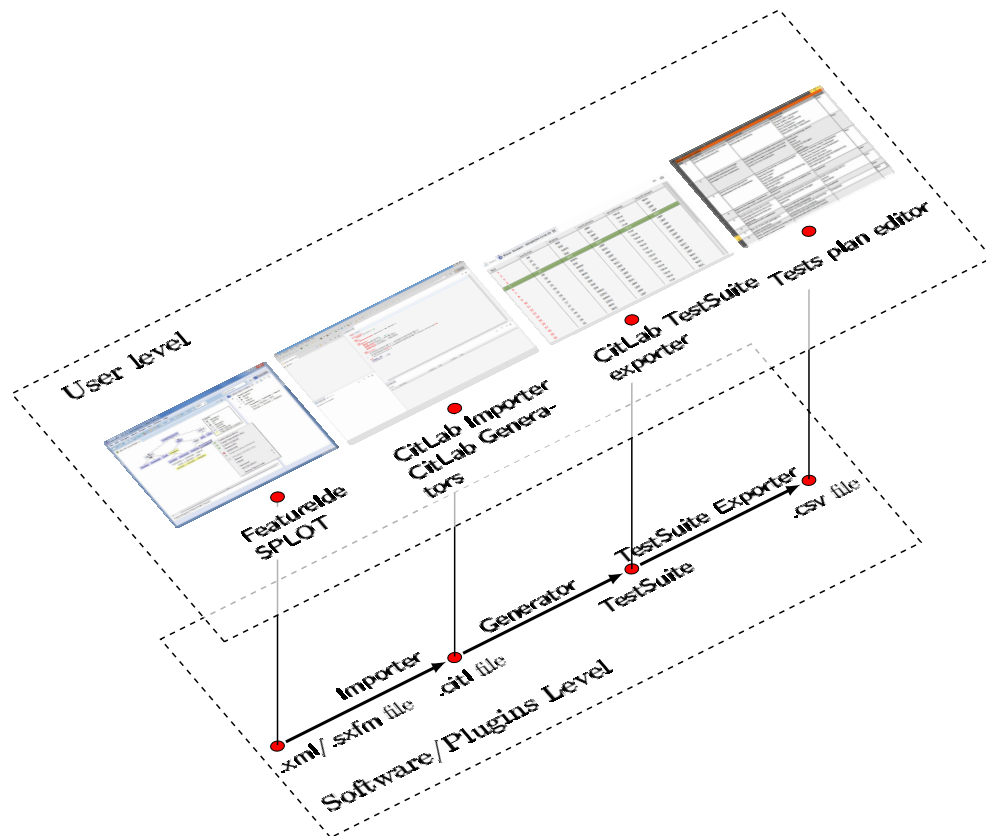


Figure 8.11: Implementation in CITLAB

For now, we support only simple feature models, but we plan to extend our translation to support feature models with extended notations, like those with cardinality and grouping.

9

Using Decision Trees for Algorithm Selection

It has been widely observed that there is no a single *best* CIT generation algorithm; instead, different algorithms perform best in terms of test suite size and time, also depending on different combinatorial models. Rather than following the traditional approach of leaving the choice of the best generator for a given class of models and for given testing requirements to the user, we want to automate the algorithm selection process among a given set of techniques (called *portfolio*). The proposed approach takes as input a distribution of combinatorial models and their test suites generated using several tools, then, using data-mining techniques, it permits to predict the algorithm that performs better given the cost estimated to execute a single test and the model characteristics. As predictors, we decide to use *decision trees* because they have been one of the most widely used decision support tool for many years. Their attraction lies in the simplicity of the resulting model, where a decision tree (at least one that is not too large) is quite easy to view, understand, and, importantly, explain even if it may not always deliver the best performances. We demonstrate the effectiveness of our approach to automated algorithm selection in extensive experimental results on data sets including models commonly presented in literature.

9.1 Algorithm Selection problem

Algorithm selection problem is widely studied in literature. John R. Rice formalized the concept of algorithm selection in [71] seeking to answer the question: "Which algorithm is likely to perform best for my problem?". In the early 1990s, the scientific community recognizes Algorithm selection problem as a learning task so, the machine learning community has developed the field of meta-learning, focused on learning about learning algorithm performance on classification problems. M. G. Lagoudakis and M. L. Littman in [56] consider the problem of algorithm selection: dynamically choose an algorithm to attack an instance of a problem with the goal of minimizing the overall execution time. This conception is similar to the one that aimed our approach even if they formulate the problem as a kind of Markov decision process (MDP), and use ideas from reinforcement learning to solve it while we used decision tree technique. We focused our work on the use of machine learning techniques applied to the combinatorial testing algorithm selection in order to minimize the sum of generation time and execution time for the generated test- suites. The SATzilla team in [83] made an interesting observation about the absence of a *dominant* SAT solver and about the fact that different solvers perform best on different instances. They suggest, rather than following the traditional approach, to choose the best solver for a given class of instances. We notice a similar situation for the different generation algorithms for combinatorial interaction testing and we advocated the need to aid the practitioners in algorithm selection. Their approach takes as input a distribution of problem instances and a set of component solvers, and constructs a portfolio optimizing a given objective function (such as mean runtime, percent of instances solved, or score in a competition).

We automated the selection of an algorithm at a time while they execute different algorithms in parallel. We plan to implement parallel execution tool and to set different execution timeouts for the different algorithms according to the different probabilities given by our classifier. Very few works that apply machine learning techniques to combinatorial testing. Jia et al. presents in [50] an algorithm for combinatorial interaction testing based on a Hyperheuristic search. They have implemented a reinforcement learning agent that is iteratively used for the tuning of the configuration parameters of the simulation annealing operators during the test-suite generation.

9.2 The importance of the "*right*" algorithm

During the last years, the combinatorial interaction testing (CIT) community has proposed many approaches for solving combinatorial testing problems. New techniques, tools, and algorithms are continuously proposed, benchmarked, and proved to improve over the state of the art in many cases. It remains unclear which approach must be considered as the *best*: it seems even impossible to state that a certain tool/technique clearly outperforms the others. In fact, even if we consider only the two main families of generation algorithms, namely greedy techniques and meta-heuristic approaches, we cannot say that one is surely better than the other. It is well known that greedy techniques are faster because they perform every decision only once while meta-heuristic ones may revisit their choices. It is reasonable to postulate that greedy algorithms run faster but meta-heuristic searches produce smaller samples size. Although recent works focus their attention to the improvement of both these techniques in order to fill their gaps in terms of performance, the contrast, between the size reduction of samples and the generation time reduction, is sharp. The reason lies in the nature of the problem: CIT is clearly a multi-objective problem where optimal decisions need to be taken in the presence of two conflicting objectives: namely generation time versus test suite size. Depending on how much the tester is willing to wait and on the cost of executing a single test, one tool or another may be the best choice. Moreover, also the features of the model under test may influence the choice of the tool. A tool may perform very well for small models but have problems of scalability. In this chapter, we use a mathematical model that formalizes the goal of minimizing the testing costs.

9.2.1 Aiding the selection

It would be good that the tester could choose the right algorithm among many. This was the original intent of CITLAB framework [19, 34]. As previously described, it allows CIT researchers to share their models and to run the major CIT tools (if supported) into a common environment for a scrupulous performance comparison. CITLAB supports already several test generation tools and new one may be added in the future. However, the choice of the right algorithm to use is still left to the testers. In this chapter, we try to introduce an automatic component that can suggest the user the right choice depending on several inputs (cost of executing a single test, characteristics of the model, and so on).

9.3 How to solve algorithm selection problem

Our approach tries to solve a classical selection problem of one algorithm in an *algorithm portfolio*. This idea was originally presented by Huberman et al. [49] to describe the strategy of running several algorithms in parallel, potentially with different algorithms being assigned different amounts of CPU time. Several authors have since used the term in a broader way that encompasses any strategy that leverages multiple black-box algorithms to solve a single problem instance.

We use the term *portfolio* to describe a set of algorithms from which to choose the one or ones best for the model under test. Namely, the algorithms for CIT test generations supported in CITLAB are ACTS [1, 57], CASA [21, 38], and MEDICI [35].

We propose to use as tool for supporting the user in the decision of the technique to be used for test generation, decision trees, which can guide in a simple way the users in the best choice. We present the process of building such decision trees by using a data mining process that starts from the analysis of the performance of CASA, ACTS and MEDICI over a selection of 114 models. The ideal solution to the algorithm selection problem would be to consult an oracle that knows exactly the amount of time that each algorithm would take to solve a given problem instance and the size of the test suite that would be generated, and then to select the algorithm with the best performance. In the context of the CIT, knowing the exact time and the exact test suite size is almost impossible without actually executing the tool itself. Moreover, in general, there is no need to know the size of and the time since the tester suffices to know which tool to execute in order to achieve a certain goal (for example a very small test suite size). For these reasons, our decision trees will learn which tool to select for a particular model under test.

