Federal University of Rio Grande do Norte

Department of Informatics and Applied Mathematics (DIMAp)

Doctoral Thesis

Cleverton Hentz Antunes

ENTITLED

A Family of Coverage Criteria Based on Patterns to the Test of Metaprograms

ADVISOR: DR. ANAMARIA MARTINS MOREIRA

Natal, RN, Brazil

March, 2018

Cleverton Hentz Antunes

A Family of Coverage Criteria Based on Patterns to the Test of Metaprograms

Thesis submitted to the Program of Pós-Graduação em Sistemas e Computação do Departamento de Informática e Matemática Aplicada da Universidade Federal do Rio Grande do Norte as a requirement to obtain the degree of Doctor in Computer Science.

Advisor: Dr. Anamaria Martins Moreira

UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE CENTRO DE CIÊNCIAS EXATAS E DA TERRADEPARTAMENTO DE INFORMÁTICA E MATEMÁTICA APLICADA PROGRAMA DE PÓS-GRADUAÇÃO EM SISTEMAS E COMPUTAÇÃO

Natal, RN, Brazil

March, 2018

Universidade Federal do Rio Grande do Norte - UFRN
Sistema de Bibliotecas - SISBI
Catalogação de Publicação na Fonte. UFRN - Biblioteca Setorial Prof. Ronaldo Xavier de Arruda - CCET

Antunes, Cleverton Hentz.
A family of coverage criteria based on patterns to the test of metaprograms / Cleverton Hentz Antunes. - 2018.
138 f.: il.

Tese (doutorado) - Universidade Federal do Rio Grande do Norte, Centro de Ciências Exatas e da Terra, Programa de Pós-Graduação em Sistemas e Computação. Natal, RN, 2018.
Orientadora: Anamaria Martins Moreira.

1. Teste de software - Tese. 2. Critérios de cobertura de teste - Tese. 3. Casamento de padrões - Tese. 4. Metaprogramas - Tese. I. Moreira, Anamaria Martins. II. Título.

RN/UF/CCET CDU 004.415.53

Elaborado por JOSENEIDE FERREIRA DANTAS - CRB-15/324

A Family of Coverage Criteria Based on Patterns to the Test of Metaprograms

Cleverton Hentz Antunes

Doctoral Thesis approved on the 15th of December 2017 by the committee composed by the following members:

Thiſ theſiſ iſ dedicated to my parentſ and my ſonſ,
for their ſupport and endleſſ love.

Acknowledgements

The thesis is the result of my Ph.D. project at the Dimap/UFRN in Natal. During this time I met a lot of friendly people. Many of them deserve my thanks.

First of all, I would like to thank my advisor Prof. Anamaria Moreira for accepting me at the Ph.D. position, and for supporting and guiding me throughout the project. I have learned a lot from all her comments and advices. Furthermore, it has been a long journey since 2008 when I started my Master project with her as my advisor. She was always available to help me when I was stuck.

Second, I would like to thank Prof. Jurgen Vinju for supervising my work during my time at SWAT in the Centrum Wiskunde & Informatica, Amsterdam. The support provided by him had a significant influence on this project. Furthermore, he kindly received me at SWAT group and always encouraged the development of this project.

Many thanks as well to all my other colleagues from SWAT: Ali Afroozeh, Bas Basten, Riemer van Rozen, Jouke Stoel, Cigdem Aytekin, Pablo Inostroza Valdera, Davy Landman, Anastasia Izmaylova, Michael Steindorfer, Paul Klint, and Tijs van der Storm. All these persons have some influence and help during my time in CWI.

I also would like to especially thank my friends Marcelo Damasceno and Isaac de Lima Oliveira Filho from Dimap/UFRN for all the help, talks and beers. Many thanks to my colleagues and staff at Dimap department.

I also would like to thank all Brazilian taxpayers that founded my Ph.D. internship at CWI by the CNPq and INES founding agencies.

Finally, I thank my parents and, especially, my sons for always give a lot of happy moments in my life. I would also like to thank my girlfriend, Viviane, for always believe in me and to encourage to go forward to reach my goals.

"Mind precedeſ all mental ſtateſ.
Mind iſ their chief; they all mind-wrought.
If with a pure mind a perſon ſpeakſ or actſ
happineſſ followſ him like hiſ
never-departing ſhadow - 2. "

THE DHAMMAPADA, 1985
The Buddha's Path of Wisdom
Translated from the Pali by Acharya Buddharakkhita

"Anybody can play weird, that'ſ eaſy.
What'ſ hard iſ to be aſ ſimple aſ Bach.
Making the ſimple complicated iſ commonplace.
Making the complicated ſimple – aweſomely
ſimple – that'ſ creativity."

CHARLES MINGUS, 1972

Resumo

Apesar da existência de várias técnicas para a geração automática de dados de teste baseados em gramáticas, poucos trabalhos foram propostos no sentido de melhorar os dados de teste gerados aplicando restrições semânticas. Nesse sentido, contribuimos neste trabalho, nesta direção para o caso particular do teste de metaprogramas, um programa que tem como dado de entrada um outro programa. Atualmente, a alternativa natural para o teste deste tipo de programa é a técnica de testes baseados em gramáticas. Essa técnica de teste pode ser aplicada de maneira relativamente simples, porém com um custo de geração e execução do conjunto de testes elevado e uma efetividade baixa. Por outro lado, diversas pesquisas e ferramentas de grupos interessados no desenvolvimento de metaprogramas fazem uso intenso do recurso de casamento de padrões durante a sua implementação e especificação. Neste caso, os padrões oferecem uma fonte de informação interessante para a criação de testes que são sintaticamente válidos e também satisfazem restrições semânticas. Dada a limitação dos testes baseados em gramáticas e considerando a informação embutida nos padrões existentes nos metaprogramas, temos a oportunidade de contribuir para a melhoria do processo de teste para esses programas. Logo, o objetivo deste trabalho é avaliar o uso da informação de padrões para o teste de metaprogramas e com isso contribuir no seu processo de teste. No intuito de sistematizar o processo de teste de software, o documento apresenta um processo de *design* de teste e também uma família de critérios de cobertura baseados em padrões para o teste eficiente e sistematizado de metaprogramas baseados em padrões. São propostos quatro critérios de cobertura baseados em padrões e nos critérios de particionamento do espaço de entrada. Também é apresentada uma hierarquia entre os critérios propostos. Com isso, diferentes níveis de rigor podem ser requeridos no processo através da escolha do critério adequado. A validação dessas contribuições é feita através de um estudo de caso e uma validação empírica. O estudo de caso apresenta uma instanciação de referência para o processo de *design* de testes de um verificador de tipos implementado como metaprograma baseado em padrões. O verificador de tipos é testado usando um conjunto de testes gerados pelos critérios de cobertura baseados em padrões. A qualidade desse conjunto é avaliada utilizando a técnica de mutação e através da comparação dos resultados obtidos com testes gerados por critérios baseados em gramáticas. Os estudos experimentais indicam a efetividade da aplicação dos critérios baseados em padrões e o ganho em custo-benefício em relação aos critérios baseados em gramáticas no contexto do teste de metaprogramas baseados em padrões.

Palavras-chaves: Teste de Software, Critérios de Cobertura de Teste, Casamento de Padrões, Metaprogramas, Linguagens Formais.

Abstract

Although there are several techniques for the automatic generation of test data based on grammars, few studies have been proposed to improve the test data generated by applying semantic restrictions. In this sense, we intend to contribute in this direction for the particular case of testing metaprograms, programs that have other programs as input. Currently, the natural alternative to testing this kind of program is using the grammar-based testing. That testing technique can be applied relatively easily, but with high costs, related to the generation and execution of the test set, and low effectiveness. Many researches and tools dedicated to the development of metaprograms make heavy use of pattern matching for their implementation and specification. In this case, the patterns offer an interesting source of information for creating tests that are syntactically valid and also satisfy semantic constraints. Given the limitation of grammar-based testing and pattern information on the metaprograms, we identified an opportunity to contribute to the improvement of the testing process for these programs. Therefore, the goal of this work is to evaluate the use of pattern information for the testing of pattern-based metaprograms and thus contribute to their testing process. In order to systematize the software testing process, a family of coverage criteria based on patterns is proposed to test pattern-based metaprograms efficiently and systematically. Four pattern-based coverage criteria are proposed, they are based on classical input space partitioning combination criteria. Furthermore, a hierarchical relationship between the criteria is presented. Therefore, different levels of rigor can be required by choosing the appropriate criterion. The validation of these contributions is made using a case study and empirical validation. The case study presents a reference instantiation for the test design process applied to a type checker implemented as metaprogram based on patterns. The type checker is tested using a test set generated by the pattern-based coverage criteria, and the quality of this set is evaluated using the mutation technique. The results obtained are compared with those produced by a test set generated by the grammar-based criteria. The experimental studies indicate the effectiveness of the application of these pattern-based criteria and a gain of cost-return in relation to the grammar-based coverage criteria.

Keywords: Software Testing, Coverage Criteria, Pattern Matching, Metaprograms, Formal Languages.

Contents

List of Figures

List of Tables

List of Listings

— 1 —

Introduction

Metaprograms are tools which read sentences of software languages, such as programming languages, and produce any kind of output [Czarnecki and Eisenecker, 2000]. There are many kinds of metaprograms, for example compilers, interpreters, refactoring tools, static analysis tools, and source code metrics tools. The amount and diversity of such tools are growing as processing power and large memory become available to the machines on which software is being developed, nevertheless developing and verifying such tools is challenging. This is because metaprograms deal with programs, with their intrinsic variety and complexity, as input data. An additional level of difficulty is related to the need to deal with different levels of abstractions (programs and metaprograms) [Sheard, 2001].

A way to deal with the metaprogram complexity is using the pattern matching mechanism [Hoffmann and O'Donnell, 1982]. Patterns offer a simple mechanism to describe specific constructions and structures over the metaprogram's input language, thereby metaprograms can reach their goals by using a simple implementation. This approach also produces a case oriented implementation and specification of metaprograms. This feature can be used in addition with a context-free grammar to define the metaprograms' input data. The use of patterns and context-free grammar features together is quite common in the development of metaprograms. We will call this kind of metaprograms *pattern-based metaprograms* (PBMP). The pattern matching feature exists in many metaprogram development environments, like ASF+SDF [van den Brand et al., 2001], StrategoXT [Bravenboer et al., 2008] and Rascal[Klint et al., 2009, 2011]. Furthermore, this feature is also present in some specification formalisms based on algebraic specifications, such as Maude [Clavel et al., 2003] and ELAN [Borovanský et al., 1996]. Although the pattern matching feature is present in those environments and formalisms, each of them have a specific implementation and set of pattern constructions. It is also common the use of patterns to create metaprograms using traditional functional languages. Nevertheless, these languages are general purpose languages and not designed to construct only metaprograms. This lack of design aiming the development of metaprograms is also true for the algebraic specifications mentioned early. Functional languages also have support to the pattern matching feature in general. Some examples of these functional languages with support to patterns are Haskell [Hudak et al., 1992b] and Scala [Odersky et al., 2004]. Even imperative languages can have support for pattern based programming. For example, the Java language can have this feature provided by TOM [Balland et al., 2007] which adds patterns on top of the language.

The development of metaprograms is a complex task and the verification of these programs is even more difficult. For example, even simple metric tools are known to contain many bugs [Lincke et al., 2008]. Metaprograms for real programming languages are complex and hard to prove or test because the languages are big (more than 400 context-free grammar rules is quite normal) and their semantics are often unclear. Apart from some notable exceptions [Leroy, 2009], proofs of correctness of metaprograms are not to be expected. As we will present in Section 1.1, there is no standard verification method used to the metaprograms. So, we tried to analyze some existent methods from different contexts to see the viability of them and focusing on the testing process. A related kind of programs that use testing as verification are the compilers, which can be classified as a kind of metaprogram [Aho et al., 2007]. Because of it, some compilers' testing techniques could be considered as alternative to the testing of metaprograms.

The traditional test techniques based on a graph representation of a program under test, graph coverage criteria [Ammann and Offutt, 2008], are difficult to be applied on pattern-based metaprograms (PBMP). The traditional coverage criteria on this category, control flow coverage and data flow coverage, are difficult to be used to the test of PBMP because the construction of a graph representation for these metaprograms is a complex task [Adsit and Fluet, 2014; Hills, 2014]. An important step for these techniques is to solve constraints in a graph representation and from it define the path to execute these constraints on the program under test. In a traditional program, there are many constraint solvers and algorithms implemented to solve this kind of problems [Monniaux, 2016], but for metaprograms to use the same approach is quite challenging. One important difficulty over the use of constraint solvers with metaprograms is to solve constraints over patterns. To our knowledge, there is no constraint solver project specifically with support to patterns. A possible solution can be adapting a string constraint solver [Golden and Pang, 2003] to solve pattern constraints. This approach is not simple either because there has been little work on constraint reasoning over strings [Kiezun et al., 2009]. To our knowledge, there is no tool that can be used directly on the test of metaprograms built with patterns. We believe that this kind of implementation is essential to the use of these graph related techniques on the PBMP.

A frequently used approach to the test of compilers is the grammar-based testing technique. *Grammar-Based Testing* [Lämmel and Schulte, 2006] is a technique which uses a formal specification of the software input data, a context-free grammar, to generate the test data. An automatic or semi-automatic generation process can be defined using a grammar and this process can produce valid and invalid sentences in relation to the grammar. This kind of generation process can be easily found in the literature [Boujarwah and Saleh, 1997; Kossatchev and Posypkin, 2005]. Another kind of generation techniques is based on the semantic information related to the input language. This semantics can be defined formally and it can be used to generate test data for the software under test. An example of this semantics-based approach is presented by Bazzichi and Spadafora [1982], their approach uses a context-free parametric grammar to formalize the input language's semantics and to generate the test data. There are many techniques to generate test data automatically based on grammars [Purdom, 1972; Lämmel, 2001; Bazzichi and Spadafora, 1982; Zelenov and Zelenova, 2005; Lämmel and Schulte, 2006], but few of them have been proposed the automatic generation of test data using the semantic information related to the language defined by the grammar. The first research effort related to the test based on grammars was developed with the objective of testing compilers, this group of works was directed to test of the compilers of Cobol, Fortran and PL/1 languages [Sauder, 1962; Hanford, 1970]. These grammar-based techniques have been vastly researched and have a high maturity level on the test of compilers. Because of these characteristics, the grammar-based testing techniques are often considered for the test of metaprograms.

The grammar-based test can be applied to testing of pattern-based metaprograms (PBMP) by the use of their input grammar on the grammar-based test technique, a similar approach is used on the compilers' testing. With this approach, we can produce a test set of syntactically valid input programs that can be used to test the PBMP [Ammann and Offutt, 2008]. The problem with this strategy is the low probability of a test case execute a pattern related code in a PBMP. Besides the presented characteristics, there is the main problem on the direct application of the grammar-based testing techniques on the PBMP testing in general. The grammar-based testing techniques were created to cover a large part of the input language used for the software under test, for example, the use of this technique by the compilers. This grammar-based approach is conceptually ineffective to the test of PBMP in general. For metaprograms in general, the total coverage of the input language or its grammar is not essential in many cases. Many metaprograms are made to deal with parts of the input language and treating only specific situations that are important to the metaprogram's final objective [Klint et al., 2011]. For the pattern-based metaprograms, the test should be directed to cover some parts of the input language, consequently only some parts of the grammar, defined by the metaprogram implementation. For compilers, for example, the test has to cover a large part of the input language [Aho et al., 2007]. Thus, the testing process of these pattern-based metaprograms should be directed to the specific interest defined by the metaprogram under test.

Analyzing a real programming language, for example Java [Gosling et al., 2005], we can illustrate the cost of using grammar-based techniques. For this kind of language, the number of sentences generated by its grammar is infinite. Analyzing the generation of the sentences for it, we have a set with 46.26×10^9 sentences only for the sentences with a derivation tree with the maximum high of 7. We can see the same complexity in another example, now with a simpler language defined by Lämmel and Schulte [2006]. This language is defined by an extraction of the C# grammar only with the arithmetical expressions defined. In this case, the number of sentences is around 883 million. The authors present some techniques to reduce the sentence set size. For the simplest cases reducing it from millions to ten sentences. The grammar analyzed by the authors has 21 nonterminals and 34 productions rules and the technique application for this case produces 200,000 test cases. However, the grammar analyzed is not complex enough to compare it with a real programming language.

As there is a lack of effective set of techniques to the testing of metaprograms, our goal is to contribute to the test process of metaprograms. The strategy is to use the pattern information which is related to these programs. This pattern information could be directly coded in the metaprogram under test or even extracted from an abstract specification of it. In this direction, there are two main contributions of this thesis: the pattern-based test design process, presented in Chapter 4, and the pattern-based coverage criteria family which support the proposed test design process, presented in Chapter 5. These two contributions aim to improve the systematization and quality of the overall testing process for pattern-based metaprograms.Furthermore, we believe that the integrated use of patterns in metaprogram source-code and a testing strategy that uses these patterns can contribute to the state of the art of the development and testing of these metaprograms. The proposed pattern-based test design process and the pattern-based coverage criteria can be used with pattern-based metaprograms, independently whether those patterns are present in the source code or in any other artifact related to this metaprogram. Our goal in this work is primarily related to the proposal and evaluation of the pattern-based testing of hard coded pattern-based metaprograms (i.e., when the patterns are present in the source code). However, in many points of the proposal, it is easy to see its generalization possibilities. We try to show them throughout the text. The detailed explanation about the motivation, goal and related research questions are presented in Sections 1.1, 1.2 and 1.3 respectively.

The pattern-based test design process proposes the necessary abstract steps to use the pattern information associated with pattern-based metaprograms. This information could be hard coded on the metaprogram itself or on another artifact associated with it. Once the pattern information is extracted, we can use one or more pattern-based coverage criterion to test the pattern-based metaprogram. The test design process could be used only to extract the patterns and generate the test requirement set. Additionally, it is possible to going further and generate the test data sets that satisfy the produced test requirement sets. The test data generation problem and an abstract reference algorithm are defined on Chapter 4. There is no mandatory test data algorithm implementation defined by the test design process, it must be defined at the instantiation moment.

Chapter 6 presents an evaluation of the test design process and the coverage criteria family. First, we present an instantiation of the test design process using the coverage criteria family in a case study. This case study is related to the test of a complex pattern-based metaprogram using the pattern-based coverage criteria test sets. This pattern-based metaprogram uses the patterns hard coded into its implementation. Consequently, the metaprogram patterns can be directly used by the proposed test design process and the pattern coverage criteria. The case study can be used as a reference to other instantiations. The test data generation algorithm used during the case study uses a grammar-based tool. Second, it is presented in the chapter a validation of the criteria family and a comparison between the pattern-based coverage criteria family and a grammar-based testing coverage criterion. Some considerations about the use of both techniques together are presented too.

The pattern-based coverage criteria family is composed of four coverage criteria, three of them are based on the equivalence partitioning [Myers et al., 2011] criteria family and one of them is not. The simplest and first elaborated, which defines a requirement for each pattern, was used as initial proof of concept to evaluate the viability of the pattern-based coverage criteria and the validation method for it [Hentz et al., 2015]. We also present a hierarchical classification of the coverage criteria in the family. This hierarchical classification shows the main goal of the coverage criteria family that provides different levels of rigor of the requirements and associated to each of them a different cost to satisfy these requirements. Using these criteria, the test of the pattern-based metaprograms can be constructed using a systematic strategy and different levels of rigor.

A second step in the proposal validation is the verification of the coverage criteria family. The verification was made to check the effectiveness of the criteria in a real usage situation and compare its results with other criteria based on grammar-based testing. The results of this step are also presented on Chapter 6. The criteria were evaluated using the mutation test technique [DeMillo et al., 1978] and uses the Rascal language as metaprogram development language. Rascal is a development environment which offers many features to develop metaprograms in an integrated way. The tool offers an integrated environment to the development, execution, test and debug of metaprograms. Besides that, Rascal provides a rich support to the pattern matching constructions. Because of these characteristics, we use Rascal as reference on this thesis. The choice of Rascal does not mean that the proposed work was constructed only to be applied with this language. The pattern-based test design process can be instantiated for any language with support to patterns and the coverage criteria family used on the specific language, the process and the instantiation are presented on Chapter 4. The pattern-based metaprogram under test used for the evaluation was the Rascal Type Checker which is written in Rascal itself. For each criterion on the pattern-based family, a related test set was generated and compared with another test set related to a grammar-based testing criterion. The result of the mutation process was collected for each test set, and the results are presented and discussed on the chapter.

1.1 Motivation

Metaprograms are a complex kind of software. The verification process for this kind of program is important to help the detection of faults as soon as possible [Sheard, 2001]. Some languages are used to elaborate metaprograms and they can help the development of this kind of programs, for example Haskell [Hudak et al., 1992a], Scala [Odersky et al., 2004], TOM [Balland et al., 2007], ASF+SDF [van den Brand et al., 2001], Maude [Clavel et al., 2003], StrategoXT [Bravenboer et al., 2008], ELAN [Borovanský et al., 1996], Prolog [Colmerauer and Roussel, 1996] and Rascal[Klint et al., 2009, 2011]. However, none of them presents a verification strategy integrated to the development process.

To know better the actual context of the verification process for metaprogram projects, we have researched about it on some projects in different metaprogramming languages. We used the Rascal, Haskell and TOM languages as a reference to look at practical metaprogram projects in order to have an overview of the actual verification process of the metaprograms projects. These language were used to do the research because we could find open-source projects or publications using these languages. It is important to note that there are few metaprogram projects. This is because the technology is relatively new and, consequently, it is difficult to find out open source projects or documentation about the verification process of these projects. An exception to this situation is the case of compilers, but in our view, it is interesting to present a broader perspective of the development project of metaprograms in general.

The first projects presented are related to the Rascal language. Three projects were chosen using this language: Rascal Type Checker, Oberon Type Checker, and OS-

SMETER. The Rascal type checker (RTC)[1] is implemented in Rascal itself and implements the language type system. Part of this project is used as a case study on this thesis and further information about it can be found in Section 6.2. The project uses testing as its primary verification process, and it is based mainly on the random testing using the QuickCheck [Claessen and Hughes, 2000] implementation on the Rascal language [Klint, 2013]. Part of the testing strategy is based on the bug related test cases, and others are based on the engineer specific knowledge. To give an idea of the size of test set used by the language, the entire Rascal project has 3,900 test cases [Klint, 1993]. The RTC, an important component of the language implementation, has 337 test cases [2].

The second analyzed project is the Oberon Type Checker (OTC)[3]. It is a type checker for the Oberon Language [Wirth, 1988] and it is implemented in Rascal as well. The project uses testing as verification strategy but without any specific test technique. The test cases consist of positive and negative Oberon programs to check for specific situations. Another metaprogram project is OSSMETER[4]. OSSMETER is an EU-founded research project that aims to develop a platform for monitoring the quality of open-source software projects in general. The project uses several programming languages from different paradigms. There is a module in the project that is implemented in Rascal[5]. The project uses testing as verification strategy but without any systematic or automated effort.

The project HaRe functional[6] is a refactoring tool developed in the Haskell functional language. The tool implements a set of refactoring rules directed to functional programming languages. Refactoring is about "improving the design of an existing code", and as such, it has been practiced as long as programs have been written [Li, 2006]. The project uses testing and formal specifications to verify its implementation [Li, 2006; Thompson et al., 2006]. The testing part is conducted using an ad-hoc strategy without any automation. Additionally, some test cases were created from detected bugs to improve the verification process.

Tom is a Java language extension designed to manipulate tree structures and XML documents [Balland et al., 2007]. The Tom compiler[7] is an example of metaprogram project that has been written in Tom itself and aims to compile Tom metaprograms. The primary verification approach on the project is testing but without a sys-

[1]The Rascal type checker is available on `https://github.com/usethesource/rascal/blob/master/src/org/rascalmpl/library/lang/rascal/types/`. The version used as reference is `4e9cd4e`.

[2]Based on the version `4e9cd4e` of the RTC test set.

[3]The OTC is available on `https://github.com/cwi-swat/oberon0`

[4]The OSSMETER is available on `https://github.com/ossmeter/ossmeter`

[5]This specific part of the project is available on `https://github.com/ossmeter/ossmeter/tree/dev/metric-providers/org.ossmeter.metricprovider.trans.rascal.OO`

[6]The HaRe project is available on `https://github.com/RefactoringTools/HaRe`.