9.4 Background

9.4.1 Defining the "best" combinatorial generation algorithm

There are many algorithms and tools for combinatorial test generation. In a recent book [84], Zhang et al. have counted 12 tools/framework actively maintained for CIT testing, while the pairwise web site¹ lists around 39

¹<http://www.pairwise.org>

tools. Another recent survey lists around 50 papers dealing with test generation [63]. The research community and some commercial activities continuously propose new algorithms (or improved version of existing techniques) and software for CIT test generation. It is apparent that the choice of the right tool for test generator can be difficult. Even if one wants to focus on one single tool, it may have several options that make even its usage not an easy task. To simplify the problem we can limit our attention to free tools, consider models containing constraints, ignore other aspects like usability, interoperability and so on, and focus only on the *test generation time* and *test suite size*. Even in this case the identification of the best test generator is not easy because the test generation for CIT is a typical multi objective problem in which test suite size and test generation time are two conflicting objectives: a tool can be very fast but produce enormous test suites, while another may guarantee to find very small complete test suite but require hours of computation.

In order to allow a fair comparison among tools, to guide the choice of the best suitable one, and to devise a technique to extract the decision tree that can help the user in the choice, we first borrow the model proposed in [38] for roughly estimating the cost of testing (*cost*) as the total time for test generation (*time_{gen}*) plus test execution time, which depends on the size of the test suite (*size*) and on the time necessary to execute every single test (*time_{test}*):

$$cost = time_{total} = time_{gen} + size \times time_{test} \quad (9.1)$$

In this model, we assume that the cost of testing (both test generation and execution) is equal to the time required for the activity. In case the cost is linearly bound to the time, for instance, because testing requires the use of a dedicated server that has an hourly cost, or because tests are manually executed by a person with a temporal cost, our model still holds (provided that a constant is introduced). We leave as future work the study of other models of costs. The cost of executing one single test is also called unitary execution cost (**UEC**).

9.4.2 CITLAB as a framework for benchmark comparison

In order to enable a real choice between several algorithms and to avoid a vendor lock-in, we use CITLAB. CITLAB allows importing/exporting models of combinatorial problems from/to different application domains, as reported in Chap. 8, by means of a common interchange syntax notation and a cor-

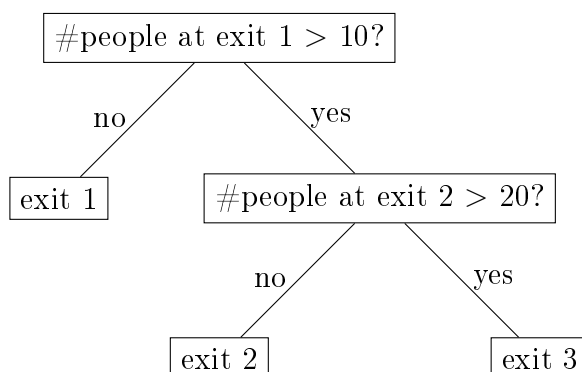


Figure 9.1: A decision tree representing the exit selection in a crowded parking area.

responding inter-operable semantic meta-model. As previously described, CITLAB already supports three main generation techniques: ACTS [1, 57], CASA [21, 38], and MEDICI [35]. All three support constraints and they are freely available. ACTS and CASA have a large user base and they are very often used in comparison studies. Using CITLAB allows us to perform all the experiments in a very controlled environment on the same computer and using exactly the same models. We assume in this chapter that the CIT portfolio is constituted by the three tools mentioned above. We plan to add more generators in the future.

9.4.3 Decision Tree

We propose the use of decision trees for the suggestion of right tool for test generator. Decision trees (also referred to as classification and regression trees) are the traditional building blocks of data mining and the classic machine learning algorithm. Since their development in the 1980s, decision trees have been the most widely deployed machine-learning based data mining model builder. Their attraction lies in the simplicity of the resulting model, where a decision tree (at least one that is not too large) is quite easy to view, understand, and, importantly, explain. Classification tree structure is used in many different fields, such as medicine, logic, problem solving, and management science. It is also a traditional computer science structure for organizing data.

Fig. 9.1 shows a simple decision tree that can be used to decide the exit in a parking area depending on the number of people in it.

9.4.4 Tools for Data Mining

For data mining, we use the free and open source software Rattle [81], built on top of the R statistical software package [70]. Rattle has been developed using the Gnome toolkit with the Glade graphical user interface (GUI) builder. Rattle provides considerable data mining functionality by exposing the power of the R Statistical Software through a graphical user interface. There is a Log Code tab, which replicates the R code for any activity undertaken in the GUI, which can be copied and pasted. Rattle can be used for statistical analysis, or model generation and it allows the partition of the dataset into *training, validation, and testing* subsets.

We consider R one of the most comprehensive statistical analysis package available. It incorporates all of the standard statistical tests, models, and analyses, as well as providing a comprehensive language for managing and manipulating data. New technologies and ideas often appear first in R. Rattle (the R Analytical Tool To Learn Easily) provides a simple and logical interface for data mining. The application runs under GNU/Linux and MS/Windows. We choose Rattle because it provides an intuitive interface that takes the practitioner through the basic steps of data mining.

9.5 Process of Building the Decision Tree

In this section, we explain the process of building a decision maker for the selection of the best algorithm for combinatorial test generation. With *decision maker*, we mean a statistical predictor that is able to forecast which generator produces the minimum total test cost for a specific model. The predictor will need some data as inputs, like the cost of execution of a single test and some of the features of the model, and will produce as suggestion an algorithm.

The process of finding such decision maker is a typical data-mining problem. The CRISP-DM (Cross Industry Standard Process for Data Mining) [76] identifies five steps within a typical datamining project:

1. Problem Understanding
2. Data Understanding and Preparation
3. Modeling
4. Evaluation
5. Deployment

9.5.1 Problem Understanding

During this phase, we try to understand the project objectives and requirements, and then identify the data that define the problem.

The main objective of this project is to find a decision tree that can help the user to choose the right CIT generator that minimizes the cost defined in Equation 9.1. We want to devise a predictor that given an estimated cost for each single test ($time_{test}$) and a certain model, is able to suggest a test generator that minimizes the final cost. Note that the time required for each test generation ($time_{gen}$) and the *size* of the produced test suite depend on both the chosen test generator and on the attributes of the combinatorial model. The cost for a single test $time_{test}$ plays a very important role in the selection of a test generator, because for small values of $time_{test}$ the cost is mainly due to the $time_{gen}$, while for big values of $time_{test}$, the *size* is more important. Since $time_{gen}$ and *size* depend on the generator and they are generally negatively correlated, the $time_{test}$ greatly influences the choice of the best test generator. So the first data that influence our problem is:

- **UEC**: the unitary execution cost. This is given by the tester as an average of the expected time required to execute a single test.

Regarding the model attributes that influence the test generation; we can identify the following possible candidates:

- **N.var**: number of variables.
- **N.constraints**: number of constraints. In this case, we can normalize the number of constraints by converting them to CNF and then by counting the number of clauses.
- **DomainSize**: number of possible configurations ignoring any kind of constraint of the model.
- **N.valid**: number of valid configurations (considering also the constraint).

All these variables are known to have some impact on the total cost of the test.

The work, presented in this chapter, is focused on finding new pattern for predicting the behavior of combinatorial generators, analyzing the main features of a given CIT model. We have developed a new approach to test suites generation based on a portfolio of CIT generators. We can distinguish two types of goals: verification and discovery. With verification, the system is limited to verifying the user hypothesis. With discovery, the system

autonomously finds new patterns. We further subdivide the discovery goal into prediction, where the system finds patterns for predicting the future behavior of some entities, and description, where the system finds patterns for presentation to a user in a human-understandable form.

9.5.2 Data Understanding and Preparation

The data understanding phase starts with an initial data collection and proceeds with activities in order to get familiar with the data, to identify data quality problems, to discover first insights into the data, or to detect interesting subsets to form hypotheses for hidden information.

During this phase, we collect all the data (models and test generation data), we compute the model attributes, and we select the relevant features that can be used as input variables for the prediction model.

Collecting benchmarks As training instances for CIT problems we have gathered a wide set of 114 models with constraints taken from the literature (Casa [21, 25, 38], FoCuS [74], ACTS [1], and IPO-S [17]) and from SPLOT SPLs repository, and used (in subsets) also by many other papers. The benchmarks can be found on the CITLAB web site and they can be used for further comparisons. Fig. 9.2 shows the distribution of the characteristics we are interested and that could influence the test generation process for the models under observation. The data in Fig. 9.2 prove that our benchmarks cover a rather wide range for every model attribute.

Test generation data We performed 50 runs over these 114 models for 4 different generators: all the 3 tools with two configurations of MEDICI. MEDICI, as many other tools, can be fine tuned by using options: we use a fast variant (MEDICI_1_1_1) and a more slow one (MEDICI_10_30_5) which should produce fewer tests [35]. We use the same tool with two different configurations in order to prove that our approach could be used to decide the parameters for a single given tool.

Using an R script we have calculated the total cost for each couple, generator-model, varying the cost of a single test execution UEC in the set {0.01, 0.1, 1, 10, 50, 100, 500, 1000, 5000} seconds.

Our machine learning work begins from this base of knowledge, the aim of this work is to predict the generator that performs better for a model according to a given single test cost and to its peculiar features.

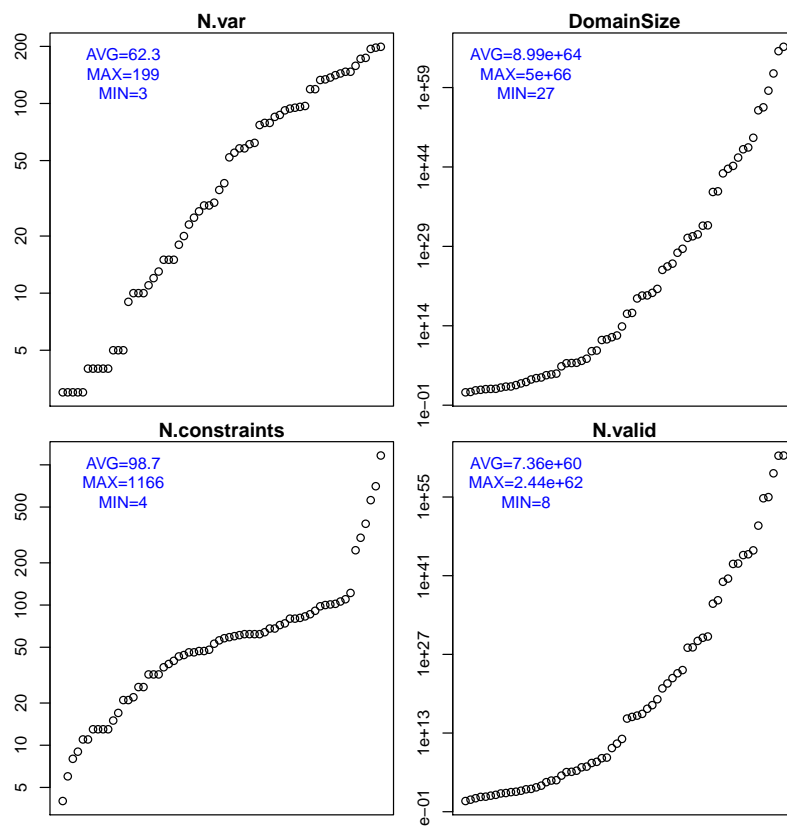


Figure 9.2: Training set attributes and characteristics

Model features In order to gather the data about the models, we use the CITLAB APIs that allow querying the model and obtain structural data like number of variables and so on. Obtaining the number of valid configurations, however, is more complex: one could easily enumerate all the configurations and count those valid. This is very time consuming. Instead, we use the capability of Multi-Valued Decision Diagrams embedded in MEDICI of counting the number of valid paths in a very efficient way.

Correlation features selection (CFS) We select the features that will constitute the inputs of the decision trees. Good candidates should be uncorrelated with each other but highly correlated with the prediction outcome [42]. In our case, the generator portfolio represents the prediction class. We start our feature selection process by the observation of the correlation test of Pearson. Its results are shown in Fig. 9.3. Pearson’s correlation coefficient between two variables is defined as the covariance of the two variables divided by the product of their standard deviations as described by the eq. 9.2:

$$\rho = \frac{\text{cov}(X, Y)}{\sigma_X \sigma_Y} \quad (9.2)$$

Fig. 9.3 shows that **UEC** is uncorrelated to other variables (as expected) and it must be considered. Regarding the model features, we note that **N.var** is correlated to all the other variables under test so it is the first candidate to be excluded from our subset of features. Analyzing the Pearson chart, we can also notice that **N.valid** and **DomainSize** are strongly correlated with each other so one of them is the next candidate for the exclusion. Performing an accurate analysis of correlation between the two candidates, we can estimate that they are interchangeable in terms of correlation with the predictive class. This condition led us to exclude **N.valid** because it requires a good amount of time to be computed. Our features subset is composed by: **DomainSize**, **N.Constraints**, and **UEC**.

9.5.3 Modeling

In this phase, we use Rattle to produce two predictive models based on decision trees. Exploring some data mining and machine learning techniques we have obtained two different automatic decision makers:

1. **CBT** cost-based decision tree: this classifier is able to select “the best” generator evaluating **only** the test execution cost (**UEC**), ignoring all the features of the model under test.

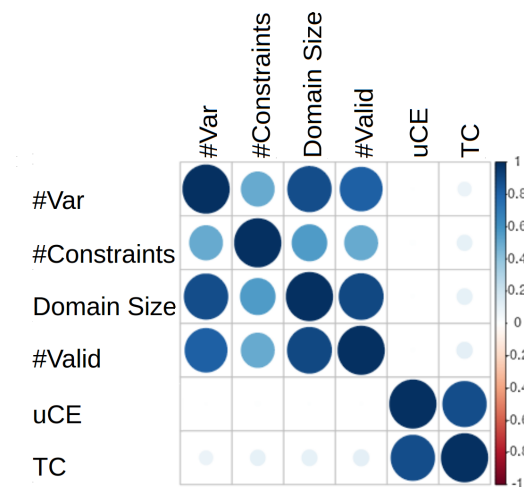


Figure 9.3: Pearson chart

2. **FBT** feature-based decision tree: this classifier receives as inputs also the domain size of the model, numbers of constraints and the cost for a single test execution and predicts the best generator.

Building the training/validate/test datasets Using the functionalities of Rattle we have divided the dataset into three different subsets: *training*, *validation*, and *test*. The training subset is used during the modeling process as base of knowledge for our predictive model (the decision tree), the other two sets are used during the evaluation process.

The R script produced by Rattle and used to build the three subsets follows.

```

# Randomly allocate 70% of the dataset to training ,
# 15% to validation , and the remaining 15%
# to testing
set.seed(crv$seed)
crs$nobs <- nrow(crs$dataset)
# Splitting the dataset in Training set (70%)
crs$sample <- crs$train
  <- sample(nrow(crs$dataset), 0.7*crs$nobs)
# Validation set (15%)
crs$validate <- sample(setdiff(seq_len(nrow(crs$dataset)), crs$
  train), 0.15*crs$nobs)
# Test Set (15%)
crs$test <- setdiff(setdiff(seq_len(nrow(crs$dataset)), crs$train
  ), crs$validate)

```

Variable selection After the sub-setting process, we have selected the input features of the decision according to the CFS previously performed, and we have set the total cost (**TC**) as the risk variable and the generator classes as target. The decision tree is in fact used as classifier for the selection of an optimal generator that minimizes the **TC** of testing. In the following paragraphs, we describe the modeling process used to produce **FBT**. **CBT** and **FBT** belong to the same modeling process but **CBT** takes as input variables only **UEC**.

The R script used to select the input variables and the target follows.

```
# The following variable selections have been noted.
# Selecting the three numeric input variable
crs$input <- c("N.constraints", "DomainSize", "UEC")
crs$numeric<-c("N.constraints", "DomainSize", "UEC")
crs$categoric <- NULL
#Imposing the generator classes as target
crs$target <- "generator"
crs$risk <- "TC"
crs$ident <- NULL
# Ignoring the other features
# of the base of knowledge
crs$ignore <- c("model", "N.var", "N.valid", "run", "size", "
  time")
crs$weights <- NULL
```

Modeling We have obtained a compact decision tree using Rattle. Its decision tree functionalities are based on the library **rpart**. Rattle allows the user to set 4 configuration parameters:

- *Min split = argument* specifies the minimum number of observations that must exist at a node in the tree before it is considered for splitting.
- *Max depth = argument* limits the depth of a tree.
- *Min bucket = argument* is the minimum number of observations in any terminal leaf node (conventionally $minsplit=3*minbucket$).
- *Cost complexity (cp) = argument* is used to control the size of the decision tree and to select an optimal tree size. The complexity parameter controls the process of pruning a decision tree.