[7]The Tom compiler project, is available on `https://github.com/rewriting/tom`.

tematic approach or strategy. Some bugs are coded as test cases, but the test set is outdated in comparison with the code. At the time of the writing, the last change in the test repository was on October 06 of 2016 (around seven months ago).

Project	Meta. Language	Verification	Comments
Rascal Type Checker	Rascal	Testing	Random Test, Ad-hoc test and bug related
Oberon Type Checker	Rascal	Testing	Ad-hoc test
OSSMETER	Multi-language	Testing	Ad-hoc test
HaRe	Haskell	Testing and formal verification	Ad-hoc test and bug related
TOM Compiler	TOM and Java	Testing	Ad-hoc test and bug related

Table 1.1: The metaprograms projects and its verification process information.

Table 1.1 summarizes the set of projects researched and its strategies related to the verification process. As we can see the majority of the projects uses tests as verification strategy, but in most cases without a systematic way. This information can give an insight into the verification process in others metaprogram projects and indicate the necessity of improvement over the verification process for this kind of programs. We can observe with these projects that none of them use the grammar-based and graph related techniques. These two techniques are evaluated as possible alternatives to the testing of metaprograms. This observation can be justified by many facts, but we do not investigate the motivation behind that. We believe that the high complexity of metaprograms [Sheard, 2001] and, consequently, the complexity of verify these programs contributes to it. So, we believe that providing a systematic testing process for the metaprograms is essential and beneficial.

1.2 The Thesis Goal

In general, metaprograms do not use a systematic approach to conduct a verification process. Many metaprogram projects use testing as verification strategy, but they do not use it with a specific approach. We believe that a test technique to support the testing process for the pattern-based metaprograms is a relevant contribution to the improvement of these programs. Patterns represent part of the semantics of these metaprograms, and they contribute to the testing of those metaprograms. Thus, the primary goal of this thesis is to contribute with a technique that uses pattern information related to the software under test, the pattern-based metaprograms, to generate test datasets and improve its test process. With this objective, the thesis proposes a new software test design process and a coverage criteria family both based on the pattern matching mechanism.

To achieve the goal, we define a set of subgoals related to it. Which are:

- Propose a test design process based on patterns to test pattern-based metaprograms;
- Propose a coverage criteria family based on patterns to support the process;
- Instantiate the process and use the coverage family to test a complex pattern-based metaprogram project and evaluate its results;
- Compare the results from the proposed coverage criteria family with the grammar-based coverage criteria.

1.3 The Hypotheses and Research Questions

In this section, based on the primary goal of the thesis, we are going to propose two hypotheses. The first hypothesis is related to the viability of using the pattern information to test pattern-based metaprograms. The second is related to the verification of a test process and techniques based on patterns in combination with the grammar-based testing techniques to the test of metaprograms. At the end of the section, we present a summary with the relation between the hypotheses and the research questions defined.

1.3.1 Hypothesis 1

In general, the metaprograms do not use a systematic approach to the verification process. As presented earlier, the projects that use tests as verification do not use it with a specific approach. Some testing strategies can be used to verify metaprograms, but the most natural of them is grammar-based testing. This technique has a limitation on its direct use with metaprograms, the cost of use is high in general. So, a test technique based on the metaprograms' characteristic of use specific parts of the input language could be used as differential over the use of a large part of the input language, as usual in the grammar-based techniques. Furthermore, the pattern information used in the development of these metaprograms can be a source of information from where these language parts can be identified and used. With this scenario in mind, our first research hypothesis is related to the test of metaprograms using the related pattern information about them.

Hypothesis 1. *The test design based on pattern information from pattern-based metaprograms contributes to the quality of the testing of these programs.*

The hypothesis asserts that pattern-based metaprograms can be tested using the related pattern information to improve the metaprogram testing. Our strategy to verify this hypothesis is to create a systematic approach to use the pattern information related to the metaprograms. A coverage criteria family is proposed to provide a systematic approach to the patterns usage because it is a standard approach to conduct a testing strategy to programs in general. The pattern-based coverage criteria family is used to verify the cost and quality of the test set that satisfies these criteria. With his coverage family, we can have different levels of rigor for the metaprograms under test. So, using it, we can evaluate different levels of expected cost and quality related to the Hypothesis 1.

A test design process with the pattern information as main concern was proposed to systematize and support the presented approach. These two proposals are used to verify the hypothesis and the goal. The first step is to instantiate the test design process to verify its viability and the major threats to its use. In the end, if we show that the coverage family can contribute to the overall quality of the testing of metaprograms then the hypothesis is considered verified. However, we cannot discard the pattern-based test strategy if those proposals do not provide enough positive information about the use of these patterns for the test of metaprograms. In this case, the justification could be based on the chosen approach to use the patterns.

To help the investigation of the viability of a pattern-based test technique, we propose a set of research questions to evaluate the cost and quality of the proposed coverage criteria family. Furthermore, these questions also help to give us more information about the viability of the use of pattern information on the test of the metaprograms. The first research question is defined below.

Research Question 1: *Can the test design based on pattern information from pattern-based metaprograms contribute to the quality of the testing of these programs?*

This first research question is related to the investigation of the pattern information capacity of being used during a systematic test process and if there is a gain to the quality of the pattern-based metaprogram under test. The proposed strategy to answer this question is presenting an example of a systematic approach using the pattern information to the testing of these metaprograms.

Research Question 2: *Is the cost of the generation of a test set using the pattern information to the test of a pattern-based metaprogram acceptable?*

With the Research Question 2, we evaluate the cost related to the generation of a test set to be used during the test of a pattern-based metaprogram. This question is important because if the cost is too high, it could make the approach unfeasible. An acceptable cost can be understood as a subjective goal, but in this case, we are using the cost related mainly to the number of test cases generated using the coverage criteria family. The size of the test set, the number of test cases in the set, is mainly related to two cost aspects: the generation cost and the test set execution cost. The former is related to the time and space used by the generation algorithm. The latter is related to the execution time of the test set using the metaprogram under test. In our work, we evaluated only the generation cost and it is used as cost variable in the rest of the text, but we know that the execution cost would also be affected by the test set size. Each proposed coverage criteria family has to be satisfied by its related test set. Another important aspect of cost variable is its comparison with the cost related to the grammar-based testing techniques. With it, we can check if there was a cost reduction when the pattern-based test was used over the traditional grammar-based ones.

Research Question 3 is related to the quality of the test sets produced using the pattern-based coverage criteria. This quality variable measures the capacity of the generated test set on the improvement of the pattern-based metaprogram quality.

Research Question 3: *Is the quality of the generated test set using the pattern information to the test of a pattern-based metaprogram acceptable?*

The question is important because it can present results related to the quality of the proposed technique. This question can be difficult to measure because the quality is a subjective property. To mitigate this aspect, we will use the number of faults detected by the test cases as a quality indicator to measure this variable. Another effort in this direction is to use the results obtained from the grammar-based test technique applied to the same evaluation environment. This process provides the information to compare the proposed test strategy with the grammar-based one and give us a relative quality indicator.

An important characteristic of a coverage criteria family is the flexibility of rigor [Ammann and Offutt, 2008; Zhu et al., 1997]. With it, a different level of rigor is associated with each criterion and its test set can be used in an appropriate situation. The choice between the different levels can be based on cost restrictions, quality goals or both. The next research question defines if the proposed pattern-based coverage criteria family provides this kind of flexibility.

Research Question 4: *Does the pattern-based coverage criteria family allow a flexibility of rigor on the testing of pattern-based metaprograms?*

Different levels of rigor are associated with the proposed pattern-based coverage criteria family. Each of these levels has an associated complexity and quality. In our proposal, the former is based on the test set generation process, and the latter is based on the number of faults detected by the associated test set. With those levels, we can control the testing when applied to a pattern-based metaprogram. An important relation between two coverage criteria is subsumption. Subsumption is a relation that verifies the capacity of two criteria to be satisfied by a test set [Ammann and Offutt, 2008] [8]. The levels of rigor associated with our coverage criteria family can be ordered using the subsumption relation. So, our strategy to answer Research Question 4 is to show that there is a subsumption relation for the proposed coverage criteria family.

The research questions presented in this section are related to the technical viability of the proposed coverage criteria family and consequently, the test based on patterns. We believe that answering these questions we can verify the Hypothesis 1 and present more information about the test of metaprograms based on patterns.

1.3.2 Hypothesis 2

A second hypothesis is proposed to verify the use of a pattern-based testing in conjunction with the traditional grammar-based testing to increase the quality of the testing of pattern-based metaprograms. First, the hypothesis is presented and after its research question is also presented.

Hypothesis 2. *The combination of pattern-based and grammar-based testing contributes to increase the quality of the testing of pattern-based metaprograms.*

The Hypothesis 2 has only one research question, and it is presented below.

Research Question 5: *What is the gain on quality, if any, for the use of the test sets from pattern-based and grammar-based criteria together in relation to the use of each of them individually?*

The Hypothesis 2 is related to the use of both techniques to check if this combination improves the quality of testing for a pattern-based metaprogram. Our strategy to verify this hypothesis is the creation of two test sets, one satisfying a pattern-based criterion and another satisfying a grammar-based criterion, and use their union on the evaluation infrastructure. This resulted test set is used to verify its quality and compare its results with each of them individually. The quality variable is the number of faults detected by the resulting test set in the same way of the Hypothesis 1. The infrastructure is the same that was created on the verification process for the Hypothesis 1.

Using the results obtained from Research Question 3 as a reference, this question investigate the increase in quality that the combined use of the test sets produces on the testing of metaprograms. For this question, the same quality metric is used, and the test sets are obtained from the investigation of previous research questions. We used different criteria to generate the test sets that present results from different levels of rigor. With the information from this question, we can know better if there is a gain on the use of these two testing techniques together. Furthermore, we can check if the

[8]The precise definition of the subsumption can be found in Chapter 2.

quality gain of use it is relevant or marginal. The cost of use both techniques were not investigated in detail because it always is higher than the use one of them isolated.

1.4 Contributions

As it has been presented, metaprograms do not use a systematic approach for the verification process in general. Many projects use testing as verification strategy, but they do not use it with a specific approach. A natural option to introduce a systematic approach to the testing of metaprograms is the grammar-based testing, but this technique has a limitation related to its cost usage with metaprograms in general. So, a test technique based on patterns can be used as differential over the grammar-based techniques.

Our goal is to contribute to the test process of pattern-based metaprograms. The proposed strategy is to use the pattern information used in these metaprograms to provide a systematic way of testing them. In this direction, there are two main contributions of this thesis: the pattern-based test design process, presented in Chapter 4, and the pattern-based coverage criteria family which support the proposed test design process, presented in Chapter 5. These two contributions aim to improve the systematization and quality of the overall testing process for pattern-based metaprograms.

The first contribution is the pattern-based test design process. This process proposes the abstract steps necessary to use the pattern information associated with metaprograms. Once the pattern information is extracted, we can use one or more pattern-based coverage criteria to test the metaprogram. The process could be used only to extract the patterns and generate the test requirement set or going further and generate the test set that satisfies the requirement set.

The second contribution is the pattern-based coverage criteria family, and it is composed by four coverage criteria, where three of them are based on the equivalence partitioning criteria family. These criteria have a hierarchical classification, this classification shows the primary goal of the coverage criteria family that is to provide different levels of rigor of the requirements. Using these criteria is possible to choose a set of test related to a criterion with a specific level of rigor. In this way, the test of the pattern-based metaprograms can be constructed using a systematic strategy and different levels of rigor.

1.5 Thesis Overview

The presentation of this work is divided into four parts: the theoretical foundation, pattern-based test design process, the coverage criteria family and the evaluation process. The first part of the document is related to the theoretical foundation. This part starts with the foundation related to the software test process, it is presented in Chapter 2. This chapter presents a necessary background to define the test design process and the coverage family. To define the family of coverage criteria, the pattern matching technique was used, and it is presented in Chapter 3. The concept of pattern matching and the Rascal metaprogramming language are presented in the chapter.

The pattern-based test design process is presented in Chapter 4. The process components and considerations over its instantiation are presented. The pattern-based coverage criteria family that supports the test design process is presented in Chapter 5. Furthermore, the subsumption relation is presented in the same chapter.

The test design and coverage family evaluation process are presented at Chapter 6. In this chapter, we present a case study using the test design process to show the instantiation of the process and a comparison between the pattern-based coverage criteria and grammar-based is presented. A systematic review is presented in Chapter 7 and it presents the research works related to the thesis subject. Finally, the conclusion and considerations about the thesis hypotheses are made in Chapter 8. Furthermore, the possible extensions related to the presented work are shown.

— 2 —

Software Test

The test activity can be defined as an evaluation of a software by its execution with predefined input values, the observation of its execution and the results obtained [Ammann and Offutt, 2008]. However, the tests are not capable to prove the absence of errors [Dahl et al., 1972]. Consequently, the tests cannot be used as formal proof of the software correction [Beizer, 1990]. In fact, they collaborate to increase the level of reliability on the software when it plays its specified functions. Thus, the effectiveness of the software test is directly related to its capacity to detect errors with a minimal effort and time [Ammann and Offutt, 2008; Pressman, 2001].

Given a program P to be under test, some terms necessaries are defined using this program. The **input domain**, $D(P)$, is the set of all possible values that P can receive as input; an **input data** are elements from the input domain and represent values which P can receive as input for its execution. The input data term also can be named as **test case value** [Ammann and Offutt, 2008] or **test data** [Delamaro et al., 2007]. Input data can be denoted by $d \in D(P)$; **test case** is an ordered pair where the first element belongs to the input domain and the second is the result that will be produced when executing the test if and only if the program satisfies its expected behavior for that input data, this result is called **expected result** [Ammann and Offutt, 2008]; the **test cases** are defined as $CT = \{(d, s) \mid d \in D(P) \wedge s \in S(P(d))\}$, such as $S(P(d))$ represents the set of possible results for the execution of P with a specific input data d [Delamaro et al., 2007]. The set of test cases can be called as **test set**.

The execution of the software under test using a test case is called **test case execution**. During the test case execution, the information specified by the input data are used as input and the expected result is used in the verification of the produced output. The test case success or failure is decided using a **test oracle**. Thus, the test oracle is used to verify the relation between the output produced by the test case execution and the expected result specified in the test case. In general, the mechanism for the oracle implementation should be external to the software under test [Ammann and Offutt, 2008; Agarwal et al., 2010].

The previous definition of the main goal to test activity used the term "error" to represent any kind of problem in the software under test. However, this term presents variations on its meaning based on the source of the problem. Consequently, the term needs a precise definition for a better understanding of the test activity and the problems related to the software under test. The **fault** is the term used to characterize a wrong or absent step, process or definition in the software. A fault can propagate many changes in the internal execution states of the software. These unexpected changes are defined as **error**. In general, the errors present as consequence a visible change in the expected software response, these changes are defined as **failures** [Delamaro et al., 2007; Ammann and Offutt, 2008].

The test process should be executed during all the software development phases. For each phase exists a test task related that should be executed. This relation between the development process and the test activity is called the "V" model. On this model, the test activity is broken into 5 steps and each of them should be elaborated in conjunction with a related phase of the development process [Ammann and Offutt, 2008].

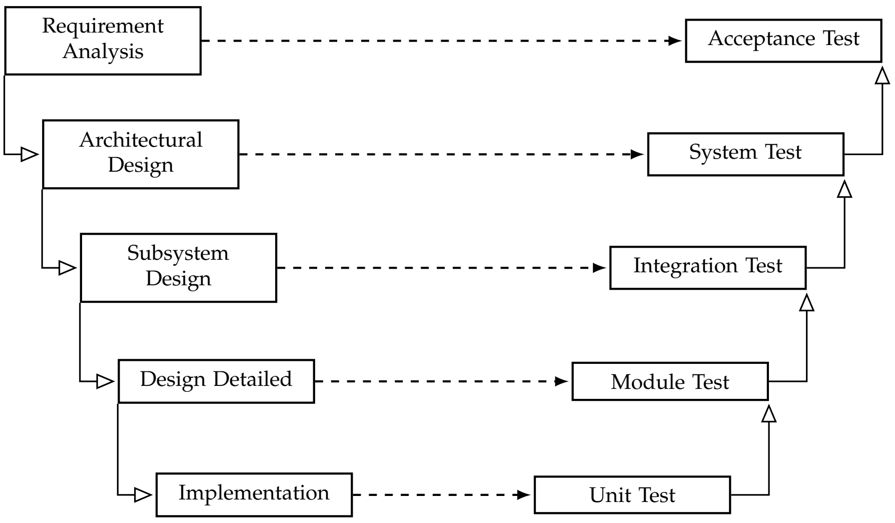

Figure 2.1: Software development activities and levels based on the model "V". Adapted from [Ammann and Offutt, 2008]

Figure 2.1 presents all model levels. Each project development phase is defined in the figure by a rectangle (presented on the left side) and the respective test level that should be elaborated with it (following the dotted arrow on the right side). When the tests have been elaborated then they should be executed from bottom to top, as it is indicated by the right lateral arrows on the figure.

Independently of the levels that are used on the test activity, it is necessary an elaboration of a test project. The test project encompasses since the planning level until the evaluation of results obtained during the execution of test process. The **test plan** will be the guide to conduct the entire test process. Its purpose is to describe the scope, approach, resources, and schedule of the testing activities. Another important activity is the **test design**, it is responsible to identify a set of features to be tested and to describe a group of test cases that will adequately test those features[Copeland, 2004]. Based on Sommerville [2003], the test process can be divided on the following steps:

– Project the test cases;

– Prepare the input data;

– Execute the test cases;

– Compare the execution results and expected results.

The test case project step is the moment to elaborate and organize the test cases. The next step is related to the elaboration of the input data using an existing test technique, like black box and white box testing. The **black box testing** techniques are those that use a representation or a specification of the software under test to elaborate the test cases. The **white box testing** techniques are those that use the implementation of the software under test as source of information to derive the test cases. In general, this classification is used to categorize the software testing techniques[Beizer, 1990; Myers et al., 2011; Copeland, 2004; Sommerville, 2003; Pressman, 2001], but there are different classification systems proposed currently, for example the one presented by Ammann and Offutt [2008].

The purpose of the test case specification is to specify in detail each test case listed in the test design specification[Copeland, 2004]. But this detail level, sometimes, is not detailed enough to execute the test case using the software under test. Consequently, some authors introduce two abstractions levels related to the level of detail of test cases: abstract test case and concrete test case. The **abstract test case** has the main information about the situation that it should cover but it is not executable. The **concrete test case** is a refinement of the abstract ones with the additional information to execute it using the software under test [Utting and Legeard, 2007; Hierons et al., 2009].

With the test restrictions related to an exhaustive evaluation, it is very important the elaboration of a test case set that detect a high quantity of faults in the software under test. Furthermore, this test set should be small to reduce the quantity of needed resources during the test execution be limited. Therefore, the careful elaboration of the test cases for the project is essential. Given the importance of the generation, selection and effective evaluation of the test cases, many coverage criteria have been elaborated to provide a systematic and rigorous way to select the test cases aiming to the faults discovering. Furthermore, these criteria have to carry in consideration the time and cost restrictions associated to the test project [Pfleeger, 2001].

A fundamental point of the test project is the choice of test cases which are used to the test process. This choice is known as **test adequacy criterion**. This concept was defined initially by Goodenough and Gerhart [1975]. Based on the authors, a test adequacy criterion is a logic predicate that defines which properties of a program should be exercised for it to be considered as tested, therefore it affirms that the well-done evaluation of these properties guarantees the absence of faults on the software under test. This conclusion is grounded on the satisfaction of the reliability and validity requirements. However, after the establishment of these criteria was recognized that there is no test adequacy criterion that has these two previous requirements and it is computationally feasible at same time [Howden, 1975]. Consequently, the focus of the research about adequacy criteria was changed from a search for theoretical criteria to the practical criteria that could be applied using approximations [Zhu et al., 1997].

The test adequacy criterion can be lead by two approaches: **test case selection criterion** and **test data adequacy criterion**. The first has a goal of defining the rules to select the test cases which satisfy the criterion. Through this approach, the test cases' definition can be accomplished in a constructive way. Thus, it is possible to construct algorithms capable to produce test cases that satisfy the selected criterion. The second approach, the test data adequacy criterion, acts as an evaluation to the test set in order to determine how much it satisfies the criterion [Zhu et al., 1997]. Through this last approach, the test set is built by an external process and the criterion is used to evaluate the level of satisfaction of that set in relation to the criterion.

There is a newer nomenclature to the terms test adequacy criterion, test case selection criterion, and test data adequacy criterion. Currently, this nomenclature is more used in the software testing field. The test adequacy criteria term is called **coverage criteria**, the test case selection criteria is called **generators**, and the test data adequacy criteria as **recognizers** [Ammann and Offutt, 2008; Budd, 1981]. Bearing in mind the simplicity of this nomenclature, these terms going to be used instead of the originals.

Definition 2.1 (*Coverage*)

Give a test requirement set TR to a given coverage criterion C, the test set T satisfies C if and only if for all test requirement tr in TR there is a test case t in T such as t satisfies tr [Ammann and Offutt, 2008]. The satisfaction relation between the criterion and the test set is denoted by $C \odot T$.

This work uses the coverage criterion definition presented by Ammann and Offutt [2008]. In this definition the authors used the concept of **test requirement** to define a specific characteristic of the software artifact which the test case should satisfy or cover. In this way, a coverage criterion is a rule or a set of rules that imposes the test requirement satisfaction to the test set [Ammann and Offutt, 2008].

There are some situations that a coverage criterion cannot be satisfied completely. In this way, it is necessary the definition of a gradual evaluation that satisfies the test cases in relation to the coverage criterion. This kind of evaluation can be defined by the concept of coverage level.

Definition 2.2 (*Coverage Level*)

Given a test requirement set TR and a test set T, the coverage level is the proportion between the number of satisfied test requirements and the size of TR [Ammann and Offutt, 2008].

Two distinct coverage criterion can be compared in relation to their capacity to be satisfied by a test set. Therefore, when a coverage criterion subsumes another criterion, the test set that satisfies the first also satisfies the second ones. Thus, it defines an order over the criteria similar the inclusion on the set theory [Weiss, 1989]. This concept is precisely defined below:

Definition 2.3 (*Coverage Subsumption*)

A coverage criterion C_1 **subsumes** the coverage C_2 if and only if all test set that satisfy C_1 also satisfy C_2 [Ammann and Offutt, 2008]. In this text, the symbol $\subseteq$ will be used to represent this relation between the coverage criteria. We can formalize the relation as:

$$C_2 \subseteq C_1 \equiv \forall T \bullet C_1 \odot T \Rightarrow C_2 \odot T.$$

2.1 Equivalence Partition

The **equivalence partition** technique assumes that it is possible to classify the input data of the software under test and uses the classes to reduce the number of test cases used during the testing. The premise of the technique is that one class represents all possible input data of the software. Consequently, the capacity of detecting a fault is the same among all elements of the same class. The set of classes should represent all the input space for a software under test and each of these classes must be mutually exclusive. Each of these categories is called **equivalence class** [Myers et al., 2011].

The equivalence classes also are called **blocks**. These blocks will partition the software input domain. Furthermore, the union of all blocks identified for a program should cover the software input domain. For each pair of blocks, there is no common element between them. Thus, all block's pairs are disjoint [Ammann and Offutt, 2008].

There is another important point beyond the partition identification and blocks choosing: the combination of pairs from the blocks. This combination is defined by a **criterion combination** and it allows to test a combination between elements from different blocks [Ammann and Offutt, 2008].

Using combination criteria is possible to define a set of coverage criteria associated to each combination strategy. These coverage criteria can be used to generate a test set and it could be used to test the software. During this project we used the following coverage criteria based on equivalence partition technique: *all combinations*, *each choice* and *pair-wise*. The first of them is the ideal to cover all possible combinations of the software input, but in practical situations the cost to test all combinations is too high. For example, if we use a four block partition for a software with three input parameters, the all combinations requires $4 \times 4 \times 4 = 64$ test requirements.

Criterion 2.1 (*All Combinations*)

All combinations of blocks from all characteristics must be used by the test set [Ammann and Offutt, 2008].