The following extract of code shows that we required that the minimum number of observations in a node is 50 before attempting a split and that a split must decrease the overall lack of fit by a factor of 0.015 (cost complexity factor), we have also limited tree depth to 10. As parameters tuning process performed, we have substantially performed a "backward tuning", where we have obtained the final configuration by generating iteratively a new model by varying one parameter at a time and discarding the model and the change in the configuration parameter if the accuracy of the new model decreased with respect to the previous one. We have pruned back the tree to avoid overfitting the data. We wanted to select a tree size that minimizes the cross-validated error. Specifically we have iteratively examined the cross-validated error results varying the complexity parameter, and we have selected the *cp* associated with minimum cross-validation error. Rattle uses an information gain measure for deciding between alternative splits. This decision algorithm is based the concept of *Shannon Entropy*. The split that provides the greatest gain in information (and equivalently the greatest reduction in entropy) is the chosen split. The R script used to build the decision tree (FBT) follows.

```
# Building decision tree
# Using the dataSet previously prepared we build
# the decision tree using rpart
crs$rpart <- rpart(generator ~ .,
data=crs$dataset[crs$train, c(crs$input, crs$target)],
method="class",
parms=list(split="information"),
control=rpart.control(minsplit=50,
minbucket=16, maxdepth=10, cp=0.015000))
```

The two final decision trees CBT and FBT are reported in Fig. 9.5 and 9.4. Although FBT is more complex than CBT, both trees are rather understandable and easy to follow in order to get guidance on what tool to use even by hand.

9.5.4 Evaluation

During this phase, we evaluate the prediction models obtained in the previous phase. The results of the evaluation are presented in Section 9.6.

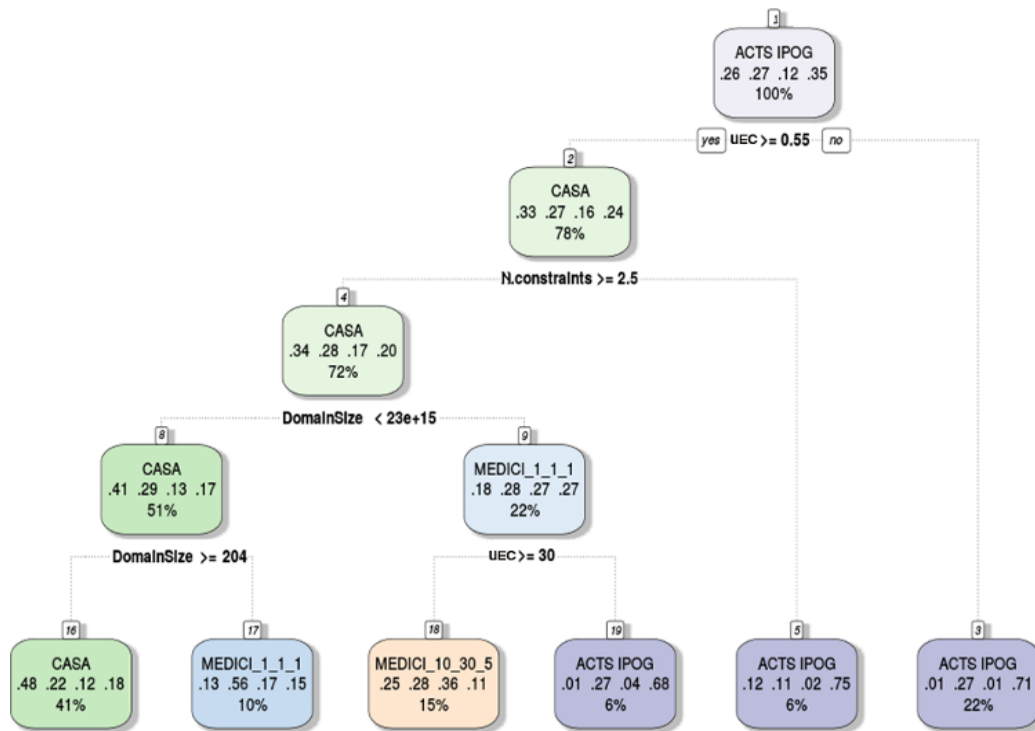


Figure 9.4: FBT: feature-based decision tree

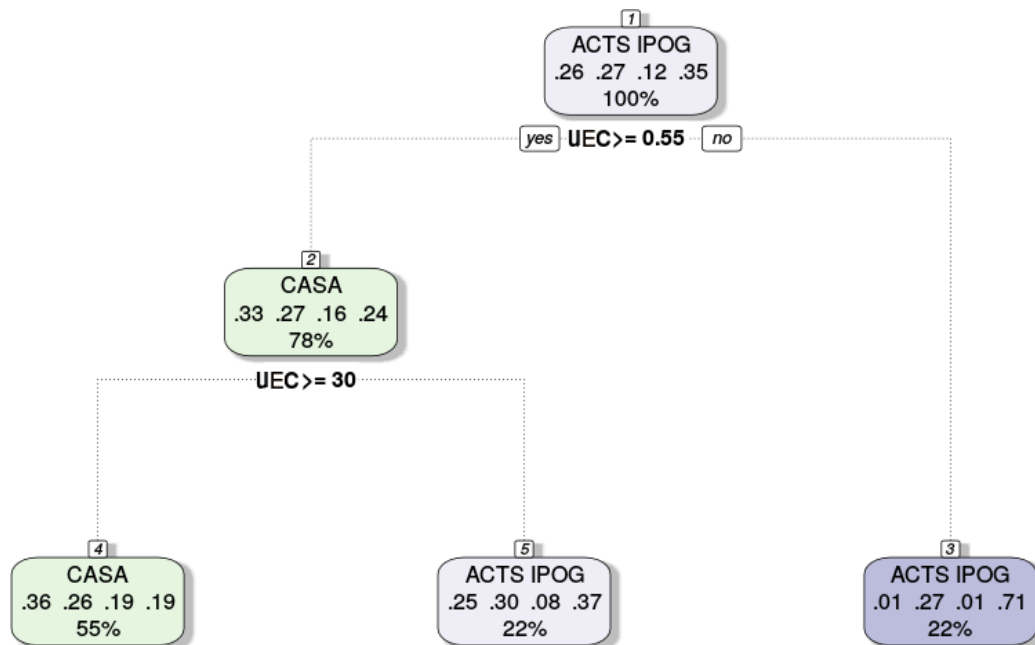


Figure 9.5: CBT: cost-based decision tree

9.5.5 Deployment

During this phase we describe how it is possible to re-use our predictive models, computed using Rattle, in a Java application and consequently integrating them in the CITLAB framework. Our deployment is based on the use of the Predictive Model Markup Language (PMML), which is an XML-based file format developed by the Data Mining Group to provide a way for applications to describe and exchange models produced by data mining and machine learning algorithms. It supports common models such as logistic regression, feed-forward neural networks and decision trees. Rattle supports the export of predictive models to PMML language. Using JPMML is possible to use PMML models in Java allowing their integration in CITLAB as a plug-in for Eclipse. Using JPMML, we have noticed these pros and cons. Advantages:

- No lock-in; you can run your models with any library that can read PMML models.
- Runs inside the Java process. No inter-process communication.
- Pure Java solution at run-time.

Disadvantages:

- Supported models depend on the PMML library (for example, JPMML does not support support vector machines SVM, even though it is in the specification of PMML 4.1).

9.6 Experiments

In the experiments, we want to compare the two decision trees and measure the advantages of using our classifiers against the use of a single generator.

First, we build the best possible predictor one could reasonably have. We identify the generator that gives the minimum average total cost for each model and for each single test cost over all the run we have performed, in order to simulate an *average optimum predictor*. Starting from these data, we build an *optimum* classifier, which would always select on average the best generator for every model. The main problem of the *optimum* predictor is that it can be computed only after the data have been generated ("a-posteriori" predictor).

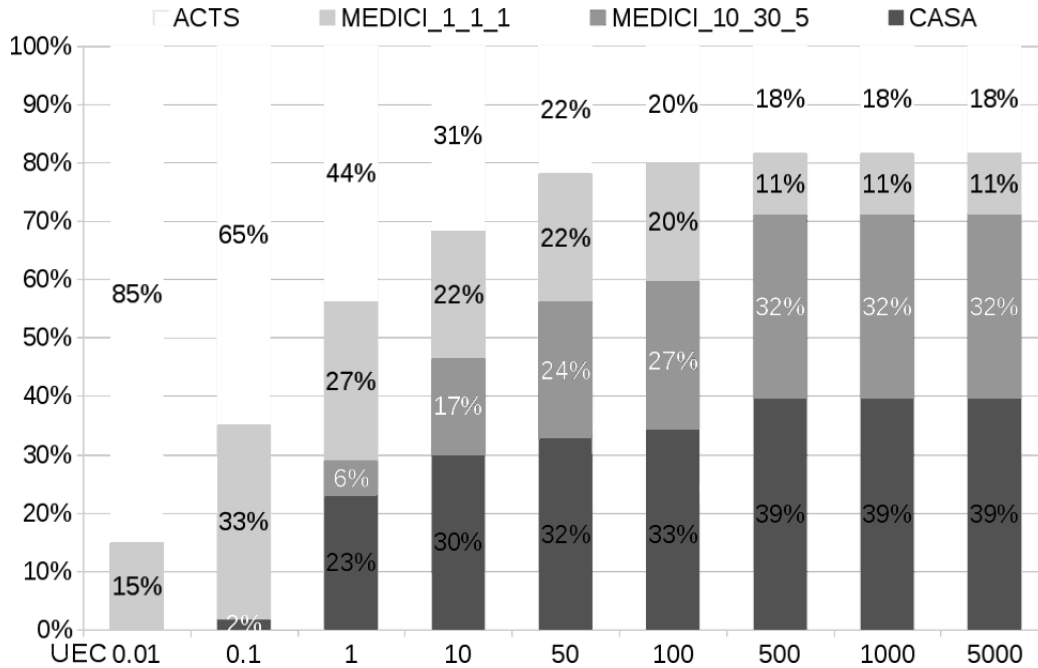


Figure 9.6: Test generator distribution of the optimum predictor

RQ1 Which generators would choose the optimum predictor?

Fig. 9.6. shows the distribution of the predicted generators performed by the optimum predictor for each UEC. ACTS is the privileged choice for small test costs due to its fast generation speed while CASA is the best solution for high test costs because it is very slow compared to ACTS but it produces smaller test-suites. MEDICI stays in between ACTS and CASA. MEDICI_1_1_1 produces test suites a little smaller than ACTS's ones but it is 10 times slower, MEDICI_10_30_5 produces test suites comparable to CASA but it can be faster [35]. Overall, ACTS is chosen in the 85% of cases for a single test cost of 0.01s and its percentage continuously decreases until 18% at the test cost of 500s while the percentages of the other generators increase. We can estimate that for a single test cost of about 100s to 500s the 4 generators under test reach a sort of stability point in the distribution between the chosen generators. After this point, the increase of cost does not produce any kind of further changes in the percentages of the selection distribution.

The figure confirms that there is no one *best* generator. By varying the UEC the distribution of generator choices may sensibly change. The data also show that, even the test cost were fixed, the choice of the best generator depends on the model features. This confirms the validity of our assumptions.

RQ2 When does FBT choose a wrong generator?

Fig. 9.7 shows the distribution of generators selected by FBT. It differs from the optimum reported in Fig. 9.6. Gray cells of Tab. 9.1 display the *confusion matrix* of the predictor. The difference between the optimum and the predicted one is reported as percentage of generator confusion. A confusion matrix displays the percentage of correct or incorrect predictions made by a classifier such as a Bayesian network or decision trees. It is automatically computed by Rattle using the test set. Diagonal elements of the matrix show the percentage of correct predictions, while off-diagonal elements show incorrect predictions. The sum of diagonal values represents the accuracy of the classifier under validation process. False negative rate (FNR) is computed, for each row, as the ratio between the sum of off-diagonal values (false negatives) and the sum of each value of the row values (false negatives + true positives). The high FNRs of the two configurations of MEDICI show a criticality of FBT in their right identification and prediction. FBT presents an accuracy of 58%.

RQ3 When does CBT choose a wrong generator?

Fig. 9.8 shows a distribution of selected generators that differs, as expected, from the optimum reported in Fig. 9.6. The difference between the optimum generator and the predicted one is reported in Tab. 9.2 as percentage of generator confusion. We notice that CBT performs a trivial choice between ACTS_IPOG and CASA ignoring the other 2 generators. For

²False negative rate is the proportion of events that are being tested for which yield negative test outcomes with the test, i.e., the conditional probability of a negative test result given that the event being looked for has taken place. False Negative Rate = (false negative)/(true positive + false negative).

³accuracy=(true positives + true negatives)/(positives + negatives)

Table 9.1: FBT confusion matrix for the test set

Optimum choice	Predicted				FNR ²
	CASA	M_1_1_1	M_10_30_5	ACTS	
CASA	22%	1%	4%	1%	25%
M_1_1_1	8%	6%	4%	7%	76%
M_10_30_5	5%	2%	5%	0%	58%
ACTS_IPOG	7%	1%	2%	24%	29%
Accuracy ³	58%				

MEDICI, the FNRs are 100%. CBT decision exchanges MEDICI_10_30_5 to CASA because MEDICI generator performance, in terms of size, is similar to CASA's one but it differs from it in terms of time consumption at the varying of models complexity. The reduction of FBT model to CBT decreases the accuracy rate of 12% (from 58% to 46%). These data suggest that a right estimation of the total cost produced by a generator and a right selection of the generator should consider other feature besides the UEC.

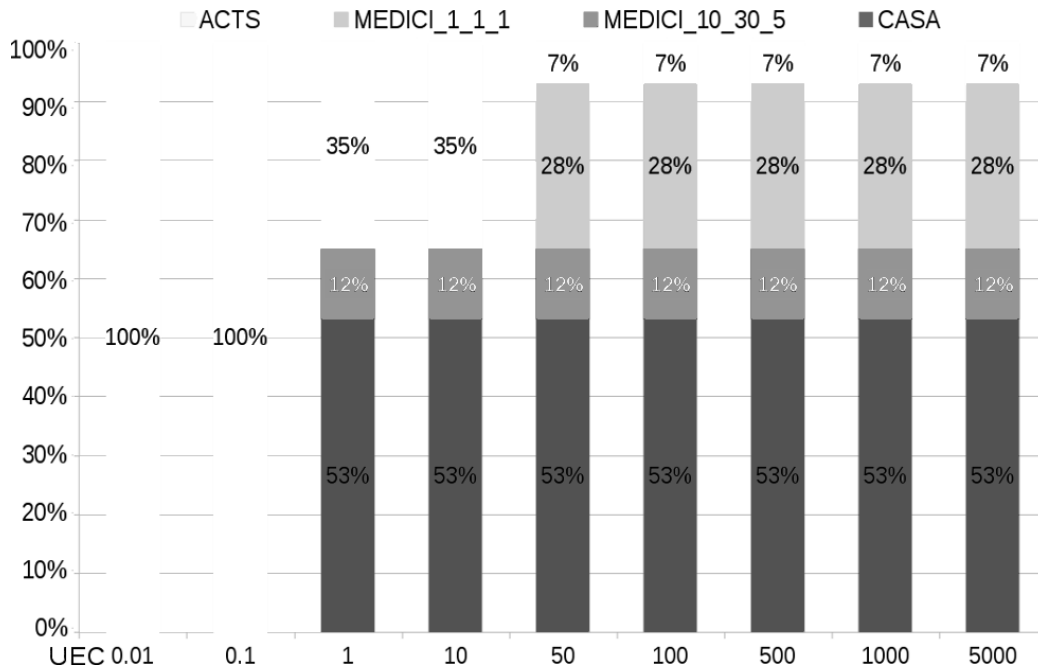


Figure 9.7: Generation tools distribution using FBT

RQ4 Can our predictors outperform the other fixed generators?