This criterion defines the total combination between the blocks. For example, if we have three partitions with blocks $[A, B]$, $[1, 2, 3]$ and $[x, y]$, then the criterion will need the following twelve tests:

$$\begin{array}{cc} (A,1,x) & (B,1,x) \\ (A,1,y) & (B,1,y) \\ (A,2,x) & (B,2,x) \\ (A,2,y) & (B,2,y) \\ (A,3,x) & (B,3,x) \\ (A,3,y) & (B,3,y) \end{array}$$

The number of test requirements for this criterion will be the product of the number of blocks for each partition. The size of the test requirement set, $|TR|$, for this criterion can be defined as:

$$|TR| = \prod_{i=1}^{n} |B_{p_i}|. \tag{2.1}$$

Where $|B_{p_i}|$ is the size of block for the partition p_i and n the number of partitions [Ammann and Offutt, 2008]. For the all combination criterion, in particular, the number of test requirements and the number of test input data are the same. Because the elevated number of requirements generated by this criterion, it is interesting to define a combination criterion that combines the blocks generating fewer requirements and without losing some interesting requirements cases. The following coverage criterion requires that we combine each block at least once.

Criterion 2.2 (*Each Choice*)

An element from each block of each characteristic must be used in at least one test case [Ammann and Offutt, 2008].

Each choice criterion guarantees that will be a test requirement representing each on of the blocks defined by the there partitioning. The number of requirements for this criterion is defined as:

$$|TR| = \sum_{i=1}^{n} |B_{p_i}|. \tag{2.2}$$

The minimal number of test cases to satisfy the each choice criterion requirements will be $|TS| = \max\limits_{i=1}^{n} |B_{p_i}|$, where "$\max$" represents the maximum number of blocks for all partitions [Ammann and Offutt, 2008]. Using the same blocks as before, $[A, B]$, $[1, 2, 3]$, and $[x, y]$, one example of test data that can satisfies the criterion is $(A, 1, x)$, $(B, 2, y)$, and $(A, 3, x)$. For this example, the $|TS| = 3$ and $|TR| = 7$.

The blocks $[A, B]$ and $[1, 2, 3]$		
$(A, 1)$	$(A, 2)$	$(A, 3)$
$(B, 1)$	$(B, 2)$	$(B, 3)$
The blocks $[1, 2, 3]$ and $[x, y]$		
$(1, x)$	$(1, y)$	$(2, x)$
$(2, y)$	$(3, x)$	$(3, y)$
The blocks $[A, B]$ and $[x, y]$		
(A, x)	(A, y)	(B, x)
(B, y)		

Table 2.1: The test requirement set for the pair-wise coverage for the example case.

A disadvantage of each choice criterion is not combine elements between the blocks, so the next criterion requires explicit combinations of each block values. The *pair-wise* criterion defines requirements to combine all the blocks two by two. This combination strategy has as a goal to detect *double mode faults* (if there is a consistent problem when specific levels of two software's parameters occur together) [Pressman, 2001]. There are some empirical results show that pair-wise testing is practical and effective for various types of software systems [Cohen et al., 1996, 1997, 1994].

Criterion 2.3 (*Pair-Wise*)

An element from each block from each characteristic must be combined with another element from every block for each other characteristic [Ammann and Offutt, 2008].

The number of test cases for it will be at least $|TS| = (\max_{i=1}^{n} |B_{p_i}|) \times (\max_{j=1, j \neq i}^{n} |B_{p_j}|)$, where "$\max$" represents the maximum number of blocks for all partitions and $j \neq i$ represents the second "$\max$" size from all blocks [Ammann and Offutt, 2008]. The number of requirements to cover the criterion is defined by

$$|TR| = \sum_{i=1}^{n-1} (|B_{p_i}| \times |B_{p_{i+1}}|) + \ldots + (|B_{p_i}| \times |B_{p_n}|). \tag{2.3}$$

Using the same example of three partitions with the blocks $[A, B]$, $[1, 2, 3]$, and $[x, y]$, then the criterion will need sixteen test requirements to cover the combinations and these are show in Table 2.1. With the size of requirement set as $|TR| = 16$ and $|TS| = 6$ for the minimal possible test set size. The pair-wise coverage criterion allows the same test case to satisfy more than one unique test requirements. So the requirements presented on the Table 2.1 can be combined and satisfied in several ways, including:

$$\begin{array}{cc} (A, 1, x) & (B, 1, x) \\ (A, 2, y) & (B, 2, y) \\ (A, 3, x) & (B, 3, x) \\ (A, -, y) & (B, -, y) \end{array}$$

The test cases with the "$-$" symbol mean that any block can be used to replace it. There are several algorithms published to satisfy the pair-wise criterion [Czerwonka, 2016]. Nevertheless, the problem of generating a minimum pair-wise test set is NP-complete [Lei and Tai, 1998].

2.2 Grammar-Based Testing

As presented by Ammann and Offutt [2008], the most important characteristic to define a coverage criterion is the abstraction it uses. The focus of this work is to test pattern-based metaprograms and they have many related characteristics with compilers. The natural choice to test compilers is **grammar-based testing**. This testing technique uses a context-free grammar to describe the software input syntax and from it create the test cases. The test data generation build using this technique is not new. The first effort in this direction was made to improve the quality of testing process of compilers. The pioneer on the use of this technique was Sauder [1962], he presented a test data generator to improve the testing process of a COBOL compiler implemented by Air Force Logistics Command.

The grammar-based coverage criteria are defined using context-free grammars. Therefore, we need a precise definition of a context-free grammar.

Definition 2.4 (*Context-Free Grammar*)

A context-free grammar G is a 4-tuple (N, T, P, S) where:

- N is a finite set of *nonterminals*;
- T is a finite set of *terminals*;
- P is a finite subset of $N \times (T \cup N)^*$ called *production rules*;
- $S \in N$ is the *initial symbol*.

An **alphabet**, denoted by Σ, is a finite set of symbols which will be used to compose the *strings*. A **string** is a finite sequence of symbol from the alphabet. The symbol λ represents an empty string. The context-free grammar can be used as a string parser or generation mechanism. A set of possible operations associated to strings is presented on the Table 2.2. The sets Σ^+ and Σ^* represent, respectively, the set of all strings formed by one or more symbols and the set of strings formed by zero or more symbols. A grammar G defines a **formal language** $L(G)$ that represents the set of strings generated by the grammar G. A string that belongs to the formal language is denoted **sentence** [Rozenberg and Salomaa, 1997].

Operation	Notation	Description	Example
Length	$\|x\|$	Number of symbols of a string	$\|abc\| = 3, \|\lambda\| = 0$
Concatenation	xy	If x and y are strings then xy is also a string	Let $x = aab$ and $y = caa$, $xy = aabcaa$
Substring	Q is a substring of R	There are strings P_1 and P_2 such that $R = P_1QP_2$	a is a substring of $abbcad$
Power	$x^n, n \in \mathbb{N}$	Repeat the string x by n times	$a^2 = aa, ab^3 = ababab$
Index	$x_i, i \in \mathbb{N}$	Return the ith symbol in the string x	Let $x = abc$, then $x_0 = \lambda, x_1 = a, x_2 = b$
Direct Derivation	$\alpha' \Rightarrow \beta'$	α' is directly derived to β' iff there are $\alpha_1, \alpha_2, \alpha, \beta \in (N \cup T)^*$, such that $\alpha' = \alpha_1\alpha\alpha_2, \beta' = \alpha_1\beta\alpha_2$ and $\alpha \rightarrow \beta$ belongs to P	$aAa \Rightarrow aba$, when $A \rightarrow b \in P$
Derivation	$R \Rightarrow^* Q$	R derives Q iff $R_0 \Rightarrow R_1 \Rightarrow \ldots R_k$ and $R_0 = R, R_k = Q$	$aAa \Rightarrow^* aba$, if there is a sequence of directed derivations $aAa \Rightarrow aBa \Rightarrow aba$

Table 2.2: Context-free grammar related operations.

With the definition of context-free grammars we can define the coverage criteria based on its elements. The coverage criterion presented by the Definition 2.4 is called *terminal symbol coverage* and it defines that all terminals symbols in the grammar must be cover by the test set.

Criterion 2.4 (*Terminal Symbol Coverage*)

A test requirement set TR must have one test requirement tr for each terminal symbol t in the grammar G [Ammann and Offutt, 2008].

The second criterion is the *production coverage* and it defines that all productions rules in the grammar must be used by the test set.

Criterion 2.5 (*Production Coverage*)

A test requirement set TR must have one test requirement tr for each production rule p of the grammar G [Ammann and Offutt, 2008].

If a test set satisfies the production coverage criterion then the test set also satisfies the terminal coverage criteria. This happens because if all production rules are used in the test set then all terminals are consequently used also. Consequently, there is a subsumption relation between these two criteria. A criterion based on the grammar derivations is presented on the Definition 2.6. A test set that satisfies it must have sentences that use all possible grammar derivations [Ammann and Offutt, 2008].

Criterion 2.6 (*Derivation Coverage*)

A test requirement set TR must have one test requirement tr for each derivable string from the grammar G [Ammann and Offutt, 2008].

While the terminal and production coverage criteria need a relative small test set, the derivation coverage criterion is impossible to be satisfied in the majority of cases. Thus, variations of these criteria are used [Lämmel and Schulte, 2006; Hoffman et al., 2011; Moreira et al., 2013] aiming to restrict the size of the test set through the application of these restrictions over the derivations.

2.3 Mutation Testing

Mutation testing was created on the 70s at *Yale University* and *Georgia Institute of Technology*. It has a strong relation with the classic method for logic error detection in digital circuits, it is based on the test model of unique fail. The technique principle is to use a set of programs with small changes called *mutants*. These programs are obtained from the original program under test and those mutants are used to evaluate how much the test cases are suitable to the test of the program under [DeMillo et al., 1978; Ammann and Offutt, 2008].

There are two fundamental hypotheses that sustain the mutation test proposed by DeMillo et al. [1978]: the *competent programmer hypothesis* and the *coupling effect hypothesis*. The former assume that expert programmers write correct programs or programs real close to correct ones. Thus, the faults are added in the programs by small syntactic deviations that change its semantic but without compromise the compilation. The latter hypothesis assumes that complex faults are related to the simply ones. Thus, it is expected that the test set which will be capable of reveal the presence of those original faults also is capable of reveal the more complex ones. Using those two hypothesis to improve the test set used on the program under test, we identify common syntactic deviations and these are inserted on the program under test using single and small transformations on its code. With this, the original test set should detect these small transformations and improve its quality at the end of the process.

According to DeMillo et al. [1978], the mutation method should be used interactively following the steps:

1. A program under test P and a set tests T are provided to the process;
2. First, the mutation process executes the original program P using the test set T to verify P in relation to T;
3. If P does not pass on the tests defined in T then the process is stopped and P contains a fault that must be corrected;
4. If P does pass on the tests using T then new programs $P_1, P_2, \ldots, P_k$ are derived from P. These new programs are called **program mutations**;
5. If the execution result of a mutation P_i using a test case from T will be different from the result obtained from P then the mutant is said to be *killed* by the test set T. Thus, meaning that the fail injected in P by the mutation process was detected by T;
6. If the execution result of the mutation P_i be the same of P then the mutation is said to be *alive*. This situation could be caused by two possibilities: P is equivalent to the P_i or the test set T was not capable to detect the fault inserted.

At the end of the process, if all mutations are dead or equivalent then the test set is adequate to the test of the program. Furthermore, if the test set cannot detect a mutation, the test set should be improved and the mutation process should be executed again with the improved test set. The **mutation score** is a proportion between the total of mutations created and the non-equivalent ones [Ammann and Offutt, 2008]. The mutation technique provides an objective measure of the confidence level to the adequacy of the analyzed test set [DeMillo et al., 1988].

2.3.1 Mutation Operators

The mutation technique efficacy is directly related to the faults injected to the program during the process. Each kind of fault injected into the program is called **mutation operators**. Consequently, the mutation operators used during the mutation process is crucial. The set of operators to be used during the process can be guided by the following important characteristics: (1) the fault coverage, if we have a test set T that is adequate to a program P in relation to generated mutations then it should reveal the many as possible number of errors in P; and (2) the application cost, it should be affordable since the cost of operator application and the number of mutations produced could be high [DeMillo et al., 1988].

In general, the mutation operators are constructed aiming at two goals: to introduce simple syntactic changes with base on typical errors made by programmers, and to force some test objectives, for example execute a decision point on the execution flux of a program. The application of a mutation operator can generate more than one mutant. It happens because if a program contains many elements that are in the domain of some mutation operator then this operator should be applied to each one of them [Offutt et al., 1996].

The mutation operator can be classified with base on the faults that they represent. This classification can help on the operator choice process to use on the application of the mutation technique. The classification can be used to select and to archive a representation level to a mutation operator set, it also can help to reduce the operator number of that set. Thereby, the application cost for the technique can be reduced. The classification presented on the Table 2.3 is based on mutation operators for imperative programming languages and it was proposed by Andrews et al. [2005].

Class	Description
1	Replace an integer constant by 0,1,-1, C+1, C-1
2	Replace an arithmetical, relational, logic, bitwise logic, increment decrement or arithmetical attribution operator by other operator of the same class
3	Negate the decision expression into conditional structures:`if` e `while`
4	Remove instructions

Table 2.3: A classification to the mutation operators based on imperative languages.

Following that classification, we select the operators listed on the Table 2.4 to the use on this work. This selection was made with the objective of choose some operators of each one of the classes. Furthermore, the selection considers the operators that will be more impact over the fault detection based on the work presented by Offutt et al. [1996].

Identifier	**Name**	**Class**	**Description**
ABS	Absolute value insertion	2	Each arithmetic expression is modified using the absolute values:negative, positive and zero
AOR	Arithmetic operator replacement	2	Each occurrence of one arithmetical operators is changed to each of the other operators of the same category
ROR	Relational operator replacement	2	Each occurrence of one relational operator is changed to each of the other operators and the absolutes values true and false
COR	Conditional operator replacement	2	Each occurrence of each logical operator is changed each of the other operators of the same category and the absolutes values true and false, and left operator or the right one
ASR	Assignment operator replacement	2	Each occurrence of one of the assignment operators is changed to each of the other operators of the same category
UOI	Unary operator insertion	2 e 3	Each occurrence of the unary operator is inserted before each compatible expression of the same type
SVR	Scalar variable replacement	1	Each variable reference is changed by other variable of the compatible type
BSR	Bomb statement replacement	4	Each statement is replaced by a special bomb function

Table 2.4: Mutation operators based on imperative languages.

— 3 —

Pattern Matching Theory

Pattern matching is an interesting feature to be applied during the development of metaprograms. The pattern matching provides a simple mechanism that can be used to describe specific patterns over the metaprogram's input language. This feature can simplify the development of the metaprograms. An example is a metaprogram that needs to treat specific patterns in the metaprogram's input program, for example, the treatment of a specific keyword in this program. This kind of simplification was used in the metaprogram's implementation for the Cyclomatic Complexity(CC) [McCabe, 1976] for Java programs [Landman et al., 2014] (This metaprogram implementation example is presented in Listing 3.8).

From the formal stand point, in general, **pattern matching** is defined as given two terms p and s, determine if there is a substitution σ such as $\sigma(p) = s$. The term p is called **pattern** and s is the **subject** [Baader and Nipkow, 1998]. Additionally, it is necessary to define which terms are possible to be attributed to p and s. The pattern term p will be defined using a **pattern language** and the subject term s using a **object language**. The object language represents the set of terms that a pattern can be applied to. The pattern language is composed by the terms from the object language and others to provides pattern features, like variables and operators.

Many programming languages and libraries offer the pattern matching resource that can be used during the development of regular programs, for example, Haskell [Hudak et al., 1992b], Scala [Odersky et al., 2004] and TOM [Balland et al., 2007]. Metaprograms specialized environments also offer the pattern matching resource as an option on the developing of the metaprograms, some examples are ASF+SDF [van den Brand et al., 2001], StrategoXT [Bravenboer et al., 2008] and Rascal[Klint et al., 2009, 2011]. During the text we call a programming language that has the pattern matching resource as **pattern based language**.

For the metaprograms context, the object language is, in general, the language of the program used as data by a metaprogram. The pattern language in this context is defined aiming at the set of features that will be supported by the metaprogramming language. This thesis was developed using the Rascal language as an implementation language. The Rascal language features and the pattern matching resources used during the thesis are presented in Section 3.1, The complete Rascal's pattern language syntax is presented in Listing A.3, Section A.3 of the appendix.

During the initial research process that was conducted to identify the options to formalize the pattern matching mechanism, we identified two approaches for it: a traditional formalization for the pattern matching for strings [Knuth et al., 1977] and a formalization using the lambda calculus [Jay, 2009]. The former is simpler because its definition is based on tree terms and it defines only variables as pattern feature in the pattern language. To simplify the coverage criteria definitions that depends on patterns and to archive the thesis goals the traditional formalization for strings is used during this work. Nevertheless, we can evaluate the lambda calculus formalization on a future work because this formalization provides a broader and more generic formalization to support, in theory, any kind of implementation. More information about this future work in Chapter 8.

The pattern matching formalization presented by Knuth et al. [1977] is based on strings. During this work, we used the tree based formalization presented by Hoffmann and O'Donnell [1982] as the reference to simplify the formalization process and its uses during the coverage criteria definitions. This reference replacement does not affect the resultant formalization because the tree objects used during the formalization always can be translated to the string objects. Because of this characteristic, the term "pattern" is used indistinctly for "patterns over trees" and "patterns over sentences".

The formalization presented here was defined by Hoffmann and O'Donnell [1982] and it was made to propose a pattern matching algorithm over trees. Thus, we start by defining a finite alphabet of symbols called Σ. This alphabet has the symbol λ to represent the empty term and constant symbols to represent functions without arguments. The symbols a and b are examples of constant symbols and λ is the empty term. The set S denotes the Σ-terms and it is defined below.

Definition 3.1 (*Σ-terms*)

1. For all $b \in \Sigma$ with arity 0, b is a Σ-term;
2. If a is a symbol of arity n in Σ, then $a(t_1, \ldots, t_n)$ is also a Σ-term when each t_i is also a Σ-term;
3. Nothing else is a Σ-term [Hoffmann and O'Donnell, 1982].

To illustrate the Definition 3.1, we present two examples of Σ-term: $a(a(b, b), b)$ and $a(b, a(b, b))$. In both of them, the symbol a has arity 2 and b has arity 0. The Σ-terms can be seen as labeled and ordered trees. Consequently, those two terms presented are considered different. **Variables** are special symbols with arity 0, denoted by v, and they are not in Σ. Variables can be replaced by any Σ-term. The set $\Sigma \cup \{v\}$-terms is denoted by S_v [Hoffmann and O'Donnell, 1982].

Definition 3.2 (*Pattern over Trees*)

A **pattern over tree** is any term in S_v.

Definition 3.2 defines the patterns possible to be used with the pattern matching relation. Using this definition the number of distinct variables in a term will be always one. In the situations where more than one variable is necessary, the S_v can be redefined as $\Sigma \cup V$ where V is a finite set of distinct variables. The next step is to define the operation of pattern matching.

Definition 3.3 (*Pattern Matching*)

A pattern $p \in S_v$ with k occurrences of the symbol v **matches** a subject tree $t \in S$ at the node n if there exist Σ-trees $t_1, \dots, t_k \in S$ (not necessarily the same) such that the Σ-tree p', obtained from p by substituting t_i for the ith occurrence of v in p, is equal to the subtree of t rooted at n [Hoffmann and O'Donnell, 1982].

In this work, the pattern matching process is denoted by the operator $\succ$, therefore if the pattern p and a subject s are given then the match between them is specified as $p \succ s$. The symbol $\not\succ$ represents the operation of not match of a pattern. Any sentence in the object language can be parsed to a derivation tree using its context-free grammar[1]. Thus, we will treat the pattern matching over trees and strings as equivalent in the text.

3.1 The Rascal Language

Rascal [Klint et al., 2009, 2011][2] is a development environment which offers many features to develop metaprograms in an integrated way. The tool offers an integrated environment to develop, execute, test and debug metaprograms. Rascal implementation is based on the Eclipse environment and it is coded in the Java Language and in Rascal itself. Rascal can also be used to develop domain specific languages and it offers many development features that help in the development of the these languages.

[1]The cases in which the grammar is ambiguous can have more than one tree per string [Rozenberg and Salomaa, 1997]. However, even on those cases the string matches with the defined pattern, since the cases with ambiguity can only produce more trees and not less for each pattern.

[2]The language semantic and syntax presented on this section is mainly based on Klint et al. [2010].

Listing 3.1: A simple Rascal metaprogram.

```
 1 module exRascal01
 2
 3 import IO;
 4
 5 //the main function on the Rascal programs
 6 void main() {
 7   int a = 5;
 8   f1(a);
 9 }
10
11 void f1(int p) = println("Hello world. Value of p argument <p>");
```

Program output

```
Hello world. Value of p argument 5
```

The Rascal's syntax is based on Java. Listing 3.1 presents a simple Rascal program that illustrates the basic syntax and semantics of the language. The first line defines the module name and the third line imports the module called "I/O" that has functions related to input/output processes. Comment lines are defined using the symbol `//` and an example is presented on line 5. The function named `main` is declared on line 6. There are two main forms to define functions in the language, the function `main` presents an example of the traditional form to declare the functions in Rascal. The second form is presented on line 11, this form can be used to define functions using more than one function declaration but with the same name. The second form can also be used to simplify the function declaration.

The `f1` function definition example has as your body the code defined on the right side of the `=` symbol. The `println` is a basic function imported from "I/O" module and it prints its argument on the console. The attributions, conditionals, and repetitions commands have a similar syntax as in Java. Rascal has support to string interpolation, with this feature it is possible to parse strings values on the execution time to compose new string values. The replacement is defined by the special symbols `<` and `>` and the value of replacement is defined by an expression enclosed by these symbols. An example of the use of this feature is presented on line 11.

On the example of Listing 3.1, the execution of the program is started by the function `main`. The function `f1` will be called using as argument the value from variable `a`. The value of `p` argument is evaluated and interleaved into the result string and the result of this process is presented on the console output by the function `println`. This console output is presented by the listing's part titled as "Program output".

Listing 3.2: Metaprogram using different types of collections in Rascal.

```
1 module exRascal02
2 import IO;
3
4 void main() {
5   set[bool] sb        = {true,false};
6   list[bool] lb       = [true,false,true];
7   map[bool, str] m1   = (true:"True", false:"False");
8   println("The set sb: <sb>");
9   println("The list lb: <lb> and the second element: <lb[1]>");
10   println("The map m1: <m1>");
11
12   rel[int, str] r1    = {<1,m1[true]>, <2,m1[false]>};
13   println("The relation r1: <r1> and the first element: <r1[1]>");
14 }
```

Program output

```
The set sb: {true,false}
The list lb: [true,false,true] and the second element: false
The map m1: (true:"True",false:"False")
The relation r1: {<1,"True">,<2,"False">} and the first element: {"True"}
```

Another important feature of the language is the support to collection data types and special expressions to deal with them. There are four main collection types in Rascal: Set, List, Map, and Relation. The Listing 3.2 presents an example of these collections in use. Lines 5 to 7 and 12 show the examples of instances constructions for each of collection type. Basically, each variable instance is declared by providing the collection type information, value type and name of the variable. For example, on line 5 the set `sb` is instantiated using the special symbols `{` and `}`, and defining one element as the **`true`** Boolean value and another as the **`false`** value. A similar form is used to define the list and map collections on lines 6 and 7. The list collection uses the special symbols `[` and `]` to instantiate, and the map uses `(` and `)`. Lines 8 to 10 show the command related to the presentation of the instances create by the last commands. The output of these commands can be seen on the program output. The expression `lb[1]` on line 9 is an image projection operation to access an element from the list collection and it is a zero-based index access. On the example, the result will be the Boolean value **`false`**. This access operation is also available to the map collection but using the key value provided by the instantiation, an example of this kind of use is presented on line 12.

On line 12, there is an example of relation collection variable declaration. On this example, two elements are declared into the collection: `<1,m1[`**`true`**`]>` and `<2,m1[`**`false`**

]>. The first element <1,m1[true]> declares a pair where its first element is the integer 1 and the second is a string. This string value is accessed from the m1 map using the index true and the result for this access, in this case, is the “True” string. The second element, defined by <2,m1[false]>, declares the pair where its first element is 2 and the second is “False”, using the same access process from the m1 map. Line 13 uses the string interpolation to print the values stored in r1 and the value of relation’s element accessed by the index 1.