Table 9.2: CBT confusion matrix for the validation set

Optimum choice	Predicted				FNR
	CASA	M_1_1_1	M_10_30_5	ACTS	
CASA	22%	0%	0%	6%	21%
M_1_1_1	13%	0%	0%	12%	100%
M_10_30_5	11%	0%	0%	2%	100%
ACTS_IPOG	11%	0%	0%	24%	31%
Accuracy	46%				

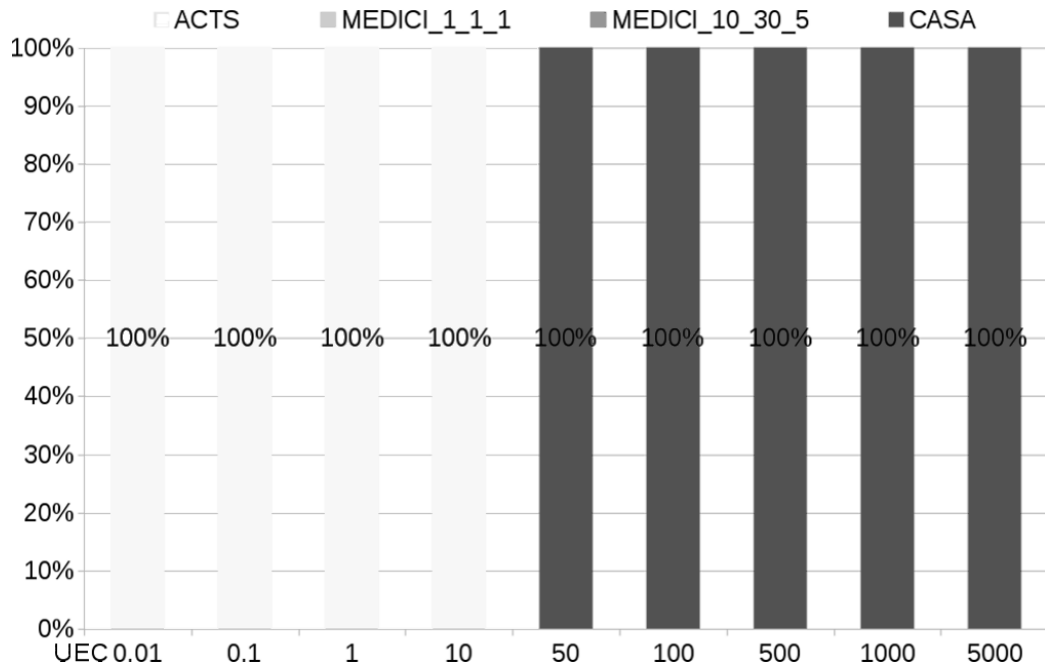


Figure 9.8: Generation tools distribution using CBT

Table 9.3 reports the total cost over all the combinatorial models for the optimum predictor in seconds and the percentage of variation w.r.t. the optimum for all the other generators by varying the single test cost. The numbers in bold are the minimum values for each cost with the exception of the optimum. It shows that some generators performs very well when the single test cost is small and other ones performs very well when the single test cost increases. It shows that FBT outperforms fixed generator selection in 7 cases over 9. In only one case, the FBT is defeated by CBT. For an UEC of 50 both the decision trees are defeated by MEDICI_1_1_1 that is never selected by CBT. The gap between both decision trees and the optimum a-posteriori predictor makes the possibility of further improvements, in the prediction strategy, concrete.

RQ5 How much is the gain achieved by using decision trees?

The performances of the 2 decision trees are reported as the Total cost for generating the test-suites of the 114 models varying their single test cost. Tab. 9.3 shows the performances of each strategy compared to the optimal one (last row) varying the single test cost.

Fig. 9.9 shows the performances of the two predictors CBT e FBT versus the use of a fixed generation tool. The data are presented as percentage of

Table 9.3: Total Mean Cost varying single test cost

	Single test cost – UEC (seconds)									
	0.01	0.1	1	10	50	100	500	1000	5000	
Optimum	87.79	388.75	3349.11	32268.57	159313.06	317345.41	1573761.56	3136876.52	15635276.52	
ACTS	21.14%	5.49%	2.94%	4.82%	5.97%	6.37%	7.23%	7.59%	7.93%	
CASA	16770.17%	3783.26%	436.13%	44.26%	8.99%	4.77%	1.88%	1.75%	1.69%	
MEDICI_10_30_5	8900.82%	2006.81%	230.68%	23.71%	5.52%	3.47%	2.32%	2.42%	2.54%	
MEDICI_1_1_1	1231.65%	277.01%	32.32%	5.65%	4.14%	4.21%	4.78%	5.10%	5.40%	
FBT	21.11%	5.48%	14.66%	2.93%	4.72%	2.56%	1.33%	1.41%	1.52%	
CBT	21.11%	5.48%	2.94%	4.82%	8.99%	4.77%	1.88%	1.75%	1.69%	
Random	6567.49%	1586.49%	201.61%	18.99%	6.25%	4.94%	4.14%	4.22%	4.39%	

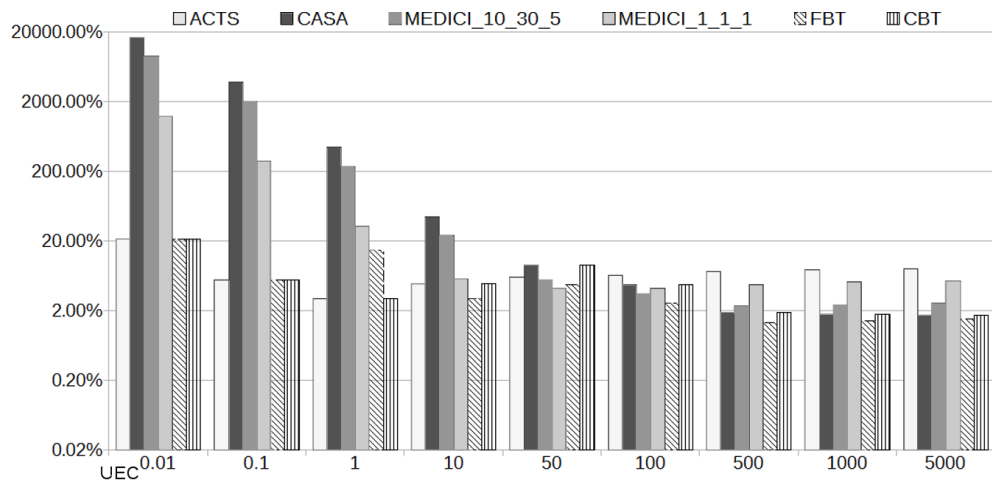


Figure 9.9: Performance comparison of Predictors versus the use of a fixed tool using the optimum as measure of comparison

the difference between the total cost for a single generator or predictor and the total cost obtained by the optimum predictor.

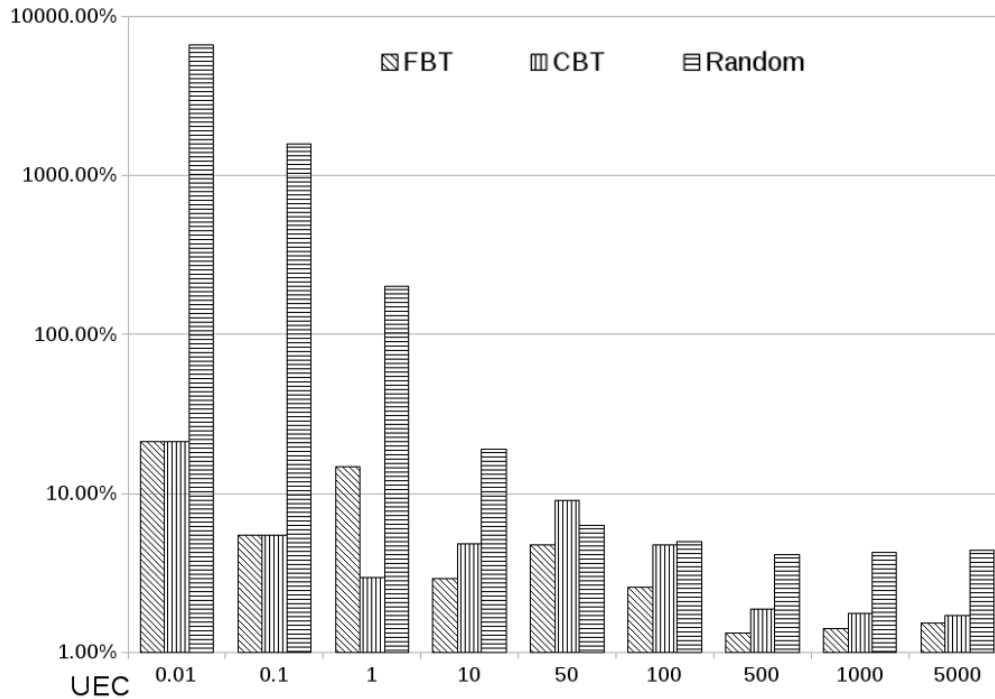


Figure 9.10: Performance comparison of predictors versus random selection of generators using the optimum as measure of comparison

RQ6 How much are decision trees better than a random selection?

We have compared decision trees with a random selector that randomly chooses the test generators. A random selector has the advantage that it does not need anything to make the prediction. The decision trees strategies always outperform a random selection policy as shown in Fig. 9.10. The gap of performance decreases with the augment of the single test cost but it remains significant if compared to the results of the optimum.

9.7 Threats to validity

Our findings are subject to the following threats to validity. First, our decision tree models may fail to predict the right test generator for a particular model and UEC. Indeed, our models can give only a statistical estimation, but they cannot guarantee to find the best solution. However, the wide range of models we have used to build the decision trees and the extensive set of UECs can give a good confidence that the right generator is most likely chosen and even if this is not the case, the loss of time is always rather small.

Even CBT, as shown in Table 9.3, behaves on average of all the models as the best fixed generator for almost each UEC. One of the most significant issues that afflicts decision tree models is the over-fitting of training data which, produces a loss of performance on new data. In general, when a decision tree model is too complex is unable to correctly match new, previously unseen data. The process that governs the complexity of a model is "Pruning". We have tried to reduce the impact of this issue iteratively pruning our models in order to find the less complex model that presents and acceptable error rate according to *Cost complexity pruning* methodology.

9.8 Conclusions

The combinatorial testing community has produced many algorithms, techniques, and tools for test generation in these years. Even if one considers only the performances, it remains unclear which is the best solution. For example, a very fast tool may produce a very big test suite that would require much more time during the test execution. We have introduced a simple cost model for comparing test generation tools and we have shown that there is no best combinatorial tests generator. Depending on the cost of executing a single test, a tool may be more suitable than another. In addition, combinatorial model characteristics should play a role in the choice of the generator. In order to automatize the decision of the right test generator, we have devised a data-mining process able to produce two decision trees to be used in the choice. Experimental results show that our decision trees can efficiently help the tester in the generation process with a significant reduction of the total testing cost. However we can isolate two points that are not worthwhile for our approach adoption. The first weakness of our methodology is that our predictor models take some data as inputs like number of variables, number of constraints, and so on, that may require some time to be computed. The time spent to find model features may be greater than the time saved by choosing the right generator. For this reason, we have chosen only very simple model characteristics and we have devised the CBT model, which requires only UEC. Moreover, we have ignored **N.valid** as input because it is rather costly to be computed. Computing all the model features used by FBT requires around 3.5 seconds for all the models.

The cost of switching to a general framework as CITLAB that allows the use of multiple generators may be not worthwhile for researchers using already a specific tool. In addition in this case, our work can be used as guidance to check if the chosen tool is suitable for the testing tasks to be performed. For instance, a serious loss of time is very likely to occur if the

tester uses ACTS for generating tests which require a lot of time each to be executed. Our study shows that the random choice can lead to very high time losses.

We plan to investigate on the influence of the complexity of constraints over the generation time and over the size of the test-suites. The cost function we have used in this chapter, could be improved by considering the complexity of the constraints and the cardinality of the parameters that compose the domains of the models under test. Another factor to consider could be the number of attempts needed to produce the final test-suite. When the system under test is very complex, it is necessary to run the generation tools many times during the different stages of software testing process. We want also to experiment other predictive techniques like *support vector machines* which seem, by a preliminary study, to have a more accuracy in terms of classification but they need further tunings to be deployed in Java.

10

Results achieved by CITLAB

Conclusion

The research activity I performed, during my period as *PhD* student, demonstrated that CITLAB is a good Framework for Combinatorial Interaction Testing, both for researcher and practitioners. In the previous chapters we presented the effectiveness of this framework through the description of its use in several research projects.

The timeline of the publications derived by the research activity on CITLAB is reported below.

CITLAB Publications Timeline

- 2012 ● Citlab: a laboratory for combinatorial interaction testing
- 2013 ● Combinatorial interaction testing with CitLab
- 2013 ● Combinatorial testing for feature models using citlab
- 2014 ● Validation of models and tests for constrained combinatorial interaction testing
- 2014 ● Efficient combinatorial test generation based on multivalued decision diagrams
- 2015 ● Using decision trees to aid algorithm selection in combinatorial interaction tests generation

CITLAB demonstrates its good usability as educational tool in university

courses. It used in the following two courses:

- **Model-based testing (NSWI157)** at *The Univerzita Karlova*
- **Software Testing and Check** at *The Università degli studi di Bergamo*.

Also, a team of students of *The Carnegie Mellon University* working on the NIST project on Combinatorial Testing released **ComTest**, an Open Source plugin for Eclipse based on CITLAB architecture.

The continued participation in the *IWCT* workshop, starting from the edition of 2013, made CITLAB known to *Combinatorial Testing Community* and facilitated its diffusion as educational tool as reported in the following timeline.

CITLAB EDU adoption Timeline

- 2013-2014 • CMU-NIST **ComTest** project
- 2013-2014 • Software Testing and Check
- 2015-2016 • Model-based testing

The current and future research activities, based on CITLAB, are focused on the problem of the definition of *Oracles* for combinatorial testing and on Specification mutation. A model of a system *configurability* must be validated in order to guarantee its conformance with respect to the software implementation. My future works will be focused on checking if a model correctly identifies constraints among the various software system's parameters in order to detect and fix faults both in the model and in the real system.

Bibliography

- [1] Advanced Combinatorial Testing System (ACTS).
- [2] Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. Automatic review of Abstract State Machines by meta property verification. In César Muñoz, editor, *Proceedings of the Second NASA Formal Methods Symposium (NFM 2010)*, NASA/CP-2010-216215, pages 4–13, Langley Research Center, Hampton VA 23681-2199, USA, April 2010. NASA.
- [3] Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. A model advisor for NuSMV specifications. *Innovations in Systems and Software Engineering*, 7(2):97–107, 2011.
- [4] 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.
- [5] Junaid Babar and Andrew Miner. Meddly: Multi-terminal and edge-valued decision diagram library. In *7th International Conference on the Quantitative Evaluation of Systems*. IEEE, 2010.
- [6] D. Batory. Feature models, grammars, and propositional formulas. *Software Product Lines*, pages 7–20, 2005.
- [7] Ilan Beer, Shoham Ben-David, Cindy Eisner, and Yoav Rodeh. Efficient detection of vacuity in ACTL formulas. In *Proc. 9th International Computer Aided Verification Conference*, number 1254 in Lecture Notes in Computer Science, pages 279–290, 1997.
- [8] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615–636, 2010.

-
- [9] J. Bézivin, H. Brunelière, F. Jouault, and I. Kurtev. Model engineering support for tool interoperability. In *The 4th Workshop in Software Model Engineering (WiSME'05)*, Montego Bay, Jamaica, 2005.
- [10] 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.
- [11] Renée C. Bryce and Charles J. Colbourn. Prioritized interaction testing for pair-wise coverage with seeding and constraints. *Information & Software Technology*, 48(10):960–970, 2006.
- [12] Renée C. Bryce and Charles J. Colbourn. One-test-at-a-time heuristic search for interaction test suites. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, GECCO '07, pages 1082–1089, New York, NY, USA, 2007. ACM.
- [13] Renée C. Bryce, Charles J. Colbourn, and Myra B. Cohen. A framework of greedy methods for constructing interaction test suites. In *ICSE '05: Proc. of the 27th int. conf. on Software engineering*, pages 146–155, New York, NY, USA, 2005. ACM.
- [14] 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.
- [15] 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.
- [16] Andrea Calvagna and Angelo Gargantini. A formal logic approach to constrained combinatorial testing. *Journal of Automated Reasoning*, 45(4):331–358, 2010. Springer.
- [17] Andrea Calvagna and Angelo Gargantini. T-wise combinatorial interaction test suites construction based on coverage inheritance. *Software Testing, Verification and Reliability*, Special issue on Model Based Testing, 2011. JohnWiley&Sons.
- [18] Andrea Calvagna and Angelo Gargantini. T-wise combinatorial interaction test suites construction based on coverage inheritance. *Software Testing, Verification and Reliability*, 22(7):507–526, 2012.

-
- [19] 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.
- [20] Andrea Calvagna, Angelo Gargantini, and Paolo Vavassori. Combinatorial testing for feature models using CitLab. In *Int. Workshop on Combinatorial Testing (IWCT)*, pages 338–347. IEEE Computer Society, 2013.
- [21] CASA: Covering arrays by simulated annealing.
- [22] V. Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3), 1979.
- [23] 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, 1997.
- [24] M.B. Cohen, M.B. Dwyer, and J. Shi. Coverage and adequacy in software product line testing. In *Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis*, pages 53–63. ACM, 2006.
- [25] 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.
- [26] Myra B. 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*, pages 129–139, New York, NY, USA, 2007. ACM Press.
- [27] Charlie Colbourn. Covering array tables.
- [28] J. Czerwonka. Pairwise testing in real world. In *24th Pacific Northwest Software Quality Conference*, 2006.
- [29] 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.