Listing 3.3: A comprehension feature example.

```
1 module exRascal03
2 import IO;
3
4 void main() {
5   set[int] si        = { e | e <- [1..10] };
6   list[int] li       = ( [] | it + [e] | e <- [1..10] );
7   println("The set si: <si>");
8   println("The list li: <li>");
9 }
```

Program output

```
The set si: {7,1,3,9,2,4,6,5,8}
The list li: [1,2,3,4,5,6,7,8,9]
```

Listing 3.3 presents an example of the comprehension feature provided by the language. This feature is a notation inspired by mathematical set-builder notation and list comprehensions that help to write succinct definitions of collections. In the example, the expression presents a comprehension used to define a set of integers between 1 to 9. The expression [1..10], in lines 5 and 6, creates a generator that provides values between 1 and 9 to the e variable, each of them will be attributed to the e variable during the instantiation. Another feature provided by the language is the reducer, it resembles the fold function found in most functional languages. On the right hand side of the attribution on line 6, we can see the reducer construction used to produce a list of integer between 1 to 9. The symbol it contains the current value during the iteration and at the end the value of it is the result of the reducer. At the end the same values are produced for both variables and the print functions present their instances values.

The Rascal language provides support to Algebraic Data Types(ADT) [Pierce, 2005]. This feature is presented on the Listing 3.4 with a simple tree definition example. In the example, the data type ADTree is defined and it has two constructors: leaf and nod. On line 10 we can see the construction of a data type instance. The lines between 12 and 14 present some ways to access the instance information on the variable. First, the label `lNode` is used to access the left node on the tree. Secondly, the operator **`has`** is used to verify if the node on the label `rNode` has the label `N`, this label is defined on Line 5 of the `leaf` construction. Finally, the operator **`is`** verifies if the value under the label `lNode` is a `nod` type. All these expressions are written on a string using the language interpolation feature.

Listing 3.4: Defining an algebraic data type in Rascal.

```
 1 module exRascal04
 2
 3 import IO;
 4
 5 data ADTree = leaf(int N)
 6             | nod(ADTree lNode, ADTree rNode)
 7             ;
 8
 9 void main() {
10   ADTree t = nod(nod(leaf(0), leaf(1)), leaf(2));
11
12   println("Left node: <t.lNode>");
13   println("Does t.rNode has a N label? <t.rNode has N>");
14   println("Is t.lNode a node? <t.lNode is nod>");
15 }
```

Program output

```
Left node: nod(leaf(0),leaf(1))
Does t.rNode has a N label? true
Is t.lNode a node? true
```

3.1.1 Patterns in Rascal

One of the most important resource in the Rascal language is pattern matching. A pattern matching in the language is represented by `Pattern := Subject` and the evaluation of this pattern expression always returns a Boolean value. Furthermore, the pattern variables are bound with the value from the subject when the matching evaluation returns true. These variables will be available in the scope in which the pattern expression was executed. In this way, pattern matching expressions can be used within any construction that accepts a Boolean value.

A simple example can be see on Listing 3.5. A function without return is defined and it contains only one conditional structure. The expression `int x := v` represents a pattern matching where `x` is a variable on the pattern and `v` is the subject. For this case the matching will always be true because the only condition to the matching evaluation is the type of variable `v`, which in this case always is **`int`**. On the first instruction in the true branch of the conditional, the variable x is created and its value will be the same value of v.

Listing 3.5: A basic example of the use of patterns in Rascal.

```
module exRascal11
import IO;

//The function definition
void func01(int v) {
  if (int x := v) {
    println("v is integer. <x>");
  } else {
    println("v is not integer.");
  }
}
```

Program output

```
> func01(5);
v is integer. 5
```

Table 3.1 presents some pattern matching instructions available on the Rascal language [3]. The first construction does the match based on the data type, as presented on the Listing 3.5. The lines 2, 3 and 4 present the pattern matching over the basic structures of tuples and lists. The last table line presents the not matching of a pattern. This operation only return true when there is no match between the pattern and subject. However, on the example case the variable N is not bound with any value regardless the result of the matching evaluation.

N.	Code Examples	Description
1	`int x := 3;`	Typed pattern, the variable x is bounded to the value 3.
2	`<int x, y> := <3, "x">;`	Pattern over tuples, the variable x and y are bound to the values 3 and x.
3	`[1, 2, x] := [1, 2, 3];`	Pattern over lists, the variable x is bound to the value 3.
4	`[1, xs*, 4] := [1, 2, 3, 4];`	Pattern slice-variable, the variable xs is bound to the list [2,3].
5	`[10, N*, 50] !:= [10, 20, 30, 40, 50];`	Anti-match operator, return false and it does not do the bind with the variable N.

Table 3.1: Some common pattern matching constructions in Rascal.

There are two pattern matching constructions that are very important and useful in Rascal: constructor pattern and deep matching. These patterns are specially important when it is necessary to deal with ADTs. The Listing 3.6 presents an example

[3]The complete pattern language syntax can be see on the Listing A.3 on the appendix.

using these patterns in use. This example uses the Listing 3.4 ADT definition `ADTree`. Line 7 shows a matching using the constructor pattern, the term `nod(l,r)` defines the constructor identifier `nod` and the variables `l` and `r`. If the match happens, the pattern matching mechanism returns true to the **`if`** instruction and the variables `l` and `r` are bound to the subterms `nod(leaf(0),leaf(1))` and `leaf(2)` respectively. These variable will be available in the scope of the **`if`** instruction.

Listing 3.6: Example of constructor and deep matching patterns.

```
1  module exRascal12
2  import IO;
3  import exRascal04;
4
5  void main() {
6    ADTree t = nod(nod(leaf(0), leaf(1)), leaf(2));
7    if (nod(l,r) := t) {
8      println("Variables values l:<l> and r:<r>");
9    }
10   for (/leaf(i) := t) {
11     println("A bind for i: <i>");
12   }
13   println(i);
14 }
```

Program output

```
Variables values l:nod(leaf(0),leaf(1)) and r:leaf(2)
A bind for i: 0
A bind for i: 1
A bind for i: 2
Error: Undeclared variable: i
```

The deep matching pattern is used on line 10. It shows a search for `leaf` nodes on the ADT subject. Thus, they will be bound to the variable `i` with the values "0", "1" and "2". It is important to note that there is an integration between the pattern matching mechanism and the **`for`** command, this integration allows go though all result values from the match. The same behavior is present in all repetition instructions in the language. On line 13 illustrate the control over the scope for the pattern matching variables. In this example an error is throw on this line because the variable `i` is out of scope. The output of example program can be seen on the listing.

A common technique used when is programming using Rascal is the use of the command **`visit`** to treat specific situation during the use of an ADT variable [Klint et al., 2011]. This command implements a standard way to deal with trees in the language, the approach is based on the visitor design pattern [Gamma et al., 1995]. Basically, the tree is processed by the instruction `visit` and there are a set of patterns declared to treat each desired case. The patterns used in these cases are defined using a similar style to those presented for the conditional structure on Listing 3.6 example.

The default algorithm used for traversing the tree structure is bottom-up, first the leafs and following the nodes in direction of the root. This algorithm can be changed by a predefined set of algorithms supported by the language. An example of the use of this instruction is shown on Listing 3.7. The lines 8 to 10 define the `visit` instruction and the unique case is defined on line 9. This code sums the integer value on the leafs and stores the sum of these values in the variable `c`. At the end, the sum is printed on the program output.

Listing 3.7: Example of use of the instruction `visit` in Rascal.

```
 1 module exRascal13
 2 import IO;
 3 import exRascal04;
 4
 5 void main() {
 6   ADTree t = nod(nod(leaf(0), leaf(1)), leaf(2));
 7   int c = 0;
 8   visit(t) {
 9     case leaf(int N): c = c + N;
10   };
11   println("The sum of nodes is <c>");
12 }
```

Program output

```
The sum of nodes is 3
```

To illustrate the use of the pattern mechanism in a metaprogram, we presented on Listing 3.8 an example of a metaprogram using the pattern matching to calculate the Cyclomatic Complexity(CC) [McCabe, 1976] for Java programs. This example is a typical implementation of the CC algorithm. This implementation of CC is coded in Rascal and was used by Landman [Landman et al., 2014] to calculate and analyze the correlation between the CC and source lines of code on a large corpus of Java code. This example is an extract from the complete metaprogram implementation and this extract represents the main core of the implementation.

Listing 3.8: Example of a pattern-based metaprogram written in Rascal [Landman et al., 2014].

```
1 int calcCC(Statement impl) {
2   int result = 1;
3   visit (impl) {
4     case \if(_,_) : result += 1;
5     case \if(_,_,_) : result += 1;
6     case \case(_) : result += 1;
7     case \do(_,_) : result += 1;
8     case \while(_,_) : result += 1;
9     case \for(_,_,_) : result += 1;
10    case \for(_,_,_,_) : result += 1;
11    case \foreach(_,_,_) : result += 1;
12    case \catch(_,_): result += 1;
13    case \conditional(_,_,_): result += 1;
14    case \infix(_,"&&",_) : result += 1;
15    case \infix(_,"||",_) : result += 1;
16  }
17  return result;
18 }
```

Starting on line 3, the `visit` statement generates a recursive traversal and each `case` represents a pattern to detect in the traversed tree, and after the colon (`:`) an action to perform when it matches (increasing a counter). Most of the cases define simple patterns which identify a particular node type, but two are more interesting, filtering only infix expressions with either `&&` or `||` as operators. The `\` symbol before the pattern in cases is used to escape a constructor identifier that is a reserved word in Rascal. The variable name as `_` is a special symbol to discard the bind value during the matching, as in this example the bind values are not relevant to the metaprogram. Variable `result` count the number of conditional branching for the input Java program and on line 17 it is returned as result of the function. It is important to note that this metaprogram example uses an ADT representation of the abstract syntax tree associated to the input Java program.

— 4 —

Pattern-Based Test Design Process

To improve and systematise the pattern-based metaprogram test process, in this chapter is presented a test design process based on patterns. As presented in Section 1.1, there is a lack of methods and techniques to the testing of metaprograms. This design process is based on patterns and provides a method to use the pattern information from pattern-based metaprograms to design a test data set for the metaprograms's test. An overview of this process' structure is presented in Figure 4.1.

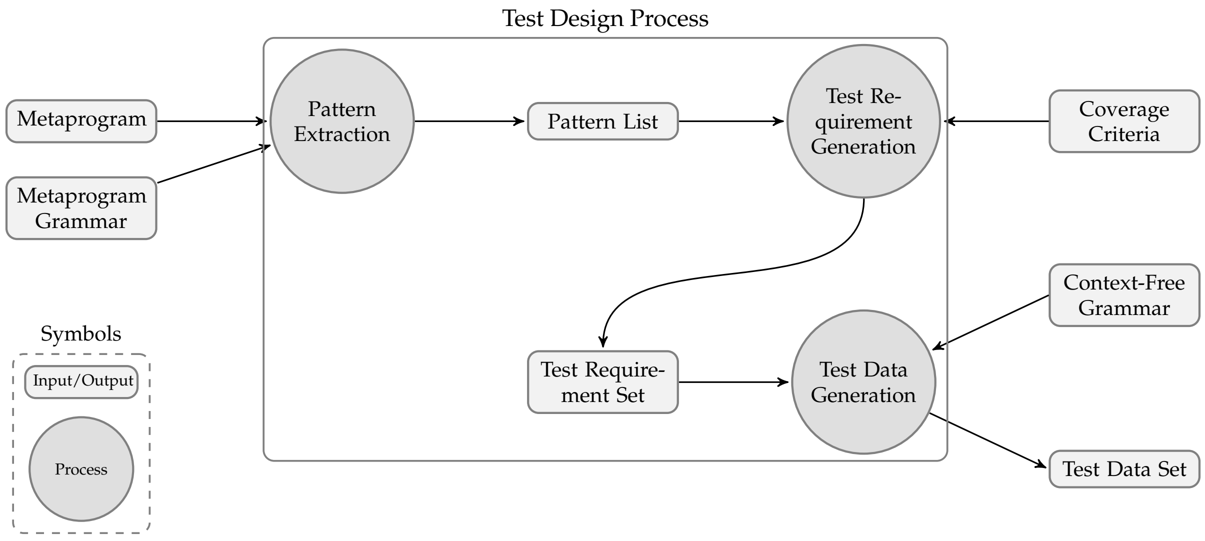

Figure 4.1: The pattern-based test design process overview.

The process receives a metaprogram, coverage criteria and a context-free grammar as input. The metaprogram is the program where the pattern information is extracted. To extract the patterns from the metaprogram, the metaprogram grammar is also necessary. The coverage criteria and the context-free grammar are used to generate a requirement set that is used during the test data generation. This grammar defines the object language used by the extracted patterns and by the test data generation process to produce the sentences. There are three subprocesses that compose the main process: the pattern extraction, the test requirement generation, and the test data generation. The first subprocess is responsible to identify and extract the patterns from the input metaprograms. The next step is to use these patterns to produce a test requirement set. This process is guided by the coverage criteria. The test requirement set is used by the test data generation to produce a test data set. This set satisfies the original requirements defined by the coverage criteria. Each subprocess presented in Listing 4.1 is explained in detail in each subsequent section in this chapter.

The test design process must be instantiated to be used. This instantiation consists of defining few concrete elements for the process. The metaprogram programming language, pattern language (the language used to define patterns), and the object language are examples of these concrete elements that have to be defined during this instantiation process. All these elements have to be defined in the pattern extraction subprocess and the test data generation subprocess. A concrete instantiation example is presented in details on Section 6.2.

4.1 Pattern Extraction from Metaprograms

The main goals in the pattern extraction process are extracting the pattern instances from the pattern-based metaprogram and processing these patterns when necessary. This process relays directly on the kind of artifact where the patterns are placed. A metaprogram with the patterns hard-coded in it or a metaprogram specification with support to a pattern language (the language that specifies the patterns on the artifact) are examples of these artifacts. The pattern language is important because it is used to identify the patterns in a specific artifact. The identified patterns are extracted from the artifact and these are added to a list of patterns. This list is used by the next step on the test design process.

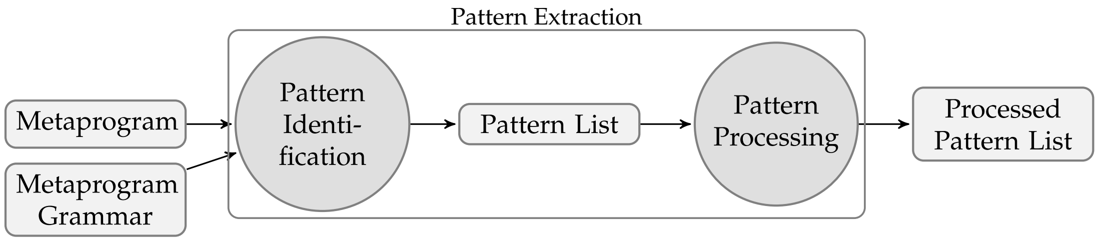

Figure 4.2: The pattern extraction process overview.

The pattern extraction process is divided into two subprocesses: pattern identification and pattern processing. These two subprocesses are presented on the Figure 4.2. In the figure, the input for the process is a metaprogram under test and the output is a processed pattern list with the processed patterns. The identification process is responsible to parse the metaprogram and identify the patterns used on it. The pattern processing step is responsible to apply a user-defined processing algorithm to the pattern list. These two subprocesses are detailed in the following sections.

4.1.1 Pattern Identification

The patterns can be extracted from many sources, like a metaprogram, a specification, or a documentation. Our current metaprogram extraction is based on the hard-coded pattern instances in the metaprogram under test. On this case, the metaprogram programming language has the pattern support builtin. The pattern language defined as part of the metaprogram programming language can be used to identify the pattern instances. When the patterns are not coded on the metaprogram itself, it is also possible to identify the patterns from another related artifact. For example, techniques of pattern recognition [Bishop, 2006] and deep learning [Aizenberg, 1999] can be used to identify these patterns. There are also some research projects using these techniques to discover a different type of patterns from source code, for example [Christodorescu and Jha, 2003; Zhang et al., 2016; Allamanis et al., 2016; Allamanis and Sutton, 2014; Kirasić and Basch, 2008]. However, the present work does not use these techniques currently. Instead, we use the hard-coded metaprogram pattern instances as the source for the extraction process. We do not use these techniques because our work is focused on proposing an initial test design process based on the pattern and, consequently, the investigation and implementation of these techniques are complex and time demand tasks.

For the patterns coded into the metaprogram, we need to have access to the grammar definition and the pattern language of the metaprogram programming language of the metaprogram under test. The definitions are used to parse the metaprogram under test and identify the pattern instances inside it. The first step in the pattern identification is parsing the metaprogram. Using the context-free grammar definition, the metaprogram is parsed and the patterns can be identified. When the abstract syntax tree is produced by the parsing process, the pattern related nodes can be identified using the pattern language definition.

Each pattern instance is identified and labeled using a unique ID per instance. It is possible to have two pattern instances with the same contents but used in different places in the metaprogram. In some scenarios, the described situation is relevant and the use of these labels solves the problem. These labels also simplify the reference to the pattern instances on the other parts of the test design process.

The pattern instance itself and its label ID can be enough information to deal with these pattern during the next steps in the process. But, there are cases where more information about the pattern instance is necessary. For example, the information about the code position of a pattern instance or module where it resides can be essential in some cases. So, it is important to have an additional pattern instance information gathering process that collects this kind of additional information. The instantiation presented on Chapter 6 for example, instantiate the pattern extraction process only using the information of the pattern instance, function name, the ID label, the instance position.

At the end of the pattern identification process, a processed pattern list is produced with the related instance information. The list below summarizes all activities related to the process:

1. Parse the metaprogram under test;
2. Identify the pattern instances over the abstract syntax tree;
3. Add a label for each pattern instance found;
4. Get additional information about the instances.

4.1.2 Pattern Processing

There are situations where the number of patterns on the metaprogram is big enough to increase the complexity of using a coverage criterion with it. For example, the case study is shown in Section 6.2 uses the Rascal type checker module to show the application of the pattern test process and the related coverage criteria. In this case, the number of pattern instances found in the module was 1,137 where 373 of them were used in the test design process. This restriction was made because of the complexity of generation process related to using all patterns. The module has 8,068 SLOC and mainly coded using patterns. Using the coverage criteria shown in Section 5.1, would be necessary 373 test requirements for the pattern coverage criterion (Criterion 5.1) and approximate 1.08×10^{223} test requirements to satisfy all combinations coverage criterion (Criterion 5.4). These cases represent the best and worst cases related to the cost of the test process using these criteria.

This kind of problem can be treated by introducing a processing step before the use of the pattern list produced by the extraction process. An implementation to deal with the described problem can be implemented in this intermediary step. This kind of customization has to be implemented during the instantiation process. The problem described early is discussed and treated in the instantiation presented in Section 6.2.2. In the process instantiation, this processing step can be specified and implemented to achieve the particular instantiation needs and integrate its results to the rest of the process.

This user-defined process can be used to implement other tasks necessary to prepare the pattern list to the subsequent steps in the test design process. For example, the pattern process can be used to externalize some information about the pattern instance and the context where it is used to a third-party test application. This kind of flexibility can help the pattern-based test design process to be integrated with an external test process.

At end of the user-defined processing, the pattern process must produce a result pattern list with the basic information of the pattern instance ID, its string representation, type of the instance, and an information about its context. This set of information is the default, but during the instantiation process this set can be changed if it is needed.

4.2 The Test Requirements Generation

The pattern extraction process identifies and prepares the pattern to be used with the coverage criteria. The next step is to use the result pattern list to generate the test requirements based on the desired coverage criteria. This process has the goal to use a coverage criterion to generate a test requirement set to be used to generate the test cases. The algorithms related to each coverage criteria based on patterns should be implemented during the pattern-based test design process instantiation. These algorithms will be used to generate the requirement set. On Chapter 5, we propose a set of coverage criteria based on patterns and inspired by the equivalence partition strategy. The test requirement generation can use any coverage criterion that is based on patterns and not only the proposed criteria. The proposed criteria are an example set and it is used during the conducted evaluation presented on Chapter 6.

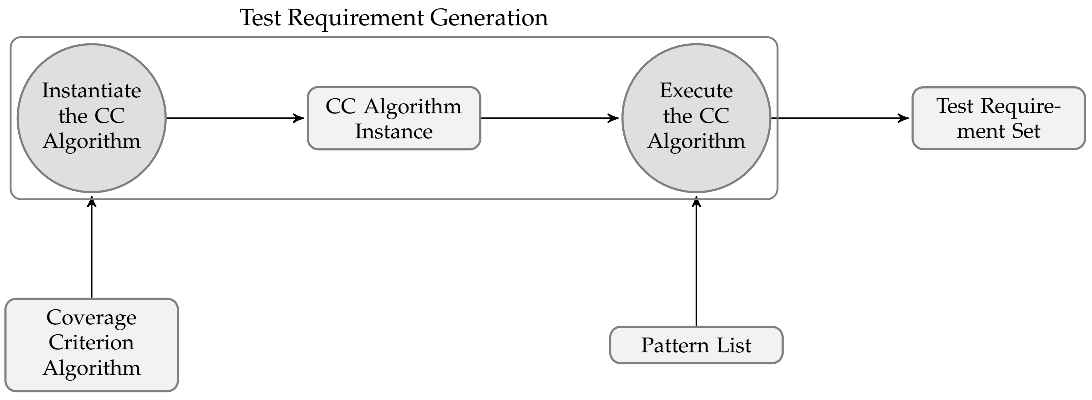
Test Requirement Generation
Instantiate the CC Algorithm
CC Algorithm Instance
Execute the CC Algorithm
Test Requirement Set
Coverage Criterion Algorithm
Pattern List

Figure 4.3: Overview of the structure for the test requirement generation process.

The key component of the test requirement generation is the coverage criteria algorithm responsible to generate the requirements based on the pattern list. The Figure 4.3 presents an overview about the test requirement generation process. The coverage criteria algorithm must be provided to the process and it is instantiated also by the process. This instantiation is responsible to configure the algorithm to a specific execution necessity. The algorithm instance is executed using the pattern list and the test requirement set is produced as the result.

The test requirement set is defined in function of the pattern instances and the matches operations. Each requirement element of the set is a tuple composed by the pattern ID and the operation over that pattern. This test requirement set is presented in Definition 4.2. To simplify that definition, we start presenting the basic notation that will be used by the requirement set definition.

Definition 4.1 (*Base definitions*)

- There is a Boolean set $B = \{True, False\}$;
- The pattern set $\mathcal{P}_m$ is the set of pattern IDs extracted from the pattern list for a metaprogram m;
- The matching operations set is $O = \{\succ, \not\succ\}$;
- For a pattern IDs $p \in \mathcal{P}_m$ and a subject s, we can simplify the notation of properties of match $p \succ s$ and not match $p \not\succ s$ as p^s and $\bar{p^s}$, respectively;
- When the subject s is clear in the context, it can be suppressed and represented by p and $\bar{p}$;
- If there is a subject s such as $p^s \wedge q^s$ for any two patterns $p, q \in \mathcal{P}_m$ then pq^s is defined. The same holds for the not match operation;
- The following equivalences are defined, where s is a subject:
  - $\bar{p}q \equiv p \not\succ s \wedge q \succ s$;
  - $p\bar{q} \equiv p \succ s \wedge q \not\succ s$;
  - $\bar{p}\bar{q} \equiv p \not\succ s \wedge q \not\succ s$;
  - $p_1 \ldots p_n \equiv p_1 \wedge \ldots \wedge p_n$.

With the basic definitions presented on Definition 4.1, we now can define the test requirement set that is shown on Definition 4.2. The test requirement set TRS defined here is important because it is used as the reference by the coverage criteria algorithm and also on the next step related to the test data generation process.

Definition 4.2 (*The Pattern Test Requirement Set*)

The pattern test requirement set $TRS \in \mathbb{P}(\mathcal{P}_m \times O)$ for a metaprogram m is a set of test requirements composed by patterns and matches operations.

To illustrate the use of the TRS, we present an example based on a pattern set

$\mathcal{P}_m = \{p_1, p_2, p_3\}$ extracted from a hypothetical metaprogram. Imagine that a coverage criterion based on patterns produced a test requirement set $TRS = \{(p_1, \succ), (p_2, \not\succ), (p_3, \succ)\}$. Applying the simplified notation, we have the set $TRS = \{p_1, \bar{p_2}, p_3\}$. For the requirement p_1, the test cases must have a test case represented by s_1 that satisfies $p_1 \succ s_1$. The requirement p_2 must have a test case represented by s_2 that satisfies $p_2 \not\succ s_2$. Finally, the requirement p_3 must have a test case s_3 that satisfies $p_3 \succ s_3$.

The process used to produce the test cases, like s_1, s_2, and s_3, is called Test Data Generation and it is discussed and defined in the next section.

4.3 Test Data Generation

The test data generation process has as main goal the generation of a test data set that satisfies a test requirement set. This requirement set was generated by the test requirement generation process. The test requirement set, specified by Definition 4.2, is used as a reference during all definitions presented in this section.

As described in Chapter 2, the test cases can have different levels of abstraction. For this proposed process, the test data set is generated at an abstract level, so it cannot be used directly during the test execution. However, the concrete test data set is directly related to a specific metaprogram environment and test execution environment. This transformation of the test data set to a concrete level is not difficult to be produced and it must be produced during the test design process instantiation. An example of this, it is presented in Chapter 6 when the pattern-based design test process is instantiated, a concrete test set is generated, and the test data is presented (Section 6.2.4). Another relevant fact is that the expected result of a test case is also related to a specific environment configuration, like abstract to concrete test data set transformation. Because of that, to produce a final concrete test case is not suitable to be defined at the process level, instead, it must be defined during the instantiation process.

4.3.1 Definition of The Test Data Generation Problem

The test data generation problem is the problem of generate a test data set that satisfies the original test requirement set. To give a precise definition to the problem we define the **Pattern-Based Test Data Satisfaction Relation** which defines precisely what represents the test requirement satisfaction by a test data. In this section this relation is defined. The first step to construct this relation is present a basic notation to use during the subsequent definitions. The Notation 4.1 is presented below:

Notation 4.1: *Consider:*

– *The set B from Definition 4.1;*

- *An object language L;*
- *A pattern set P that describes structures over L;*
- *The sets O from Definition 4.1 and TRS from Definition 4.2;*
- $p \equiv (p, \succ)$ *and* $\bar{p} \equiv (p, \not\succ)$*;*

The Definition 4.3 presents the function which verify the satisfaction relation between a test requirement from TRS and a subject from the object language L. If the subject of L satisfies a requirement of TRS then the function result is true, otherwise, the result is false.

Definition 4.3 (*Requirement Pattern Matching Function*)

Given an object language L, the test requirement set TRS, and the set B, a function *requirement pattern matching*, denoted by $\theta : TRS \times L \to B$, defines the relationship between the requirement pattern, the match operation, and the subject sentence. This function is defined as:

$$\theta(t, s) = \begin{cases} p \succ s, & \text{if } t = p \\ p \not\succ s, & \text{if } t = \bar{p} \end{cases}$$

To illustrate the use of the function, we present an example using the pattern set $\mathcal{P}_m = \{p_1, p_2, p_3\}$ and the test requirement set for the set $\mathcal{P}_m$ is $TRS = \{p_1, \bar{p_2}, p_3\}$. For this case, if there is a set of subjects $\{s_1, s_2, s_3\}$ where each of them matches with a pattern in the TRS using the presented order then the application of the function θ on the requirements and subjects resulting in true, false, and true. The result for the requirement $\bar{p_2}$ and subject s_2 is false because was assumed that $p_2 \succ s_2$.

With the Definition 4.3, we can verify the satisfaction of only one pattern associated with a requirement at the time. But the set TRS defined on the Definition 4.2 is possible to have multiple pattern and matches operations associated with each requirement element. So, the *pattern matching relation* expands the satisfaction verification to multiple elements.

Definition 4.4 (*Pattern Matching Relation*)

Given an object language L, a subject $s \in L$, and a test requirement $tr = m_1 \ldots m_n$, s **satisfies** tr iff $\theta(m_1, s) \land \ldots \land \theta(m_n, s)$. This relation is denoted as $s\,\theta\,tr$.

Using the pattern matching relation definition is possible to verify for a subject s if it satisfies a requirement element from TRS composed of many patterns and matches operations tuples. For example, a situation where we have a pattern set $\mathcal{P}_m = \{p_1, p_2, p_3\}$, a $TRS = \{p_1p_2, p_2\bar{p_3}\}$, and a subject $s \in L$. Aditionally, the $s \,\theta\, p_1p_2$ and $s \,\theta\, p_2\bar{p_3}$ are true. For the described situation, the s can satisfy all requirements. But, this possibility is not always the case. In this new situation, we have the same $\mathcal{P}_m$ and a new $TRS = \{p_1p_2, \bar{p_2}p_3\}$. In this case, it is impossible to find a subject s that satisfy both requirements p_1p_2 and $\bar{p_2}p_3$.

The last relation to be defined is a result of the previous definitions and a generalization using a set of subjects. This set of subjects is the test data set that will be used during the testing of the metaprogram to satisfy the original test requirement set.

Definition 4.5 (*Test Data Set Satisfaction Relation*)

Given a test data set $TD = \{s \in L\}$ and a test requirement set $TR \subseteq TRS$, TD **satisfies** TR iff $\forall tr \in TR, \exists s \in TD \bullet \theta(tr, s)$. This relation is denoted by $TD \,\Theta\, TR$.

Finally, using the defined relation, we can present more precisely the problem of generating a test data set that satisfies a test requirement set. This problem can be stated as "given a test requirement set TR, produce a test data set TDS where $TDS \,\Theta\, TR$". In the next section, the generation problem is treated in details and an abstract reference algorithm is presented.

4.3.2 The Reference Test Data Generation Algorithm

The test data set TD that satisfies the test data set satisfaction relation can be generated using an algorithm. In this section is presented a reference abstract algorithm that can be used to generate this test data set using a predefined pattern-based test requirement set. At the end, few examples are presented to illustrate the main concepts showed by the algorithm.

The Definition 4.6 presents the algorithm. It has as input a context-free grammar G and the pattern-based requirement set TR. The G grammar is used to produce the test data sentences. The requirement set is an instance of the set defined on the Definition 4.2 and it can be constructed by a coverage criterion. The output of the algorithm is a test data set TD that satisfies the original requirement set. These elements of the test data set are sentences from the object language defined by the grammar given as input. These sentences can be used to test the metaprogram under test, but to have an executable test set it is necessary to complement the test data set with additional information like the expected results.

Definition 4.6 (*Test Data Generation Reference Algorithm*)

Input:

1. A context-free grammar $G = (N, T, S, F)$. This grammar is reduced [Reghizzi, 2009] and it represents the object language;
2. A pattern-based test requirement set $TR \subseteq TRS$.

Output:

1. A test data set $TD = \{s \mid s \in \mathcal{L}(G)\}$ and $TD\,\theta\,TR$

Assuming:

1. TR only contains satisfiable requirements. So $\forall t \in TR, \exists s \in \mathcal{L}(G) \bullet s\,\theta\,t$;
2. The relation $\circ : N \times TRS \times B$ defines the satisfaction relation between a nonterminal A and a requirement t. This relation is defined as
$A \circ t \equiv \exists w \in T^{+} \bullet A \Rightarrow^{*} w \wedge w\,\theta\,t$;
3. The set $V_{\circ}^{t} = \{A \in N \mid A \circ t\}$;
4. The set TD starts empty.

For each $t_i \in TR$, the steps of the algorithm are:

1. If $\exists s \in TD \bullet s\,\theta\,t_i$ then go to the next requirement t_i;
2. Discover a symbol $A \in N$ such as $A \circ t_i$. The set of all symbols for t_i is denoted by $V_{\circ}^{t_i}$;
3. Generate a sentence set W such that $W = \{w \in \mathcal{L}(G) \mid S \Rightarrow^{*} \alpha A \beta \Rightarrow^{*} w\}$;
4. Choose a sentence s from W such that $s\,\theta\,t_i$;
5. Add s to TD.

The main strategy of the algorithm is to find a nonterminal symbol in G for each requirement in TR such that it can generate a subject that satisfies the requirement. The first step has as the goal the restriction of the sentences in the result test data to those which contributes to satisfy a new requirement. The next step defines the process responsible to discover a nonterminal symbol that satisfies the current test requirement. This process is always possible to define and always return a symbol, it because all requirements in the TR are satisfiable. This fact means that the object language always has a string that satisfies the current requirement. The step three generate a set of sentences W from the object language that each of them has to use the nonterminal symbol found in the prior step. This is always possible because the context-free input grammar has to be reduced. The next step is to choose a sentence from the W set which it satisfies the current requirement. These two last steps process can be refined to achieve an optimization over the process, for example, join the two step into one to improve the search for the sentences that satisfy the requirement. Additionally, these steps can be modified aiming some additional characteristic over the result test data. This kind of refinements can be implemented on the test design process's instantiation. The final step adds the produced sentence to the result test data set TD.

To illustrate the use of the algorithm, there is an example using it for a hypothetical situation. This example illustrates a simple test requirement set where there are two requirements for a match and not match situations. The context-free grammar $G_1 = (\{S, A, B, C\}, \{a, b, c\}, S, F)$ is used and the set of rules F is defined below:

1. $S \to AC$
2. $A \to aB$
3. $B \to bB$
4. $B \to b$
5. $C \to c$

The number on the left of each rule is an identifier used as reference. Additionally, a pattern $p_1 = vv$, a variable $v \in T^+$ and the test requirement $TR = \{p_1, \bar{p_1}\}$ are defined. Starting the algorithm using the step 1, we have the requirement $t_1 = p_1$:

1. There is no $s \in TD$ such as $s\, \theta\, t_1$;
2. The set $V_\circ^t = \{S, A, B\}$, so there are multiple nonterminal that satisfy the condition. For this example we choose B;
3. From B symbol it is possible to generate the sublanguage b^n where $n \geq 1$. To satisfy the p_1 requirement it possible to use any s from bb^m where $m \geq 1$. To generate W we choose to use the derivation from $S \Rightarrow^* aBc$ and from these strings we can produce a infinite set expanding B symbol. A possible set W based on this is $W = \{abbc, abbbc, abbbbc\}$;
4. From W we select the string $abbc$ that satisfies p_1, so $bb\, \theta\, p_1$;
5. Now $TD = \{abbc\}$.

Now, we follow the algorithm to the next iteration. Again, using the step 1 from the algorithm, the requirement $t_2 = \bar{p_1}$:

1. There is no $s \in TD$ such as $s\, \theta\, t$;
2. The set $V_\circ^t = \{S, A, B, C\}$ and for this example we choose B;
3. To satisfy the $\bar{p_1}$ requirement there is only one possible s string and it is b. So, using the same strategy to generate W, we choose to use the derivation from $S \Rightarrow^* aBc$. The unique set possible for this requirement is $W = \{abc\}$;
4. From W we select the string abc that satisfy $\bar{p_1}$, so $b\, \theta\, \bar{p_1}$;
5. Now $TD = \{abbc, abc\}$.

As there is no requirement left into the requirement set, the result test data set is $TD = \{abbc, abc\}$. The result of the generation algorithm is the TD set, but this set is not a concrete test set. Additional processing is necessary to produce a concrete test set to use directly on the test of a metaprogram. The concrete test set must have information like expected results.

— 5 —

Pattern-Based Coverage Criteria Family

This chapter presents the pattern-based coverage criteria. These criteria are used to systematize and improve the quality of the pattern-based metaprogram testing process. The patterns are extracted from the metaprograms under test, as presented in Chapter 4, and used to define the test requirements for each coverage criterion proposed in this chapter.

The first coverage criteria presented is the *Pattern Coverage Criterion* (PCC). In general, this criterion defines that each one of the pattern instances of a metaprogram should be tested by the test set. In this way, the source-code related to the pattern instance should be executed during the execution of the metaprogram with the test set. This criterion was the first one to be defined during the research and its results were presented in the paper [Hentz et al., 2015]. It is the simplest coverage criterion proposed.

Criterion 5.1 (*Pattern Coverage Criterion - PCC*)

For a pattern-based metaprogram m and $\mathcal{P}_m$ being the set of all pattern instances on m, the test requirement set TR_{PCC} contains at least one test case that matches each pattern in $\mathcal{P}_m$. Therefore, we can define the test requirement set as

$$TR_{PCC} = \{(p, \succ) \bullet \forall p \in \mathcal{P}_m\}.$$

When the metaprogram m is clear in the context, $\mathcal{P}$ is used instead. During the coverage criteria presentation, examples will be given based on the Rascal metaprogram presented on the Listing 5.1. The metaprogram implements simple repetition commands counter for Java programs. The repetition commands are `while`, `for`, and `foreach`. The metaprogram is based on the concrete patterns to localize the commands in a Java program instance. Thus, its pattern set is $\mathcal{P} = \{p_1 =$ `while(a,b)`, $p_2 =$ `for(a,b,c,d)`, and $p_3 =$ `foreach(a,b,c)`$\}$. Using the pattern coverage criterion on this example we have the requirement set $TR_{PCC} = \{p_1, p_2, p_3\}$, the tuple $(p_i, \succ)$ is represented by p_i, as presented on Definition 4.1. The size of TR_{PCC} is equal to $|\mathcal{P}|$, so in this case $|TR_{PCC}| = 3$. The *minimal test set size* for this criterion is 1 because the test set can have only one test case that satisfies all requirements in general. This example presents this kind of situation as a Java program can contain the three repetition commands. Thus, a generated test set should satisfy each of the patterns in the $\mathcal{P}$.

Listing 5.1: A Rascal metaprogram to count the repetition commands in Java programs.

```
int calcRept(Statement impl) {
  int result = 0;
  visit (impl) {
    case \while(a,b) : result += 1; //P1
    case \for(a,b,c,d) : result += 1; //P2
    case \foreach(a,b,c) : result += 1; //P3
  }
  return result;
}
```

A possible test set which satisfies the TR_{PCC} test requirement set is presented on Listing 5.2. Observing the example, we can see that each pattern in the TR_{PCC} is satisfied by a Java program listed into each line of the example. For example, the requirement p_1 is satisfied by the program on line 1. The pattern `while(a,b)` associated with the requirement matches with the code `while (true) return ;`. The variables `a` and `b` are bound to codes `true` and `return` in this particular case. Each Java program presented in the example is semantically meaningfulness but they are valid and compilable. It is important to remember that this example is only one possible test set that satisfies the TR_{PCC}.

Listing 5.2: An example of a test set that satisfy the TR_{PCC} set.

```
1 public class id0 {static { while (true) return ; }} //P1
2 public class id0 {static { for (  ;  ;  ) return ;}} //P2
3 public class id0 {static { for ( int id1 : id2 ) return ; }} //P3
```

5.1 The Pattern-Based Equivalence Partitioning Criteria

Although the pattern coverage criterion presents a viable alternative and with relatively low cost related to the generation of the test data, we can use more elaborated criteria for the pattern-based test process. A way to improve the quality of the test requirements is using the equivalence partitioning techniques. The use of these techniques facilitates the test data generation process because each of the criteria has its own cost and quality attributes [Myers et al., 2011]. These techniques were adapted from the traditional definition and they will be presented below.

5.1.1 The Equivalence Partition Methodology

To improve the first proposed pattern-based coverage criterion, we inspired on the main concepts of the equivalence partitioning techniques. In this section, we will explain the instantiation of those techniques applied to the patterns. Because that, the main structure and terminology used in this presentation are defined in Section 2.1. The first step is the discussion about the definition of the partitions and blocks. After that, it ends with an example of this process on the metaprogram presented in Listing 5.1.

On pattern based criteria we have a pattern set $\mathcal{P}_m$ which will be used to define the partitions of the input space. In this case, we assume a metaprogram m that always has one argument, a program written in the object language, as presented in Chapter 3. On the current example case, this input program is written in Java language.

Partitions \ Blocks	b_1	b_2
q_1 based on p_1	p_1	$\bar{p_1}$
...	...	...
q_n based on p_n	p_n	$\bar{p_n}$

Table 5.1: Pattern matching partitioning for a pattern set $\mathcal{P}_m$.

An essential characteristic of the partition equivalence is that the identified blocks are mutually exclusive. This means that two blocks from the same partition cannot have common values. In our pattern case, we do not have this kind of exclusion between patterns, but we have for a pattern in relation to a subject that matches or not. Our blocks are based on the match and not match operations for a pattern. The partitions are defined using each pattern in the pattern set as a partition. This model is presented on Table 5.1, in it $n = |\mathcal{P}|$ and each q_i represents a partition.

A test requirement $(p_1, \ldots, p_n)$ is an example of an element from this partitioning. This requirement means that a metaprogram m should be tested with a set of input sentences that matches, in this case, the patterns $p_1, \ldots, p_n$. Using Definition 4.1 from Section 4.2, we use the notation $p_1 \ldots p_n$ to denote the same meaning.

The metaprogram example of Listing 5.1 is used to illustrate an application of this partitioning and this partitioning is presented on Table 5.2. In this case, the $\mathcal{P}_m = \{p_1, p_2, p_3\}$ and the partition set is $Q = \{q_1, q_2, q_3\}$ each of them composed by two blocks. The block 1 represents the match and the block 2 the not match. A test requirement example, in this case, is $p_1\bar{p_2}p_3$.

Partitions \ Blocks	b_1	b_2
q_1 based on p_1	p_1	$\bar{p_1}$
q_2 based on p_2	p_2	$\bar{p_2}$
q_3 based on p_3	p_3	$\bar{p_3}$

Table 5.2: Pattern matching partitioning for a pattern set $\mathcal{P} = \{p_1, p_2, p_3\}$ from the example.

The presented partitioning model will be used to define the pattern-based coverage criteria family in the following sections.

5.1.2 The Each Choice Pattern Coverage Criterion

The first coverage criteria based on the equivalence partition technique is the *Each Choice*, Criterion 2.2 in Section 2.1. This criterion leads to two requirements for each pattern on the metaprogram pattern set, one of them defines that at least one sentence on the test set matches the pattern and other requirement defines that at least one sentence does not match the pattern. This coverage criterion is defined more precisely below:

Criterion 5.2 (*Each Choice Pattern Coverage Criterion - ECPCC*)

For each pattern p contained in the set $\mathcal{P}$, two requirements are defined: (1) the test set must have at least one sentence s_1 such that it satisfies the pattern requirement p, (2) the same test set must have at least one sentence s_2 that does not satisfy the pattern requirement p. The test requirement set for this coverage can be defined as:

$$TR_{ECPCC} = \{p, \bar{p} \bullet \forall p \in \mathcal{P}\}.$$

To illustrate the use of the each choice criterion is used the previous example, Listing 5.1. Thus, we can define the requirement set using the ECPCC for the example, the set is defined as $TR_{ECPCC} = \{p_1, p_2, p_3, \bar{p_1}, \bar{p_2}, \bar{p_3}\}$. To satisfy the each choice criterion the test set has to include sentences that match and do not match each pattern in $\mathcal{P}$. For example, the pattern $p_1 =$ `while(a,b)` can be satisfied by a Java program that includes the command `while (true) return ;` and its negation can be satisfied by any program that does not contain the `while` command.

The size of the test requirement set for this criterion is $|TR_{ECPCC}| = 2|\mathcal{P}|$ because we must have two requirements for each pattern: one for the match and another one for not match. Using Equation 2.2 from Section 2.1, we can explain it with more details. In the ECPCC case, we have one partition with two blocks $[p_i, \bar{p_i}]$ for each pattern $p_i \in \mathcal{P}$. Because the size of each block is always the same, with $n = |\mathcal{P}|$, we have as result $|TR_{ECPCC}| = 2n$ or $|TR_{ECPCC}| = 2|\mathcal{P}|$. Applying this on the current example we have $|TR_{ECPCC}| = 6$. The minimal test set size can be defined using the same principle presented on Section 2.1 that defines this value based on the maximum value amoung the sizes of each block. For ECPCC case we have the blocks with the same size 2, so the minimum test set always has the size of 2.

Listing 5.3: A test set example that satisfies the TR_{ECPCC}.

```
1 public class id0 {static { while (true) return ; }}
2 public class id0 {static { for (  ;  ;  ) return ; }}
3 public class id0 {static { for ( int id1 : id2 ) return ; }}
4 public class id0 {static { if (true) return ; }}
5 public class id0 {static { while (true) return ;  for (  ;  ;  ) return ;  for
     ( int id1 : id2 ) return ;}}
```

Listing 5.3 presents one possible test set that satisfies the criterion. The three first sentences cover the requirements p_1, p_2, and p_3. The sentence on line 4 covers at same time the three negatives requirements: $\bar{p_1}, \bar{p_2}$, and $\bar{p_3}$. Besides that, these three requirements could be also covered by the sentences at lines 3, 2, and 1 respectively. The last line sentence covers at the same time three match requirements and none not match. This test set has redundant sentences in relation to the criterion satisfaction and it could be optimized. The generation of a minimal test set [1] is an interesting feature, but generating a minimal test set for a coverage criterion is not trivial. Because of this difficult, it is not aimed at this work.

5.1.3 The Pair-Wise Pattern Coverage Criterion

The next coverage criterion defined is the *Pair-Wise* pattern coverage, it defines that for each pair of patterns four requirements are defined, one for each combination of match and not match with a sentence. Thus, will be tested the pair-wise combinations for each pattern and its negation. This criterion is precisely defined below:

[1]The minimal test set is a set with the lowest number of elements necessary to satisfy a coverage criterion.

Criterion 5.3 (*Pair-Wise Pattern Coverage Criterion - PWPCC*)

For each pair of patterns $p_i, p_j \in \mathcal{P}$, there are four test requirements defined: (1) must have at least one sentence s_1 in the test set that matches both patterns; (2) must have at least one sentence s_2 that matches the first pattern and does not match the second; (3) must have at least one sentence s_3 that does not match the first pattern and matches the second; (4) must have at least one sentence s_4 that does not match either of patterns. The resulting requirement set is defined as:

$$TR_{PWPCC} = \{p_i p_j, p_i \bar{p_j}, \bar{p_i} p_j, \bar{p_i}\bar{p_j} \bullet p_i \neq p_j, \forall p_i, p_j \in \mathcal{P}\}.$$

Using the example of Listing 5.1, Table 5.3 presents the test requirement set for this example. To calculate the test requirement size for this criterion we used Equation 2.3 as a reference. However, for the pattern based coverage criteria family, we can simplify that equation based on the fact that our block always has the same size 2. So, we can calculate the size using the *Binomial coefficient* [Goodman, 2012; Cormen et al., 2009] to calculate the number of k combinations for a set of n elements, these elements on our case is the number of patterns in the pattern set. In the case of pair-wise the k is 2 and we have to multiply the result of the binomial coefficient by the result of the power of the block size by k. As result, we have the following equation to the pair-wise case:

$$2^2\binom{n}{2} = 4\binom{n}{2}. \tag{5.1}$$

Using it for the current example, the test requirement size is $4\binom{3}{2} = 12$.

The blocks $[p_1, \bar{p_1}]$ and $[p_2, \bar{p_2}]$	
p_1p_2	$p_1\bar{p_2}$
$\bar{p_1}p_2$	$\bar{p_1}\bar{p_2}$
The blocks $[p_2, \bar{p_2}]$ and $[p_3, \bar{p_3}]$	
p_2p_3	$p_2\bar{p_3}$
$\bar{p_2}p_3$	$\bar{p_2}\bar{p_3}$
The blocks $[p_3, \bar{p_3}]$ and $[p_1, \bar{p_1}]$	
p_3p_1	$p_3\bar{p_1}$
$\bar{p_3}p_1$	$\bar{p_3}\bar{p_1}$

Table 5.3: The test requirement set for the pair-wise pattern coverage for the example case.

Using the pair-wise minimal test set equation from Section 2.1, the minimum test set size for the current example is 4. In this case, the one possible test set is $\{p_1p_2\bar{p_3}, p_1\bar{p_2}p_3, \bar{p_1}p_2p_3, \bar{p_1}\bar{p_2}\bar{p_3}\}$. Table 5.4 shows the satisfaction relation between each abstract test case and the satisfied test requirement.

A possible concrete test set to satisfy the TR_{PWPCC} is presented in Listing 5.4. The first line satisfies the requirements p_1p_2, p_1p_3, and p_2p_3. The second one satisfies again the requirement p_1p_2 and the new $p_1\bar{p_3}$ and $p_2\bar{p_3}$. The requirements $p_1p_3, \bar{p_2}p_3$, and $p_1\bar{p_2}$ are satisfied by the sentence on line 3 and the $p_2p_3, \bar{p_1}p_2$, and $\bar{p_1}p_3$ by line 4. The others $\bar{p_1}\bar{p_2}, \bar{p_1}\bar{p_3}$, and $\bar{p_2}\bar{p_3}$ by the sentences in the last lines.