-
- [30] 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.
- [31] I. S. Dunietz, W. K. Ehrlich, B.D. Szablak, C.L. Mallows, and A. Ianino. Applying design of experiments to software testing. In IEEE/Computer Society, editor, *Proc. Int'l Conf. Software Eng. (ICSE)*, pages 205–215, 1997.
- [32] Bruno Dutertre and Leonardo de Moura. The Yices SMT solver. Technical report, SRI Available at <http://yices.csl.sri.com/tool-paper.pdf>, 2006.
- [33] E. Engström and P. Runeson. Software product line testing—a systematic mapping study. *Information and Software Technology*, 53(1):2–13, 2011.
- [34] 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.
- [35] 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. Springer International Publishing, 2014.
- [36] B.J. Garvin, M.B. Cohen, and M.B. Dwyer. An improved meta-heuristic search for constrained interaction testing. In *Search Based Software Engineering, 2009 1st International Symposium on*, pages 13–22. IEEE, 2009.
- [37] Brady J. Garvin, Myra B. Cohen, and Matthew B. Dwyer. An improved meta-heuristic search for constrained interaction testing. In *Proceedings of the 2009 1st International Symposium on Search Based Software Engineering, SSBSE '09*, pages 13–22, Washington, DC, USA, 2009. IEEE Computer Society.
- [38] Brady J. Garvin, Myra B. Cohen, and Matthew B. Dwyer. Evaluating improvements to a meta-heuristic search for constrained interaction testing. *Empirical Software Engineering*, 16(1):61–102, 2011.

-
- [39] Mihaela Gheorghiu and Arie Gurfinkel. VaqUoT: A tool for vacuity detection. In *Posters & Research Tools Track, FM 2006*, 2006.
- [40] Mats Grindal, Jeff Offutt, and Sten F. Andler. Combination testing strategies: a survey. *Softw. Test, Verif. Reliab*, 15(3):167–199, 2005.
- [41] Tarik Hadzic, Esben Rune Hansen, and Barry O’Sullivan. On automata, MDDs and BDDs in constraint satisfaction. In *Proceedings of the ECAI 2008 Workshop on Inference Methods based on Graphical Structures of Knowledge*, 2008.
- [42] Mark A Hall. *Correlation-based feature selection for machine learning*. PhD thesis, The University of Waikato, 1999.
- [43] Mary Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2(3):270–285, 1993.
- [44] Alan Hartman. Ibm intelligent test case handler: Whitch.
- [45] Alan Hartman. *Graph Theory, Combinatorics and Algorithms Interdisciplinary Applications*, chapter Software and Hardware Testing Using Combinatorial Covering Suites, pages 237–266. Springer, 2005.
- [46] Alan Hartman and Leonid Raskin. Problems and algorithms for covering arrays. *DMATH: Discrete Mathematics*, 284(1-3):149–156, 2004.
- [47] Constance Heitmeyer, Ralph D. Jeffords, and Bruce G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, July 1996.
- [48] Aymeric Hervieu, Benoit Baudry, and Arnaud Gotlieb. PACOGEN: Automatic generation of pairwise test configurations from feature models. In *Proc. of Int. Symp. on Soft. Reliability Engineering (ISSRE’11)*, 2011.
- [49] Bernardo A. Huberman, Rajan M. Lukose, and Tad Hogg. An economics approach to hard computational problems. *Science*, 275:51–54, 1997.
- [50] Yue Jia, Myra B. Cohen, Mark Harman, and Justyna Petke. Learning combinatorial interaction testing strategies using hyperheuristic search. Research Note RN/13/17, University College London, Department of Computer Science, 2013.

-
- [51] M.F. Johansen, Ø. Haugen, and F. Fleurey. An algorithm for generating t-wise covering arrays from large feature models. In *Proceedings of the 16th International Software Product Line Conference-Volume 1*, pages 46–55. ACM, 2012.
- [52] 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.
- [53] 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.
- [54] R. Kuhn, R. Kacker, Yu Lei, and J. Hunter. Combinatorial software testing. *Computer*, 42(8):94–96, aug. 2009.
- [55] Orna Kupferman and Moshe Y. Vardi. Vacuity detection in temporal model checking. *International Journal on Software Tools for Technology Transfer (STTT)*, 4(2):224–233, 2003.
- [56] Michail G Lagoudakis and Michael L Littman. Algorithm selection using reinforcement learning. In *ICML*, pages 511–518, 2000.
- [57] Yu Lei, Raghu Kacker, D. Richard Kuhn, Vadim Okun, and James Lawrence. IPOG/IPOG-D: efficient test generation for multi-way combinatorial testing. *Software Testing, Verification and Reliability*, 18(3):125–148, September 2008.
- [58] Yu Lei, Raghu Kacker, D. Richard Kuhn, Vadim Okun, and James Lawrence. IPOG/IPOG-D: efficient test generation for multi-way combinatorial testing. *Software Testing, Verification and Reliability*, 18(3):125–148, September 2008.
- [59] C. Lott, A. Jain, and S. Dalal. Modeling requirements for combinatorial software testing. In *A-MOST '05: Proceedings of the 1st international workshop on Advances in model-based testing*, pages 1–7, New York, NY, USA, 2005. ACM Press.
- [60] Inês Lynce and João P. Marques-Silva. On computing minimum unsatisfiable cores. In *Online Proceedings of SAT 2004, 10-13 May 2004, Vancouver, BC, Canada*, 2004.

-
- [61] Marcilio Mendonca, Moises Branco, and Donald Cowan. S.P.L.O.T.: software product lines online tools. In *Proceedings of the 24th ACM SIG-PLAN conference companion on Object oriented programming systems languages and applications*, OOPSLA '09, pages 761–762, New York, NY, USA, 2009. ACM.
- [62] S. Nagayama and T. Sasao. Compact representations of logic functions using heterogeneous MDDs. In *Multiple-Valued Logic, 2003. Proceedings. 33rd International Symposium on*, pages 247–252, 2003.
- [63] Changhai Nie and Hareton Leung. A survey of combinatorial testing. *ACM Comput. Surv.*, 43(2):11, 2011.
- [64] 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.
- [65] Sebastian Oster, Florian Markert, and Philipp Ritter. Automated incremental pairwise testing of software product lines. In Jan Bosch and Jaejoon Lee, editors, *Software Product Lines: Going Beyond*, volume 6287 of *Lecture Notes in Computer Science*, pages 196–210. Springer Berlin Heidelberg, 2010.
- [66] Pairwise web site. <http://www.pairwise.org/>.
- [67] G. Perrouin, S. Sen, J. Klein, B. Baudry, and Y. Le Traon. Automated and scalable t-wise test case generation strategies for software product lines. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 459–468. IEEE, 2010.
- [68] Gilles Perrouin, Sebastian Oster, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves Le Traon. Pairwise testing for software product lines: comparison of two approaches. *Software Quality Journal*, 20(3-4):605–643, 2012.
- [69] Steffen Prochnow, Gunnar Schaefer, Ken Bell, and Reinhard von Hanxleden. Analyzing robustness of UML State Machines. In *MARTES 06*, pages 61–80, 2006.
- [70] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2008. ISBN 3-900051-07-0.

-
- [71] John R. Rice. The algorithm selection problem. *Advances in Computers*, 15:65–118, 1976.
- [72] Elke Salecker, Robert Reicherdt, and Sabine Glesner. Calculating prioritized interaction test sets with constraints using binary decision diagrams. In *Proceedings of IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 278–285. IEEE Computer Society, 2011.
- [73] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. Generic semantics of feature diagrams. *Computer Networks*, 51(2):456 – 479, 2007.
- [74] 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.
- [75] Gadiel Seroussi and Nader H. Bshouty. Vector sets for exhaustive testing of logic circuits. *IEEE Transactions on Information Theory*, 34(3):513–522, 1988.
- [76] Colin Shearer. The crisp-dm model: the new blueprint for data mining. *Journal of data warehousing*, 5(4):13–22, 2000.
- [77] 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.
- [78] K. C. Tai and Y. Lie. A test generation strategy for pairwise testing. *IEEE Trans. Softw. Eng.*, 28(1):109–111, 2002.
- [79] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming*, 2012. to appear.
- [80] Alan W. Williams and Robert L. Probert. Formulation of the interaction test coverage problem as an integer program. In *Proceedings of the 14th International Conference on the Testing of Communicating Systems (TestCom) Berlin, Germany*, pages 283–298, march 2002.

-
- [81] G. Williams. *Data Mining with Rattle and R*. NY: Springer New York, 2011.
- [82] Xtext. <http://www.eclipse.org/xtext/>.
- [83] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: Portfolio-based algorithm selection for SAT. *J. Artif. Intell. Res. (JAIR)*, 32:565–606, 2008.
- [84] Jian Zhang, Zhiqiang Zhang, and Feifei Ma. *Automatic Generation of Combinatorial Test Data*. SpringerBriefs in Computer Science. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014. 00000.