Test Case $p_1p_2\bar{p_3}$		
p_1p_2	$p_2\bar{p_3}$	$\bar{p_3}p_1$
Test Case $p_1\bar{p_2}p_3$		
$p_1\bar{p_2}$	$\bar{p_2}p_3$	p_3p_1
Test Case $\bar{p_1}p_2p_3$		
$\bar{p_1}p_2$	p_2p_3	$p_3\bar{p_1}$
Test Case $\bar{p_1}\bar{p_2}\bar{p_3}$		
$\bar{p_1}\bar{p_2}$	$\bar{p_2}\bar{p_3}$	$\bar{p_3}\bar{p_1}$

Table 5.4: The satisfaction of TR_{PWPCC} for the current example using the minimum test set.

Listing 5.4: An example of test set that satisfies the TR_{PWPCC}.

```
public class id0 {static { while (true) return ;  for (  ;  ;  ) return ;  for
    ( int id1 : id2 ) return ;}}
public class id0 {static { while (true) return ;  for (  ;  ;  ) return ;}}
public class id0 {static { while (true) return ;  for ( int id1 : id2 ) return
    ;}}
public class id0 {static { for (  ;  ;  ) return ;  for ( int id1 : id2 )
    return ;}}
public class id0 {static { if (true) return ; }}
```

5.1.4 The All Combinations Pattern Coverage Criterion

The last coverage criterion to be defined is the *All Combinations*, it defines that all combinations between the matches and not matches must be tested. This is the more complete and rigorous of the pattern criteria.

Criterion 5.4 (*All Combinations Patterns Coverage Criterion - ACPCC*)

All combinations of matches and not matches amoung all patterns in $\mathcal{P}$ are present in the test requirement set. Thus, it defines requirements that a sentence match with all patterns does not match with all patterns and all intermediary combinations between them. The test requirement set is defined as

$$TR_{ACPCC} = \{ \begin{array}{c} p_1 \dots p_n, \\ p_1 \dots \bar{p_n}, \\ \dots, \\ \bar{p_1} \dots \bar{p_n} \end{array} \}$$

with $n = |\mathcal{P}|$ and $n \geq 1$.

Using the current example, Listing 5.1, the test requirement set for this example is presented below. Using the same principle of Equation 5.1, we can define the equation to calculate the test requirement size for the all patterns combinations criterion as

$$2^n \binom{n}{n} = 2^n. \tag{5.2}$$

As we can see in this case the k is equal to n because we have to create a combination of all elements in the set. Using this equation we can calculate the set size for the current example as $2^3 = 8$. Below, the test requirement set is presented

$$\begin{aligned} TR_{ACPCC} = \{ \quad & p_1p_2p_3, \quad p_1p_2\bar{p_3}, \\ & p_1\bar{p_2}p_3, \quad p_1\bar{p_2}\bar{p_3}, \\ & \bar{p_1}p_2p_3, \quad \bar{p_1}p_2\bar{p_3}, \\ & \bar{p_1}\bar{p_2}p_3, \quad \bar{p_1}\bar{p_2}\bar{p_3}\}. \end{aligned}$$

For this coverage criterion, in particular, the size of the test requirement set and the minimal test set is the same, as presented in Section 2.1. So, the $|TS|$ is also equal to 8.

5.2 The Subsumption Relation for the Coverage Criteria Family

Similarly to what happened with the equivalence partitioning criteria [Ammann and Offutt, 2008], the proposed pattern-based criteria present a subsumption relation (Definition 2.3 from Chapter 2) between them. Thus, we present and demonstrate in this section that there is a hierarchy relationship between the presented criteria family. The strategy to demonstrate it starts from the criteria definitions and it will demonstrate that it defines a test requirement set that is a subset of another criterion. Consequently, building a satisfaction hierarchy relation between them when they are analyzed.

We start assuming a set $\mathcal{P}$ of pattern instances from a metaprogram. Being TS_{PCC} a test set which satisfies the pattern coverage criterion (PCC), then, by definition, all the requirements p_i defined by the criterion must be satisfied by TS_{PCC}. Being TS_{ECPCC} a test set which satisfies the each choice criterion, then, by definition, all requirements p_i and $\bar{p_i}$ must be satisfied by TS_{ECPCC}. Therefore, the TS_{ECPCC} satisfies all requirements from the each choice criterion and consequently all the requirements of the pattern criterion. Hence, we have the subsumption relation between the pattern coverage criteria and the each choice coverage criteria. It can be denoted by

$$PCC \subseteq ECPCC.$$

The set TS_{PWPCC} is a test set which satisfies the pair-wise converge criterion and by definition, all combination pair wisely of the patterns p_i and p_j are satisfied by this set. By definition, an element $p_i p_j$ can be written as $p_i \wedge p_j$ and as result both patterns p_i and p_j are satisfied by it. Consequently, we can assume that any requirement p_i or $\bar{p_i}$, defined by the each choice criterion, this also be satisfied by the test set TS_{PWPCC}. For this reason, the TS_{PWPCC} satisfies all requirements from the each choice criterion. It can be denoted by

$$ECPCC \subseteq PWPCC.$$

The last case is the relation between the all combinations and the pair-wise criteria. Assume a test set TS_{ACPCC} which satisfies the all combinations criterion. By definition, all combinations of requirements p_i and p_j are satisfied by this test set. These combinations include all pair-wise combination of p_i and p_j, as result the requirements for the pair-wise criterion. Thus, the TS_{ACPCC} satisfies all requirements of the pair-wise criterion. This relation can be denoted by

$$PWPCC \subseteq ACPCC.$$

Therefore, we can conclude that the following subsumption relation is true for the pattern-base coverage criteria family.

$$PCC \subseteq ECPCC \subseteq PWPCC \subseteq ACPCC.$$

Thus, assuming the defined combination criterion and the same pattern set $\mathcal{P}$, we can conclude that all test set which satisfies the all combinations criterion also satisfies the pair-wise, all test set which satisfies the pair-wise also satisfies the each choice and all test set which satisfies the each choice also satisfies the pattern coverage criterion.

— 6 —

The Evaluation

This chapter presents the evaluation used during the verification of the pattern-based test design process and the coverage criteria family. The main goals of this evaluation are to verify the hypotheses proposed in Section 1.3, and to present the results of the use of the proposed test design process and the criteria family in a real situation. The evaluation shows some evidence on the effectiveness of use these proposals in a real-world situation.

To achieve the evaluation goals we have two main efforts: a case study using a complex pattern-based metaprogram and a verification process of the coverage criteria based on the mutation testing technique. The primary objective of the former is to present an instantiation of the pattern-based test design process, presented on the Chapter 4, using a complex metaprogram as an example. This instantiation can be used as a reference to the adoption of the test design process in practical situations. The latter objective is to measure the effectiveness and efficacy of the proposed coverage criteria family. This evaluation was executed in the first moment in a simple example, and then applied in the context of the complex metaprogram. These two aspects are the main concern related to the two hypotheses defined in Section 1.3. To achieve this goal we use the mutation testing technique to measure the quality of the test set associated with the proposed criteria.

To start the evaluation we define the evaluation methodology in Section 6.1. The information related to the specific goals, variables, and method used to conduct the evaluation are presented on the section. Section 6.2 presents the pattern-based test process applied on the Rascal type checker module. This case study helps to illustrate the different aspects related to the use of the test process. In Section 6.3, a detailed validation of the pattern coverage criteria family is presented. The execution of the coverage criteria family using two different software under test as an example is presented and the results obtained are analyzed and compared with other test techniques.

Some limitations and threats to validity related to the evaluation of the test process and coverage criteria are detailed and discussed in Section 6.4. All tools and experimental data are available online.[1]

6.1 Evaluation Methodology

The goal of the evaluation presented in this chapter is to investigate the hypotheses defined in Section 1.3. Hypothesis 1 is verified using two distinct strategies: a case study and a controlled experiment using the pattern-based coverage criteria family. The case study consists on the instantiation of the pattern-based test design process using a pattern-based metaprogram as a program under test. The controlled experiment consists of using a metaprogram to evaluate its test sets by the mutation technique. The metaprogram used is the same defined in the case study. The mutation technique is used to evaluate the test set quality and, consequently, the give information about the quality of the associated coverage criteria. The main assumption here is that the quality of the test sets that satisfy a given coverage criterion indicates the quality level of the coverage criterion. The mutation technique has been used in many studies to verify the test case quality [Andrews et al., 2005] and based on it, this technique was chosen to verify the quality of the generated test sets.

Hypothesis 2 is verified using the same mutation technique used in the controlled experiment. The test sets used to evaluate during the mutation process is the union of different test sets produced by the pattern-based and grammar-based coverage criteria. In the end, the cost and quality are analyzed in the same way that was used in the controlled experiment.

The first step to investigate Hypothesis 1 is to elaborate a test design process based on patterns. To investigate it, we elaborate a case study that instantiates the test design process, defined in Chapter 4, for a real pattern-based metaprogram. The test design process is instantiated using the Rascal as metaprogram language and the Rascal Type Checker(RTC), it is presented in Section 6.2.1, as metaprogram under test. The Rascal Type Checker is the Rascal component responsible for implementing the language type checker, and it uses the Rascal language to the code itself. This metaprogram was chosen because a type checker is a complex program in general and this metaprogram implementation as well. The complexity evaluation of the RTC was based on the number of lines of code and the kind of code used to develop it. Besides that, this program represents a real-word metaprogram that is used in a real-world project that still under development, the Rascal language project. Because of these characteristics, we believe that the RTC is a good choice to evaluate the effectiveness of a new test strategy.

[1] https://github.com/chentz78/PatternTesting

The goal of this case study effort is to answer the Research Question 1 and 2, and it is defined below:

Evaluation Goal 1: *Evaluate the cost of generating a test set using the pattern-based test design process for the Rascal Type Checker.*

The cost considered on Goal 1 is related to the instantiation of the test design process. The process instantiation is presented using each step defined on the abstract process of Section 4. Each of these steps has an associated cost, and it was considered during the case study to provide the total cost related to the goal. This cost evaluation is based on the size of the test sets. Another cost-related analysis is based on the complexity of each step necessary for the instantiated process. The amount of effort necessary to implement these steps is used for this analysis.

Evaluation Goal 2: *Provide an example instantiation to the pattern-based test design process.*

A secondary goal is to provide an example instantiation for the pattern-based test design process that can be used for another situation. This goal is related to the first one, and it is defined by the Evaluation Goal 2. The expected results are to improve the process documentation and make all artifacts produced by this instantiation process public. We believe that these actions will facilitate the future use of the test design process.

Another important goal of the evaluation process is answering Research Question 3. This goal is related to the quality of the test set generated by the test design process aiming to satisfy the pattern-based coverage criteria family. To answer the question we conducted a controlled experiment using the artifacts produced by the case study and validating it. The quality measure used on it is based on the mutation testing technique. This technique provides a metric based on the number of faults detected by the test set under validation. This validation is presented in Section 6.3 and we have the following evaluation goal:

Evaluation Goal 3: *Evaluate the mutation score for each test set related to each pattern-based coverage criteria.*

Each coverage criterion in the family has a generated test set that satisfies it and this set is used to evaluate the criterion. The mutation score represents the concrete quality indicator to evaluate the coverage criteria family. With the results produced by this evaluation, we can provide enough information to analyze the level of quality of the test set generated by the coverage criteria family.

Evaluation Goal 4: *Compare the mutation score and test set size between the pattern-based coverage and the grammar-based coverage criteria test sets.*

The Evaluation Goal 4 is related to the Research Question 3. On this goal, the quality variable is compared with the same information obtained from the grammar-based coverage test set. With the information produced by this goal, we can improve the understanding of the quality level associated with the pattern-based coverage criteria. The result of this goal is the test set related to the grammar-based criterion and their mutation score.

Evaluation Goal	Research Question	Hypothesis	Location
1	1	1	Sections 6.2 and 6.3
1	2	1	Section 6.2
2	-	-	Section 6.2
3	3	1	Section 6.3
4	5	2	Section 6.3

Table 6.1: Summary of the relation between evaluation goals and research questions.

Table 6.1 presents a summary of the relation between the evaluation goals, the hypotheses, research questions, and the parts of the chapter that investigate each of them. The evaluation goal 2 is not associated to any research question presented, but it is important to provide a concrete example of the test design process instantiation. All code and experiments executed during the evaluation phase used a reference computer MacBook Air 2013, CPU Intel i7, clock 1.7GHz, model 4650U, 8GB RAM, 128GB SSD and Mac OS version 10.11.2. The Rascal environment used was the "CommandLine REPL" run-time and its version was "0.8.4-20170622.094421-159".

6.2 Case Study

To illustrate and discuss with more details the use of the pattern-based test design process, and the coverage criteria family, this section presents a case study using a pattern-based metaprogram as an example. The example in use to the case study is a type checker implementation to verify Rascal metaprograms, and it also is made in Rascal. This metaprogram is used because of its complexity and proximity to a real-world metaprogram. The case study is also used in the verification process of the coverage criteria family. Besides that, it is used in a comparison with other test techniques.

Another important goal of this case study is to show the entire process of testing using the pattern-based test design process applied to a real metaprogram. The instantiation of the test design process presented here can be used as a reference to the future application of the process for different environments.

The case study presents an instantiation of the pattern-based test design process (PBTDP), presented in Chapter 4. It defines three steps to conduct the process: pattern extraction, test requirement generation, and test case generation. These steps are illustrated by the Figure 4.1 and these are used to conduct the case study process. Before starting, the process is important to define the metaprogram under test. In the next section, this metaprogram is presented with the motivation about its choice, its code, and its complexity.

Each step of the test design process is detailed in the following sections. The instantiation is structured following the same structure proposed by the test design process and divided into the following sections. The artifacts and results produced by these steps are also presented and discussed in its respective section.

6.2.1 The Rascal Type Checker Metaprogram

The Rascal type checker(RTC) is a metaprogram responsible for verifying the integrity of metaprograms written in the Rascal programming language. Its source code is available, and we used it as an example to illustrate the PBTDP. The RTC version used on this work was `4e9cd4e` [2] and it is available on the Rascal GitHub site [3]. This module implements the main core of the Rascal type system, it represents 71.44% of total implementation of the type system based on *source lines of code* (SLOC)[4][Conte et al., 1986; Pressman, 2001].

The RTC module has 8068 SLOC and 342 functions. The module uses the pattern matching mechanism in 166 functions, almost half of the functions implementations. There are 1137 pattern instances on the module. In our implementation, we treated three pattern kinds supported by Rascal and using this restriction the module has 373 pattern instances of these kinds. A detailed explanation of this restriction will be presented in the next section. The module also has a test set that is distributed on 17 test modules, and these are related to all kind of verification over the RTC module.

6.2.2 The Pattern Extraction in Rascal

This step is responsible for identifying and processing the pattern instances there are in the metaprogram under test. The main goal of this process is to collect these patterns and prepare the pattern information to provide it to the next step, the pattern-based test design process. This entire process is defined in Section 4.1. The result of this step is a pattern list, here denoted by $\mathcal{P}$, that is used by the next step in the process.

This process is divided into pattern identification and processing parts. In the first part is presented the Rascal metaprogram under test and information about the pattern list identified using the Rascal type checker. The second part is responsible for

[2]This is a simplified version of the complete hash `4e9cd4e2e4584363ae115a4e7f2631574ef0900f` used by the GitHub site.

[3]The Rascal type checker module is available on `https://git.io/vHPcb`.

[4]The SLOC value was collect from GitHub program's information page.

presenting the process over the list to collect the necessary information and to improve the performance of the next step related to the test requirement generation.

Pattern Identification

Following the proposed structure to the pattern extraction process, the first subprocess is to identify the pattern in the metaprograms. This step is composed of the following substeps:

1. Parse the metaprogram under test;
2. Identify the pattern instances over the abstract syntax tree;
3. Add a label for each pattern instance found;
4. Get additional information about the instances.

The substep 1 is responsible for parsing the metaprogram to identify the patterns. This parsing in this study case is based on the Rascal language grammar [5]. The Rascal environment defines his grammar, and for it, the Rascal parser is used. In this instantiation case, it is not necessary to create a parser manually. Another important point is the definition of the pattern language used by the metaprograms. The Rascal pattern language is defined as part of the Rascal grammar and the main syntax symbol that defines it is "`Pattern`", this symbol is fully defined in Listing A.3 presented in Appendix A.3.

The identification of pattern from the abstract syntax tree, the substep 2, is coded in the Rascal language and its implementation is listed in Listing 6.1. This implementation identifies the patterns from a tree and generates a pattern list. The function `getPatternTree` is the function used to implement the identification process in Rascal for the case study.

This function iterates over all function definitions in the metaprogram under test and identifies the patterns on each of them. The `for` command on line 8 implements this iteration, the `if` command on line 10 gets information about the current function, and the `if` commands on line 16 and 20 identify the type of the function implementation. At the end, the map variable `r` will store the resultant pattern list. On lines 25 and 27, the pattern's abstract syntax tree extraction is done. An auxiliary function `extractPattTree` extracts the patterns from the tree variable st.

[5]The Rascal grammar is defined on the module `Rascal.rsc` and the version used is available on `https://git.io/vQs3a`.

Listing 6.1: The pattern identify process implemented in the Rascal language. The metaprogram source language is also Rascal.

```
alias aGPRet = map[str , tuple[str fSig, list[Tree] patts]];

public aGPRet getPatternTree(Tree t, set[str] lblFSet) {
  aGPRet r = ();

  str nm,sig;

  for (/FunctionDeclaration f := t) {
    Tree st;
    if (f has signature) {
      nm = "<f.signature.name>";
      sig = "<f.signature>";
    }

    try {
      if (f has expression) {
        st = f.expression;
      }

      if (f has body) {
        st = f.body;
      }

      if (nm in r) {
        r[nm].patts += extractPattTree(st,lblFSet);
      } else {
        r += (nm:<sig, extractPattTree(st,lblFSet)>);
      }
    } catch e : println("getPatternTree exception: <e>");
  }

  return r;
}
```

The algorithm presented was used on the Rascal type checker to identify and extract the pattern list using the type checker as the metaprogram m. The total number of patterns in the metaprogram is 1137. Currently, the implementation only treats three type of patterns: `callOrTree`, `concrete`, and `descendant`, these are related to node, concrete, and descendant patterns respectively. Each pattern found by the function `extractPattTree` is stored in a map structure to generate the pattern result set. For this case, the algorithm identified 373 patterns and it represents 32.81% of the total. The time necessary to extract the pattern set was 59 seconds using the reference computer.

Function Name	Patt	UP
checkModule	51	17
calculatePatternType	37	27
checkStmt	30	12
bind	25	25
assignInitialPatternTypes	23	18
checkExp	17	13
fullyCheckSingleModule	10	10
checkSingleModulePastAliases	10	10
checkSingleModuleNamesOnly	10	10
loadItem	16	9
checkAssignment	16	2
loadConfiguration	12	9
resolveProduction	12	8
checkDeclaration	11	7
checkSingleModuleAndFinalize	8	8
getDeclarationNames	6	6
extractPatternTree	6	6
checkSingleModuleThroughAliases	6	6
buildAssignableTree	6	3
comparableOrNum	6	3
convertAndExpandThrowType	5	3
loadConfigurationTypes	4	4
computeAdditionType	4	2
loadConfigurationCons	3	3
concreteType	3	3
checkLiteral	3	3
check	3	3
checkConstructorKeywordParams	3	2
importProduction	2	2
directedConnectedComponents	2	2
checkSyntax	2	2
checkLocationLiteral	2	2
checkBooleanOpsWithMerging	2	2
unknownConstructorFailures	1	1
undefer	1	1
tooManyMatchesFailures	1	1
stripLabel	1	1
resolveSorts	1	1
loadTransType	1	1
loadTransSort	1	1
loadTransConstructor	1	1
inBooleanScope	1	1
getParameters	1	1
finalizeFunctionImport	1	1
convertAndExpandUserType	1	1
checkFunctionDeclaration	1	1
checkFormals	1	1
checkCatch	1	1
arityFailures	1	1
addNameWarning	1	1

Table 6.2: The list of function with patterns extracted from the RTC.

The list of functions which use pattern feature in the Rascal type checker module is presented in Table 6.2. The first column presents the function name, the second "Patt" shows the number of patterns instances for this function, and the "UP" column the number of unique pattern instance by each function. This list is sorted by the number of pattern instances for each function, and these instances are filtered using the pattern kind specified early. The unique patterns are those not identical to other patterns inside the same function. From 342 functions in the module, there are 50 using the filtered patterns and they are listed in the table.

A sample of the patterns extracted from the RTC module is presented in Table 6.3. The case study uses the processed pattern list produced in this step and the sample presented in the table is extracted from the processed list. The first column shows the name of the function where the pattern was extracted. The second column presents the pattern instance extracted from the metaprogram. This instance will be used by the next step in the pattern-based test process. The third column presents the pattern instance ID and the last shows the line number of the pattern instance in the metaprogram. The information collected and used during this case study was the function name, pattern's type, ID, and line number.

The pattern identification process instantiated in this case study produced a pattern list with the following information:

Function Name	Pattern	Pattern ID	Line Num.
checkStmt	(Label)‘<Name n> :‘	p113	4388
checkStmt	(Target)‘<Name n>‘	p120	4814
checkStmt	(DataTarget)‘<Name n>:‘	p124	5037
checkStmt	(LocalVariableDeclaration)‘<Declarator d>‘	p126	5097
checkStmt	(LocalVariableDeclaration)‘**dynamic** <Declarator d>‘	p127	5097
checkStmt	(Declarator)‘<Type t> <{Variable ","}+ vars>‘	p128	5098
checkStmt	(Variable)‘<Name n> = <Expression _ >‘	p129	5108
checkStmt	(Variable)‘<Name n>‘	p130	5108
checkStmt	(Variable)‘<Name _> = <Expression init>‘	p131	5109
checkDeclaration	(Variable)‘<Name n> = <Expression init>‘	p10	5868
checkDeclaration	(Variable)‘<Name n>‘	p11	5868
checkDeclaration	(Variable)‘<Name _> = <Expression init>‘	p14	5876

Table 6.3: A sample of the pattern set extracted from the RTC module.

- 373 is the number of patterns identified by the process;
- 32.81% of patterns from the total patterns in the RTC;
- 50 functions are related to these patterns;
- 59 seconds are spent to extract these patterns.

Pattern Processing

The pattern processing step is an instantiation related process that can be used to change or adapt the pattern list before its uses. In this case study, its was used to filter the patterns based on their kinds and to eliminate duplication. The number of patterns in the list can influence the test requirement and test data generation time. Consequently, the test case execution time also is influenced by that. This relation is detailed in the following sections.

The Processed Pattern List

The practical result of the pattern extraction process is the processed pattern list. This list is represented in the case study by a list of files that contains the patterns extracted from the pattern-based metaprogram. These files are organized by the function where the pattern instance was founded. As result, each function that contains pattern instances will have a file. The function name is used as the file name for each of these files.

Listing 6.2: An example of the content of the pattern processed list file for the function `buildAssignableTree`.

```
buildAssignableTree%p1%5221%(Expression)‘<DecimalIntegerLiteral dil>‘
buildAssignableTree%p2%5266%(OptionalExpression)‘<Expression eFirst>‘
buildAssignableTree%p3%5269%(OptionalExpression)‘<Expression eLast>‘
buildAssignableTree%p4%5300%(OptionalExpression)‘<Expression eFirst>‘
buildAssignableTree%p5%5306%(OptionalExpression)‘<Expression eLast>‘
```

The pattern processed list file for the function `buildAssignableTree` is presented on Listing 6.2. Each line of the file contains a pattern instance extracted from the metaprogram and another information about it. These information are separated by the character '%'. The first field in the line is the function name and the second is pattern ID. The third is line number in the metaprogram where of pattern instance was founded and the last is the pattern instance.

6.2.3 Test Requirements Generation

This section presents the step related to the generation of the test requirement set using the patterns identified on the previous step. This process is conducted using the description detailed on the Section 4.2 from Chapter 4. The current process uses the pattern set as input to it and from it generates the requirement set related to each pattern-based converge criteria defined in Chapter 5.

The use the original pattern set for all criteria can elevate the cost of the generation and execution related to the test cases. So, first we discuss the modification of the original pattern set to optimize it reducing its size. The use of unmodified pattern set can elevate the generation and test execution times because the number of patterns instances in a regular pattern-based metaprogram. To show this kind of behaviour, we present an analysis based on the RTC module and the number of test requirements associated to it when using the original pattern set.

Description	**Patterns**	**PCC**	**PECCC**	**PPWCC**	**APCCC**
Unique Patterns	163	163	326	52812	1.169×10^{49}
Filtered Patterns	373	373	746	277512	1.924×10^{112}
Total Patterns	1137	1137	2274	2583264	1.867×10^{342}

Table 6.4: The test requirement set size related to each pattern-based criteria to the RTC module.

Table 6.4 shows the number of test requirements necessary to satisfy each pattern-based coverage criteria for the RTC. The column "Patterns" presents the number of patterns for each situation, the "PCC" presents the number of test requirements for the pattern coverage, the column "ECPCC" presents the number of requirements for the each choice pattern coverage, the "PWPCC" shows the number for the pair-wise pattern coverage and the "ACPCC" the number for the all combinations pattern coverage. The test requirement set size was calculated using the equations presented in Section 5.1 of Chapter 5. By the results presented in the table, we can see that the number of requirements are high to the majority of cases when the total unfiltered patterns is used as reference.

An alternative to mitigate the problem is use only certain kind of patterns. The number of patterns applying this approach is shown in the line titled "Filtered Patterns" on the table. With this approach we can reduce the pattern set size significantly, from 1137 to 373 pattern in the set. The set size reduction has effect on the test requirements for all coverage criteria, as we can see in the same table line, but to the highest cost related criterion, the "APCCC" even this reduction is not enough. An additional restriction can be applied to the result pattern set to reduce its size even more: the elimination of duplicity on the set. Using this approach we can reduce the pattern set size to 163, as it can see on the line title "Unique Patterns". Using these restrictions over the patterns we can use the majority of the criteria without elevate too much the cost of the test process. In the case of all pattern combinations criterion even using these techniques the number of requirements still high, but can be improved by other techniques. This number is reasonable on this context because the complexity related to the RTC module.

To reduce the time related to the instantiation of the pattern-based test design process, we divided the pattern set using their source location. This division is based on the function where the pattern is coded in the metaprogram under test. The result of this process is a list of pattern set related to the functions coded on the metaprogram. With this approach we have more flexibility to choose which patterns will be used in the test process. Using it, we also can restrict the number of test requirement to use during the test process by select only some patterns related to a set of functions. On the present test design instantiation we used this strategy and select the most relevant functions to conduct the rest of process.

Function Name	**UP**	**PCC**	**PECCC**	**PPWCC**	**APCCC**
`calculatePatternType`	27	27	54	1,404	$134,217.728 \times 10^3$
`checkModule`	17	17	34	544	131.072×10^3
`checkStmt`	12	12	24	264	4.096×10^3
`buildAssignableTree`	3	3	6	12	8
Sum of functions	59	59	118	6,172	7.205×10^{16}
Unique patterns	163	163	326	52,812	1.169×10^{49}

Table 6.5: The test requirement set size for the selected functions based and the pattern-based criteria.

In Table 6.5 we summarize the main information about the unique patterns and test requirement set size for all coverage criteria by function. The last line of the table is a reference to the values obtained only using the initial optimizations. We select four functions based on the number of unique patterns to analyse in detail: `calculatePatternType`, `checkModule`, `checkStmt`, and `buildAssignableTree`. The first function calculate the pattern type related to a pattern expression, the second verify the type information for a module and the third verify the type information for a statement. As we can see on the table, the number of test requirements for each function is relative low, but the APCCC criterion still higher than the others. This elevate number of requirements to the APCCC is related to its rigor over the requirements so the cost of it will be higher on the majority of the cases. Even this criterion presenting the highest value among the other, its value compared to the value using the unique patterns is lower and possible of the generation and execution if the rigor justifies its cost.

Using these considerations into the test requirement set generation, this process starts using the pattern list with unique patterns and categorized by functions. The process is defined in Section 4.2 and we are using the `buildAssignableTree` as example during the presentation. This function has the pattern set $\mathcal{P}_m = \{p_1, p_2, p_3\}$ and the partition set for it is $Q = \{q_1, q_2, q_3\}$ each of them composed by two blocks. The block 1 represents the matches (p_i) and the block 2 the not match ($\bar{p_i}$). The patterns extracted from the function are listed into Table 6.6. The table presents the pattern label, pattern instance text, and line number source of the pattern.

Pattern Label	Pattern Instance	Line Number
p_1	`(Expression)‘<DecimalIntegerLiteral dil>‘`	5221
p_2	`(OptionalExpression)‘<Expression eFirst>‘`	5266
p_3	`(OptionalExpression)‘<Expression eLast>‘`	5269

Table 6.6: Patterns extracted from the function `buildAssignableTree`.

The test requirement set for the two initial coverage criteria are the $TR_{PCC} = \{p_1, p_2, p_3\}$ and the $TR_{ECPCC} = \{p_1, p_2, p_3, \bar{p_1}, \bar{p_2}, \bar{p_3}\}$. These sets are related to the pattern coverage (Definition 5.1) and pattern each choice (Definition 5.2) criteria respectively. The requirement set for the pair-wise pattern criterion (Definition 5.3) has 12 elements. Table 6.7 shows these elements and they were combined by the block grouping.

$$\begin{aligned} TR_{ACPCC} = \{ \;& p_1p_2p_3, \quad p_1p_2\bar{p_3}, \\ & p_1\bar{p_2}p_3, \quad p_1\bar{p_2}\bar{p_3}, \\ & \bar{p_1}p_2p_3, \quad \bar{p_1}p_2\bar{p_3}, \\ & \bar{p_1}\bar{p_2}p_3, \quad \bar{p_1}\bar{p_2}\bar{p_3}\}. \end{aligned}$$

For the all combinations criterion (Definition 5.4), the requirement set has 8 elements. These elements are defined using the three partitions from the set Q. The set definition for the TR_{ACPCC} presents its elements.

The blocks $[p_1, \bar{p_1}]$ and $[p_2, \bar{p_2}]$	
p_1p_2	$p_1\bar{p_2}$
$\bar{p_1}p_2$	$\bar{p_1}\bar{p_2}$
The blocks $[p_2, \bar{p_2}]$ and $[p_3, \bar{p_3}]$	
p_2p_3	$p_2\bar{p_3}$
$\bar{p_2}p_3$	$\bar{p_2}\bar{p_3}$
The blocks $[p_3, \bar{p_3}]$ and $[p_1, \bar{p_1}]$	
p_3p_1	$p_3\bar{p_1}$
$\bar{p_3}p_1$	$\bar{p_3}\bar{p_1}$

Table 6.7: The test requirement set for the pair-wise pattern coverage using the patterns from function `buildAssignableTree`.

The test requirement generation process detailed here using the function `buildAssignableTree` as an example is the same made with for the other functions chosen from the module RTC. The next step is the test data generation process, it uses the requirement sets produced here to generate the test data set to satisfy each of those requirement sets.

Listing 6.3: An example of the content of the test requirement file for the function `buildAssignableTree` and the pattern coverage criterion.

```
p1
p2
p3
p4
p5
```

The practical result of the process is the production of a list of files, each file contains the requirements for a specific function and coverage criterion. These files are used to produce the test data sets in the next step. For example, at the end of the process, the function `buildAssignableTree` will have a folder with the function name and this folder contains a file for each criterion. The files have the extension ".ts". Listing 6.3 presents an example of the contents for the function `buildAssignableTree` and the pattern coverage criterion. Each line of the file represents a test requirement.

6.2.4 Test Data Generation

The test data generation has the goal of generating a test data set that satisfies the test requirement set. This data set is presented in Definition 4.5 from Chapter 4. To generate the set it is necessary to know which sentences from the input language satisfy the requirements. So, each pattern requirement must be satisfied by at least a sentence and the union of these sentences will be the resulting test data set. A reference algorithm to generate the test data set was presented in Section 4.3.2 and we use it during this case study.

A central problem that the algorithm tries to solve is to discover a sentence that satisfies the operations of match or not match for a pattern associated to a requirement. This problem is defined in the algorithm definition (Definition 4.6) and the relation $\circ$ represents this problem in the algorithm. The step 2 of the algorithm looks for a symbol A that $A \circ t_i$ with t_i being a test requirement. This symbol A can generate many words and some of these words can be used to satisfy t_i. The step 3 constructs from the result word set a W set with sentences containing these words. Perhaps, this set W has some sentences that satisfy the requirement t_i and some that don't. The step 4 filters the set W, choosing only sentences that satisfy the requirement. All this process is defined abstractly and in the test design process must be instantiated.

On this case study instance, we used the reference algorithm to guide the implementation necessary to generate a test set that satisfies each pattern-based criteria. Our implementation is based on a sentence generator LGen [Hentz, 2010; Moreira et al., 2013]. The tool is used to generate sentences that match a pattern in the pattern set using the object grammar definition. This sentence generation process is an implementation of the step 3 of the reference algorithm. In our instantiation of the process, the chosen strategy was to divide the module in its functions and use the coverage criteria on each of them. Because of that, the context information about the function signature is used by the sentence generation process. This signature is used to extract the start symbol S defined on the reference algorithm. The relation $\circ$ is implemented by a Rascal pattern processing implementation that uses the pattern in the requirement to discover the nonterminal symbol A used by the LGen sentence generation process. The requirements that has a not match operation over the pattern needs special treatments because it cannot be directed generated by the LGen tool using the defined strategy. Because this characteristic, we adopt the strategy of use a unique negative case to satisfy all requirements for this kind in a function. For this case, a systematic manual generation was used to generate sentences for all requirements of this kind.

The test data generation instance used in this case study the following steps to produce a test data file used during the case study. To illustrate the process, the pattern instance `(Expression)‘<DecimalIntegerLiteral dil>‘` from `buildAssignableTree` function will be used as an example.

1. Extract the nonterminals from the pattern instances. The nonterminal symbols in the pattern instance are identified and extracted. This step is specific to each metaprogramming language environment and needs to be implemented during the instantiation. In our implementation to Rascal environment, this step is

made automatically using the processed pattern list and the metaprogram input grammar. In the example, the nonterminals identified are 'Expression' and 'DecimalIntegerLiteral'.

2. For each nonterminal, a string set is generated using the LGen tool. The example case we have a string set for the 'DecimalIntegerLiteral' symbol. This string set is produced using the metaprogram input grammar. Each symbol identified is used as start symbol to the generation process executed by the LGen. In this case study, each set of strings is limited to the size of 50 strings. This string set is used as a cache to provide string instance when a nonterminal symbol is used into more than one pattern instance. The number of strings generated into the set was chosen to reduce the generation time taken by the LGen and to have enough string instances in the cache to the posterior use. The const `0` is an example of string generate for the nonterminal 'DecimalIntegerLiteral'.

3. The strings produced in the last step are used to compose new strings using the pattern instances as a template. These new strings will be used to produce sentences that are used to test the metaprogram. Using the pattern example `(Expression)‘<DecimalIntegerLiteral dil>‘` and the string produced early for the 'DecimalIntegerLiteral' nonterminal, we replace the pattern variable `<DecimalIntegerLiteral dil>` by the string generated by the LGen. In this particular case, the new result string is `0`, that is the same of original string in this case but it is not always the case.

4. To produce a language sentence is necessary to complement the produced string with a valid prefix. In the case study, this process is also made using the LGen. The LGen is used to produce the string prefix by generation using the grammar start symbol until the pattern instance nonterminal. In the example, we used the Rascal start symbol 'Module' defined for the input grammar and the pattern nonterminal 'Expression'. The process of chose one string prefix from those generated by the LGen is manual in the case study. This process was conducted to achieve the compatibility between the pattern instance and the function signature where the instance was founded.

Listing 6.4: The Rascal type checker test set to the pattern coverage criterion for the function `checkDeclaration`.

```
//p3, Line 5875
public int a3 = 0+0;

//p5, Line 5876
public int a5 = 0+0;

//p6, Line 5996
data d6(int a=0);

//p4, Line 5875
public int a4;

//p1, Line 5868
public int a1 = 0+0;

//p7, Line 6039
data d7(int a=0) = d7();

//p2, Line 5868
public int a2;

//p8, Line 6061
data d8(int a = 0, int b = 0) = d8();
```

The function `checkDeclaration` is presented as example. This function has 8 patterns and its pattern set is $\mathcal{P}_{ck} = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8\}$. From these 8 patterns, we have 8 pattern coverage, 16 each choice, 112 pair-wise, and 256 all combinations requirements. Listing 6.4 presents the generated test data for the pattern coverage criterion. The comments shown over each test data line presents the pattern label and the line number on the `CheckTypes.rsc` module where the pattern was extracted. The presented test data set satisfies all requirements for the pattern coverage criterion. Here, it is possible to note that this test data is not minimal, as this feature was not aimed during the test data generation implementation.

Function	**Patterns**	**TR Sizes**	**TS Sizes**
checkFunctionDeclaration	1	(1,2,-,2)	(1,2,-,2)
buildAssignableTree	5	(5,10,40,32)	(3,4,9,9)
checkDeclaration	8	(8,16,112,256)	(8,9,37,256)
checkStmt	19	(19,38,684,524288)	(19,20,191,-)

Table 6.8: The Rascal type checker functions used during the case study.

In this case study, we used 5 function implementations from the Rascal type checker to apply the pattern test design process. Table 6.8 presents these functions and the results obtained by applying the strategy and implementation described. The table shows in the second column the number of concrete patterns in each function. The third column presents the number of test requirements for each function for the pattern, each choice, pair-wise, and all combinations criteria respectively. The last column shows the number of generated test cases in our experiment to satisfy each of these criteria, in the same order. The symbol '-' in the first and last line represents the not calculated values. The first case because of the pair-wise criterion needs at least two patterns and the second case because of the test set for the all combination criterion could not be generated by our reference machine.

6.2.5 Conclusion

The process described in the previous sections presents the main points related to the instantiation of the process defined in Chapter 4. Additionally, all material produced during this evaluation is available online.[6] We believe that the presented instantiation and the complete material available satisfy the evaluation goal 2.

The Research Question 1 is partially answered by the presented case study. This case study presents an example of a systematic approach using patterns to the testing of pattern-based metaprograms. The other part related to the quality is discussed in the next section.

As presented in this section, the cost associated with the instantiation process is high for the first time in a new metaprogramming environment. After this initial effort, the cost of the process is low because, as presented in the section, it is only related to the execution of the produced software with the new metaprogram project. With these considerations, the evaluation goal 1 is considered archived.

6.3 The Coverage Criteria Family Validation

Aiming the evaluation of the coverage criteria presented in Chapter 5, in this section we present a validation using these criteria. The goal of this validation is to answer the Research Questions 1, 2, 3, and 5, the first two related to the Hypothesis 1 and the last related to Hypothesis 2. This validation process is based on an empirical process to achieve its goal. The main theoretical base to this validation is the mutation testing technique. This technique presents an important set of characteristics to the validation of coverage criteria in general and it has been used in many research projects [Andrews et al., 2005]. As presented in Chapter 2, this technique basically consists in inject arti-

[6]https://github.com/chentz78/PatternTesting

ficial faults into the software under test and check the test set capacity to detect these faults. The evaluation goals 3 and 4 are used to guide this validation process.

6.3.1 Validation Planning

The validation method presented is based on the mutation technique. Basically, that consists in a test set generated to satisfy a coverage criteria, a software under test and a mutation operator set. The main advantage of using this technique is to quantify the quality variable associated to the pattern-based coverage criteria family. This quality variable defined to answer Research Question 3 in Chapter 1. Additionally, this strategy can help answering Research Question 5.

To conduct this validation the main control variables are the test case cost and quality associated to test sets that satisfy each pattern coverage criterion. The first variable, the cost, is defined as the size of the test set, the number of test cases in it. The second variable is the quality of these test sets in the test of pattern-based metaprograms. This variable is associated to the capacity of a test set to detect injected faults. These faults are injected by the mutation operators used. The variable was defined as the mutation score obtained by a test set during this validation process. Consequently, we have some results related to the efficacy for the test set been provided by the quality variable and other results associated to the efficiency been provided by the cost variable. To help the understanding of the relation between these two variables and compare their results, we added a rate variable "Mutation Score / Test Cases". This variable represents the score per test case of each result obtained for each different scenario.

Figure 6.1 presents the flowchart to the coverage criteria validation process. In the flowchart notation used, the circles represent the process and the rectangles represent data. It illustrates an overview of the used process, and presents its steps, requirements, and results. This process has as requirement a test set, a software under test, and a mutation operator set. At the beginning of this process these requirements must be instantiated to be used during the execution. The mutation score is the result of this process execution.

The validation process begins by the execution of the software under test(SUT). This execution uses the test set as input and this test set is associated to the coverage criterion under validation. If the SUT pass on the execution of the test set, the flow goes to the next step: the mutation process. If the SUT does not pass, the validation process is stopped. The mutation step uses the mutation operator set to generate mutants from the SUT. After mutants generation, it is necessary to identify those that are equivalent to the SUT. This step was conducted manually during this validation and it was made using code analysis and run-time execution results comparison. The equivalent mu-

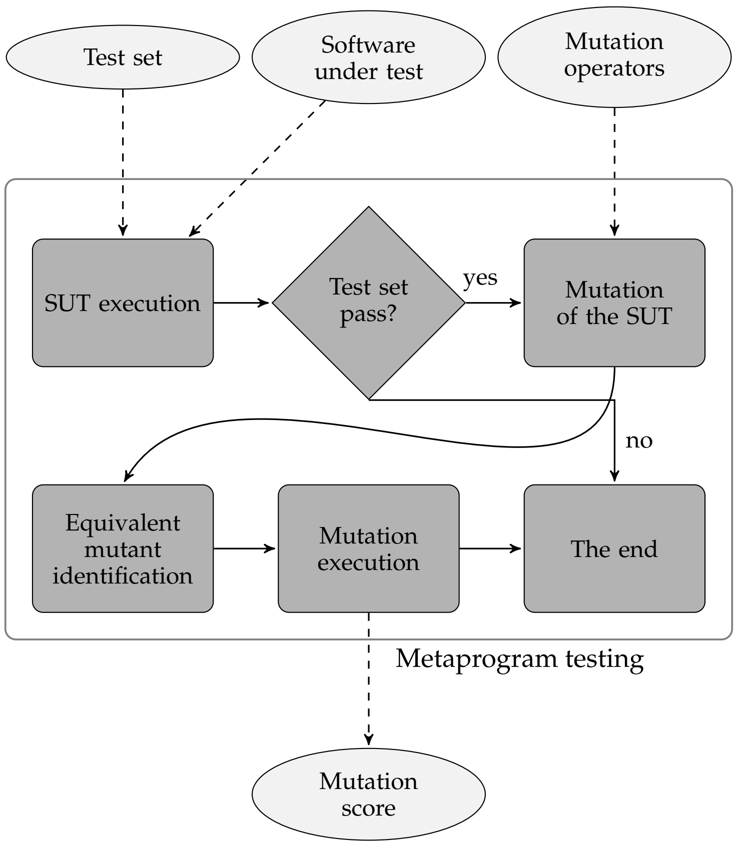

Figure 6.1: Process flow for the coverage criteria family validation process.

tants are discarded and they are not used in the next steps of the process. The next step is the mutants execution using the input test set. At the end of this step, we have the mutation score calculated.

On the mutants execution step, the concrete test cases used are adapted from the test data generated from the coverage criteria and their expected results are obtained from the SUT execution. These results are obtained from the manual SUT code analysis and the result of its execution without using an test oracle. Our main goal here is the validation of the coverage criteria and not the SUT. Because of that, we believe that we can justify the use of this strategy without definition or a detailed discussion about the test oracle.

The first effort to conduct this evaluation started by a validation of the pattern coverage criteria applied to a simple metaprogram that calculates the cyclomatic complexity, the example presented in Listing 3.8. The initial result obtained from this initial evaluation using only the pattern criterion was presented in the paper [Hentz et al., 2015]. A complete version using all criteria proposed and the same metaprogram is presented in the following section. After that, the results obtained from the Rascal type checker are presented.

6.3.2 Cyclomatic Complexity Case

Aiming to collect more information about the quality and cost of the pattern-based coverage criteria, we conduct the same experiment using another metaprogram. This program is a Rascal implementation of the cyclomatic complexity algorithm, and it was presented in Listing 3.8 into Section 3.1. The experiment uses two implementations of the algorithm one for Java, the one presented in the listing, and another one for Pico [Deursen et al., 1996]. In both situation we used the LGen to generate the test set, it using the same principles defined in the Rascal type checker case. The coverage criterion used in these cases is also the production coverage criterion. The grammar used in the process for this case is translated from Rascal library [7] The test set generated to the Pico case has 10 test cases, and it achieves 100% of production coverage.

Structure	**Size**	
	Pico	**Java**
Production Rules	18	272
Terminals	23	98
Nonterminals	9	142

Table 6.9: Grammar statistics for Pico and Java language used in the cyclomatic complexity experiment.

The grammar used for the Java case was translated from the ANTLR grammar [8]. In this case, the reached coverage level was 80% obtained with 85 sentences in the test set. Table 6.9 presents the size information about the two grammars used in the experiment.

The process used in this experiment follows the same flow defined in Figure 6.1. Perhaps, the metaprogram used here is different, and consequently, a new set of mutants were generated for it. The number of mutants generated to the Pico CC implementation was five with two equivalent and three resulting mutants. The Java case, we have 22 mutants with eight equivalent and 14 resulting mutants.

Listing 6.5: The central function of the Pico CC implementation.

```
public int calcCC(Statement impl) {
  int result = 0;
  visit (impl) {
    case \ifElseStat(_,_,_) : result += 1; //P1
    case \whileStat(_,_) : result += 1;  //P2
  }
  return result;
}
```

[7]Pico grammar defined in Rascal : https://git.io/vFVP8

[8]ANTLR Java grammar : https://git.io/vFVPk

The function `calcCC` implements the core cyclomatic complexity implementation to the Pico language. To illustrate the mutation process used in this case during the validation, the function implementation is presented on Listing 6.5. The function uses the visitor command, presented on line 3 to 6, to treat each decision structure existent in the input program. An example of mutation generated from the original function is presented on Listing 6.6. Line 5 presents the resultant code of the pattern with action replacement (PWA) mutation operator. This operator replaces an action part of the visitor command with an exception statement.

Listing 6.6: A mutation of the function `calcCC` using the operator PWA during the validation process.

```
1 public int calcCC(Statement impl) {
2   int result = 0;
3   visit (impl) {
4     case \ifElseStat(_,_,_) : result += 1; //P1
5     case \whileStat(_,_) : throw "mutant! PWA: Remove action from
         pattern and action.";  //P2
6   }
7   return result;
8 }
```

Listing 6.7 presents the pattern coverage criteria test data set for the Pico cyclomatic complexity case. Using this test data set, the mutation presented on Listing 6.6 is killed during the mutation process.

Listing 6.7: The test data set for the Pico CC metaprogram using the pattern coverage criterion.

```
begin declare id0 : natural ; if id0 then id0 := id0 else id0 := id0 fi
    end
begin declare id0 : natural ; while id0 do od end
```

Table 6.10 presents the results obtained in the Pico case.The results obtained from the production coverage is presented in the last line. Observing the table, we see low number test cases for the pattern-based criteria. This result is justifiable because the CC implementation only treats a small part of the input grammar. Additionally, the metaprogram is mostly implemented using patterns. Consequently, this fact justifies the mutation score/test cases ratio for the production coverage test set.

The results obtained from the Java case are presented in Table 6.11. Concerning the generation test set size, we can observe that the sets from the pattern and each

Coverage Criterion	Test Set Size	Killed Mutants	Mutation Score	M. Score / Test Cases
Pattern	2	3	100%	50
Each Choice	2	3	100%	50
Pair-wise	4	3	100%	25
All Combinations	4	3	100%	25
Productions	10	3	100%	10

Table 6.10: Results obtained to the Pico CC implementation.

choice criteria have the lowest number of cases. Besides this fact, these two criteria sustained the quality measure by the mutation score. Looking the mutation score/test cases rate, we can perceive a better result obtained by the pattern coverage criterion. The all combinations criterion was not used in this cases because of its high number of test requirements, 65536 in this specific case. This value is justifiable by the generation of the test set be a manual process and the high number of test cases. Because this, the all combinations criterion results is not shown in the table. It is important to note that the mutation score to the CC implementation is high because of the characteristic of this metaprogram which has the majority of its implementation using pattern.

Coverage Criterion	Test Set Size	Killed Mutants	Mutation Score	M. Score / Test Cases
Pattern	16	13	92.9%	5.8
Each Choice	17	13	92.9%	5.5
Pair-wise	137	13	92.9%	0.7
Productions	85	4	28.6%	0.3

Table 6.11: Results obtained from the Java CC implementation.

6.3.3 Rascal Type Checker Case

Based on the structure and the techniques presented, this section presents the results obtained by the execution of the Rascal type checker using the test sets generated from some of functions of the module. These functions are `checkFunctionDeclaration`, `buildAssignableTree`, `checkDeclaration`, and `checkStmt`. These results are organized by function and for each of them the pattern coverage criteria information is presented.

Listing 6.8: Sample of code from function `checkDeclaration`.

```
for (v <- vars, v@\loc notin c.definitions<1>, v@\loc notin {l | error(_
    ,l) <- c.messages}) {
  if ((Variable)`<Name n> = <Expression init>` := v || (Variable)`<Name
     n>` := v) {
    RName rn = convertName(n);
    c = addTopLevelVariable(c, rn, false, getVis(vis), v@\loc, rt);
  }
}
```

To illustrate the mutation process used in this case during the validation, a sample of function `checkDeclaration` is presented on Listing 6.8. The example shows the original code of the function and the code after the mutation process. This code sample checks the variable declaration in the Rascal code. Listing 6.9 presents the mutation of that code using the mutation operator bomb statement replacement (BSR). In this mutation, the commands associated with the first condition of `if` command are replaced by a bomb statement.

Listing 6.9: A mutation of the function `checkDeclaration` using the operator BSR during the validation process.

```
for (v <- vars, v@\loc notin c.definitions<1>, v@\loc notin {l | error(_
    ,l) <- c.messages}) {
  if ((Variable)`<Name n> = <Expression init>` := v || (Variable)`<Name
      n>` := v)
    throw "mutant! BSR1: Remove if conditionals. Case 1.";
}
```

For the example presented before, the test data generation process produced the test data set shown on Listing 6.10. This set was produced using the pattern coverage criterion in the context of the `checkDeclaration` function. This test data set example kills the mutant presented in the Listing 6.9.

Listing 6.10: The Rascal type checker test set to the pattern coverage criterion for the function `checkDeclaration`.

```
public int a3 = 0+0;

public int a5 = 0+0;

data d6(int a=0);

public int a4;

public int a1 = 0+0;

data d7(int a=0) = d7();

public int a2;

data d8(int a = 0, int b = 0) = d8();
```

Coverage Criterion	Test Set Size	Killed Mutants	Mutation Score	M. Score / Test Cases
Pattern	1	4	21.05%	21.05
Each Choice = Pair-wise = All Combinations	2	4	21.05%	10.53

Table 6.12: Results obtained to the function `checkFunctionDeclaration`.

Now we present the results produced during the validation for the RTC case. The first data presented is related to the function `checkFunctionDeclaration`. Table 6.12 shows the results obtained by running each test set related to the criteria. The first column presents the criterion name, the second the number of test cases in the test set, the third presents the number of killed mutants, the fourth presents the mutation score for it, and the last column presents the rate of mutation score by a test case. The each choice, pair-wise, and all combinations coverage criteria have the same results for this specific function case.

Coverage Criterion	Test Set Size	Killed Mutants	Mutation Score	M. Score / Test Cases
Pattern	3	11	31.43%	10.48
Each Choice	4	11	31.43%	7.86
Pair-wise	9	11	31.43%	3.49
All Combinations	9	11	31.43%	3.49

Table 6.13: Results obtained to the function `buildAssignableTree`.

Table 6.13 presents the results obtained by the execution of the mutation process for the function "`buildAssignableTree`". This function is responsible to extract a tree representation of the assignable instructions and perform basic checks over it.

Coverage Criterion	Test Set Size	Killed Mutants	Mutation Score	M. Score / Test Cases
Pattern	8	17	60.71%	7.59
Each Choice	9	17	60.71%	6.75
Pair-wise	37	17	60.71%	1.64
All Combinations	256	17	60.71%	0.24

Table 6.14: Results obtained to the function `checkDeclaration`.

Table 6.14 presents the results obtained by the execution of the mutation process for the function "`checkDeclaration`". This function is responsible to check the module declarations in Rascal.

Coverage Criterion	Test Set Size	Killed Mutants	Mutation Score	M. Score / Test Cases
Pattern	19	41	30.15%	1.59
Each Choice	20	41	30.15%	1.51
Pair-wise	191	41	30.15%	0.16
*All Combinations**	524288	-	-	-

Table 6.15: Results obtained to the function `checkStmt`.

Table 6.15 presents the results obtained by the execution of the mutation process for the function "`checkStmt`". This function is responsible for checking the type information of Rascal statements. The all combinations criterion was not used to generate a test set because of its elevated number of test cases. Because this, the table line with the information about this coverage criterion was suppressed. Based on the test requirement number for this criterion, the number of test cases for it will be 524288. Consequently, the generation process for that test set takes more the four hours without finish the process. Associated to this fact, the execution time for this test set will be too high.

In each result presented for the selected functions, the number of killed mutants is the same for each coverage criterion presented. For example, in the function "`buildAssignableTree`" the number of killed mutants are 11 (Table 6.13) for all criteria. This value repeats because of a characteristic of the metaprogram under test for the analyzed functions. The Rascal type checker module is implemented using the patterns in a specific way, the patterns are used to parse specific constructions in the input program. During the type checking process, an external information is necessary to be used in conjunction with the patterns to execute some parts of the program. Because of the described characteristics, the use of different pattern-based criterion during this evaluation does not present influence over the number of killed mutations.

Listing 6.11: A sample from the function `buildAssignableTree`.

```
if (isTupleType(atree@atype) && isIntType(tsub)) {
  if ((Expression)`<DecimalIntegerLiteral dil>` := sub) {
    tupleIndex = toInt("<dil>");
    if (tupleIndex < 0 || tupleIndex >= size(getTupleFields(atree@atype))) {
      failtype = makeFailType("Tuple index must be between 0 and <size(getTupleFields(atree@atype))-1>",sub@\loc);
      return < c, subscriptNode(atree,tsub)[@atype=failtype][@at=assn@\loc] >;
    } else {
      return < c,subscriptNode(atree,tsub)[@atype=getTupleFields(atree@atype)[tupleIndex]][@at=assn@\loc][@literalIndex=tupleIndex] >;
    }
  } else {
    return < c, subscriptNode(atree,tsub)[@atype=Symbol::\value()][@at=assn@\loc] >;
  }
}
```

To illustrate the metaprogram characteristic, Listing 6.11 presents a sample extracted from the function "`buildAssignableTree`", for example. This sample is composed of a conditional command `if` coded on line 5246. This `if` command is based on the evaluation of the variables `atree` and `tsub`. Inside this command on line 5247, there is another `if` based on the pattern `(Expression)‘<DecimalIntegerLiteral dil>‘` and the function parameter `sub`. This pattern was used during the validation process and has a test case to cover it. But this test case only can execute the line of code where the pattern is located when the variables `atree` and `tsub` have a specific configuration. In this case, these variables are associated with a specific pattern and they are not controlled by the validation process. The internal `if` command on line 5247 cannot be executed without a specific valuation for those variables. This characteristic illustrated by the presented example is recurrent in the other functions analyzed. Because of this, the number of killed mutation in those functions' tables are also the same.

The problem of the association of an input data with the determined code execution is called *recheability* [Ammann and Offutt, 2008] and it is a complex test problem related to the automatic test data generation. As result, all mutations related to the pattern-based `if`, in this presented case, are not killed by any test case from any coverage criterion. It is important to note that the coverage criteria and test design process do not take in consideration during the requirements and the test data generation the reachability problem. As result of this, the situation described early can happen during the test process. In some cases, this reachability problem can mitigate the quality of the generated test set.

The example presented in Listing 6.11 has many `if` commands, but only one command with a pattern associated with it. With this kind of pattern usage, the use of more elaborated pattern-based criteria is not justifiable. The presence of only one pattern and the combination of this pattern with itself does not make sense. Another situation is when there are many patterns in a function implementation but these patterns do not produce a change in the metaprogram execution when combined. This last case is observed in some of the other functions analyzed. Because all functions analyzed present one of these situations, the data results presented in Tables 6.12, 6.13, 6.14, and 6.15 can be justified.

Grammar-Based Testing

The grammar-based testing strategy is used to generate a test set and execute it using the same mutation strategy explained previously. The goal of this process is to obtain data about this test set and compare with those obtained from the pattern-base criteria. The test set was generated using the LGen tool and using the Rascal grammar as input. The grammar was translated from the Rascal grammar definition module, "`Rascal.rsc`" [9], to the LGen notation. Table 6.16 presents the basic information about the Rascal grammar, this information gives a notion about the size of the grammar. The coverage criteria used during this process was production coverage criterion, and it is available in the tool. Because the grammar's size and its complexity, the tool generates 130 sentences with 41.48% of production coverage. The generated test set was adapted to replace the lexical symbols that are not generated by LGen, it because the LGen does not support the generation of lexical symbols.

Structure	**Size**
Production Rules	518
Terminals	180
Nonterminals	131

Table 6.16: Grammar statistics for Rascal language used in the evaluation.

To evaluate the capacity of the grammar-based test set to be used with the Rascal typer checker module, we used the Rascal parser and type checker to verify the sentences in the test set. With this process, we verified that 121 sentences from the grammar-based test set are parsable. The nine sentences are not parsable because the Rascal grammar and parser have support for more verification constructions than the LGen input grammar notation. An example of these verification constructions is the declaration and detection of ambiguities (LGen assumes that the input grammar is not ambiguous.). During the verification using the type checker, none of 121 parsable sentences was checked. This result was expected because the LGen tool only generates sentences using the syntax definition provided by the context-free input grammar and it does not provide direct support to more advanced constructions.

To collect information to answer the evaluation goal 4, we need to use this grammar-based test set in the mutation process. Because of it, we executed the mutation process using this test set and with the same environment used for the pattern-based test sets. The execution using this test set did not kill any mutants, and its mutation score is 0%. This result is justifiable because the grammar-based technique uses the syntax definition of the input language and it does not include the semantic restrictions over the language. Because of this, the type checker detected all invalid sentences generated by the LGen tool.

In this case, the test set generated by the grammar-based technique can be used to represent the invalid sentences that can be used during the test of the type checker. Using this approach, the test set can be a complementation for the test set generated by the pattern-based technique. As result, there is a gain in quality when is used the grammar-based and pattern-based test sets in conjunction. This gain is the capacity of

[9]The module version used on this work was `4e9cd4e` and it is available on `https://git.io/vQs3a`.

the result test process to cover invalid situations for the pattern-based metaprogram under test.

6.3.4 Conclusion

To help the discussion about the presented results, we summarized the information previously shown using the average of the value obtained on each function. Table 6.17 presents these results. The table is formatted using the same column structure of the previous section, and each of his values is calculated from the results of the selected functions. The average pattern instances used in the functions is 8.25.

Coverage Criterion	**Test Set Size**	**Killed Mutants**	**Mutation Score**	**M. Score / Test Cases**
Pattern	7.75	18.25	35.84%	10.18
Each Choice	8.75	18.25	35.84%	6.66
Pair-wise	59.75	18.25	35.84%	3.96
All Combinations	*131138.75*	*10.67*	*37.73%*	*4.75*

Table 6.17: The summarized results using the average calculation obtained from the selected functions.

The test set size information presented in the table represents the cost obtained from each coverage criterion executed. The pattern coverage requires a small test set size to be satisfied. This reduced size is a consequence of few requirements and it is proportional to number of patterns to cover. Each choice criterion has a test set size close to the pattern criterion which also represents few test cases in average. This value is close because the strategy used to generate the negative cases is based on a unique negative case that does not match with any pattern in the requirement set.

The pair-wise test set is larger than those two firsts criteria because the number of requirements for this criterion is also higher. The number of tests in the set is not close to the number of requirements because each test usually satisfies many requirements, eliminating of some duplicity during the test data generation. The test size for all combinations is always the number of requirements. The result value for this case is not higher because when the number of requirements was high in the function "`checkStmt`" case, then the test set was not generated.

Based on the given cost definition, the results present a low cost to the pattern and each choice criteria. In the pair-wise case, this cost increased but without a gain on the mutation score. Finally, the all combinations criterion presents the highest cost. With this information, we can add more evidence to construct an affirmative answer to the research question 2.

The mutation score information presented in the table is interpreted as the quality variable associated with a coverage criterion. The pattern and each choice criteria present the same mutation score. The all combinations differ from others because it does not have the killed mutation information for all used functions.

During the evaluation, we used one test set from the Rascal implementation to compare its results with those obtained from the pattern-based sets. This module, "`DataTypeTCTests.rsc`" [10], has 44 test cases and 49 SLOC. Using the same evaluation environment, this module killed 6 of all 218 mutants generated during the process or the mutation score of 2.75%. It is important to note that the module was not developed to test only the selected functions, and it is not used as a reference to compare with our results but can provide us an insight about the relative cost and quality of the pattern-based test sets.

At this point, the quality variable can be analyzed using the presented results. The pattern coverage criteria present an improvement over the tested metaprogram. Although the resulting mutation coverage is not as good as the one obtained in the cyclomatic complexity example, it is better than the coverage obtained with the grammar-based test set. This result can partly be justified by the quality of the test data generation used during the validation and partly by the characteristic of the selected functions. Using the data presented obtained using the pattern coverage criterion, we can conclude that the metaprogram under test can be improved in relation to its quality using the criterion. Because of this improvement, we can answer completely the Research Question 1. Aditionally, the Research Question 3 is answered as acceptable, but the quality will be related to the quality of the test data used in the test process. This test data quality can probably be improved to achieve better results.

Coverage Criterion	**Test Set Size**	**Killed Mutants**	**Mutation Score**	**M. Score / Test Cases**
Pattern	31	73	33.49%	1.08
Each Choice	35	73	33.49%	0.96
Pair-wise	239	73	33.49%	0.14
All Combinations	*524555*	*32*	*14.68%*	*0.00*

Table 6.18: The global results using the sum calculation from the selected functions.

Table 6.18 presents the global results obtained from each function and calculated using a summed version for each criterion result. This table uses the same structure presented earlier and gives a global view of the numbers obtained with each criterion. We can observe that the test set size is increasing from the pattern to all combinations criteria, it is reasonable because they have a different level of rigor. This same increment can be observed in the CC case. It is important to note that the increase rate in the test set size column may be influenced by the test case generation process. Additionally, the test set size and killed mutants for the all combinations criterion do

[10]The "`DataTypeTCTests.rsc`" test module represents 13.05% of the entire RTC test set and it is available on `https://git.io/vHPCE`.

not present a conclusive information based on the evaluation conducted. This situation happens because the test set for the function "`checkStmt`" can not be generated and, consequently, the values for this particular criterion are lower than the expected ones.

As defined in the evaluation methodology, we defined two evaluation goals related to the results obtained from the use of the pattern-based coverage criteria using the mutation technique as validation. During this validation, we used two metaprograms as case studies: the Rascal Type Checker and the Cyclomatic Complexity. With the results presented in Tables 6.17, 6.18, 6.10, and 6.11, we consider the evaluation goals 3 and 4 achieved.

Based on the results obtained during the evaluation, the pattern coverage criterion is the only criterion recommended to be used in the testing of pattern-based metaprograms that have the same characteristics of the present case studies. This recommendation is based on its mutation score and test case relation that presents the best results for all functions analyzed in the evaluation. During the evaluation, we only used one type of pattern for the Rascal language. This fact also contributes to the recommendation over the use of the pattern criterion. The evaluation process presented and the metaprograms analyzed do not provide a conclusive evidence of the advantage on the use more elaborated patterns criteria instead of the pattern coverage criterion.

6.4 Threats to the Validity of the Experiments

The main threat to the validity of the presented validation is the number of functions used to conduct the validation. Besides the use of a real and complex pattern-based metaprogram, the set of functions used should be close to all functions with the pattern instances to provide a broader and a more detailed information about the use of the proposed criteria. Another point that can influence the presented results is the type of pattern instances used during the validation. In this case, the resulting mutation scores could have been improved because with a larger variety of pattern types it is more probable that the test case can detect the mutations.

The mutation operator set used in the validation consisted of some traditional operators and some specifically designed for the mutation of pattern based metaprograms. This combination, however, was not itself validated in the context of simulation of common faults in pattern-based metaprograms. This limitation can contribute to imprecise results. This problem can be treated by validating and improving the mutation operator set used during the validation. Another improvement in this direction is the use of a source code coverage information collected during the validation process. This information can contribute to a more realistic knowledge about the code execution and, consequently, the quality of the test set used in the process.

The test data generation process used during the case study influences the cost and quality of the results. This characteristic can be improved by using a more elaborate process to generate the test data set. The current version only uses the syntax information from the pattern and the function signature to generate these data, and it can be improved by using another information, like the pattern instance context, on the process.

Another important consideration is the use of a unique test set representing the criterion to validate it. The generation of the test data to construct the test set is out of the scope of this project, but this process influences the results obtained by a criterion during the validation. An alternative to mitigate this problem is to use a generation algorithm that always generates a minimal set [11]. Without using a minimal test set can be difficult to obtain conclusive information about the cost and quality precisely. However, generating a minimal test set for a coverage criterion is not trivial, and in the case of grammar-based testing this problem is not treated as well [Kossatchev and Posypkin, 2005; Purdom, 1972; Lämmel, 2001].

[11]The minimal test set is a set with the lowest number of elements necessary to satisfy a coverage criterion.

— 7 —

Related Work

This chapter presents a comparison and discussion of some research and tools that can be related to the current work. It is important to note that research dealing with metaprogram testing is relatively new, with some particular exceptions, such as development (and testing) of compilers. So, we could not find research works aiming the test of these metaprograms directly. We present in this section some related research works that can be applied to the testing of metaprograms with some modification or adaptation.

The presented works in this chapter are divided by the approach used in the testing process: Grammar-Based testing 7.1, Bounded-Exhaustive Testing 7.2 and Concolic Testing 7.3.

7.1 Grammar-Based Testing

Grammar-based testing has been used in compiler testing for many years [Boujarwah and Saleh, 1997; Kossatchev and Posypkin, 2005]. Because compilers are a specific kind of metaprograms, other kinds of metaprograms could be tested using similar techniques. Based on these premises, we used as a comparison basis these testing techniques because they are consolidated and present positive results in many compiler applications.

The first tool that we can compare is the YouGen [Hoffman et al., 2011]. This tool generates test data based on a context-free grammar definition. The important features of the tool are the restriction over the generation process based on the grammar non-terminal coverage and the control over the depth of derivation trees based on annotations. Besides the power of these features, as presented in the original work, the tool does not provide or discuss the support for different coverage criteria. Consequently, the use of the tool only provides the support to the production coverage criterion. The main problem to apply directly YouGen to the pattern-based testing of metaprograms is that it relies on the filters using the grammar information. Consequently, this technique only reduces the test set generated by the tool but with the same cost related to the generation process using all grammar information. So, the pattern information related to the metaprogram under test is not used. Using the proposed approach is possible to reach a more efficiently generation process because not all grammar information like productions is used, but only those related to the patterns associated with the metaprogram.

A similar tool based on the context-free grammars is the LGen [Hentz, 2010; Moreira et al., 2013]. The tool is based in the language enumeration using the input grammar and imposing restrictions over the process. This tool has a similar strategy as YouGen, but with two main differences: the tool provides a set of grammar-based coverage criteria, and it allows the implementation of new coverage criteria. The implementation of new criteria can be done using a framework provided by the tool. Consequently, the implementation task is relatively simple. The tool provides the traditional grammar-based coverage criteria implemented using this framework, and these implementations can be used as a reference to implement a criterion. As in the case of YouGen, the language enumeration algorithm makes the cost of the generation process using the patterns is high. Additionally, the size of the tool generated test set is bigger, in general, comparing to those generated using the pattern-based coverage criteria. It can be justified by the enumeration and filter strategy and the not use of the pattern information from the metaprogram under test. Consequently, the cost associated with the execution of the resulting test set is also higher.

Besides the limitations presented by the LGen tool, we used it to generate parts of the test data during the case study. The strategy was to use the tool to generate these parts test data for a pattern based on the types, nonterminals in the tool input grammar, declared on it. These types were used as start symbol to the input grammar and the sentence generated saved to a posterior composition to form complete test data. This adaptation was made with the tool as concept proof to the use the LGen with the pattern information. We did this process with the LGen because it was developed by us, consequently, the support and implementation are faster and simpler than with the other tools. Additionally, we planning to add support to the pattern-based coverage criteria to the LGen tool, using the lessons learned during this project, in a future project.

In the same direction, the XTextGen [Härtel et al., 2014] generates test-data based from grammars. This tool has a different approach to generate the data. It uses Cantor pairing and mandatory multiplicity control. With these approach, the generation process is split into two phases. First, an enumeration process and then a semantics-directed post-processing over the result a set of transformation. With XTextGen it should be possible, in principle, to generate the test data that satisfies the pattern-based coverage criteria. An appropriate encoding would have to be developed to do so, which would amount to an alternative implementation of our sentence generation implementation.

7.2 Bounded-Exhaustive Testing

Bounded-exhaustive testing [Khurshid, 2004] is an automated approach that exhaustively testing all inputs within a given bound. The main idea behind this strategy is that many faults can be revealed within small bounds and it covers all “corner cases" within this bound. The testing of metaprograms can also use the bounded-exhaustive testing to generate the input programs to test these metaprograms. Because that, we present here a set of research efforts that can be related to the test of metaprograms and compare they with the pattern-based approach.

The first work related to the bounded-exhaustive testing is the ASTGen [Daniel et al., 2007]. The ASTGen is an imperative, iterative, composable and generic test data generation framework used to create abstract syntax trees(AST). It uses bounded-exhaustive approach for generating test programs within a given bound. These programs are generated in AST format and the framework can generate a complex structured test program. The ASTGen uses an imperative generator strategy to define how the programs are generated. It can be used to test applications that take programs as input in general, but the main results presented are related to the Java language and refactoring engines.

Jagannath et al. [Jagannath et al., 2009] propose several ways of reducing the cost of bounded-exhaustive testing by sparse test generation and structural test merging. The sparse test generation consists in skip some tests cases to reduce the time to the first failing test case. The structural test merging smaller inputs into fewer bigger inputs instead a larger number of smaller inputs. This is done to achieve a reduction in the test case generation and execution time. The authors also proposed an oracle-based test clustering technique to reduce the inspection time over the results produced by the process. As results, the authors obtained a cost reduction from the bounded-exhaustive testing of the Eclipse refactoring engine while preserve the original fault-detection capability.

The SafeRefactor [Soares, 2010] is a tool that aiming to make the refactoring process safer. This tool is based on testing to verify a refactoring process in the Java code. The tool *JDolly* is used to automatic generate the Java programs that is used during the refactoring verification. This tool uses the ASTGen framework in the program creation process. An important feature of the tool is it generates the programs using constraints received as parameters, these constraints can be specified using the Alloy language[Jackson, 2002]. This specification is used during the generation to find inputs that satisfy the input constraints. As result, the tool uses the bounded-exhaustive testing approach to generate all input programs for a given bound. The JDolly tool was used later to generate java programs [Soares et al., 2013]. A new tool *CDolly* [Mongiovi et al., 2014] using the same JDolly strategy to generate C programs. A new tool composed from JDolly and CDolly is defined as *Dolly* with support to the both languages.

UDITA [Gligoric et al., 2010] is a language extension of Java with support to non-deterministic choices. This tool can also generate Java programs based on constraints that are resolved by the *Java Path Finder* model checker and it implements a bounded-exhaustive testing. The extension provides two main features: a new language for describing tests and a new test generation algorithm. The language adds to Java the support to non-deterministic choice commands and an object pool abstraction. This pool allows the tester to control generation of linked structures, including trees, DAGs, cyclic graphs, and domain-specific data structures. The new test generation algorithm presents efficient techniques for test generation by systematic execution of non-deterministic programs. These techniques are based on a systematic exploration performed by explicit-state model checkers to obtain the effect of bounded-exhaustive testing.

These presented tools based on the bounded-exhaustive testing can be used in conjunction to these tools. First, the pattern test design is not based on a specific technology. Consequently, during the instantiation phase of the process these tools could be used to generate the test sets that satisfy the coverage criteria proposed on this work. The main challenge to do this process is the support for these test tools to support the generation for different languages and even for different programming language paradigms, for example the case of the Rascal language.

The second important aspect is the model checker implementation with support to different languages and specially with support to a pattern related theory. The first point could be difficult to resolve for the UDITA tool because its use of the Java Path Finder model checker. This because this model checker is based on the Java bytecode execution by a *Java Virtual Machine*. The Dolly tool can be used if there is an extension to support the object language used by the metaprogram under test. For this tool, the model checker Alloy is not a language related and there is no language restriction associated to it. Perhaps, the pattern base theory must be translated to the Alloy specification. If there is this kind of translation, the restrictions defined by the proposed pattern-based criteria can be specified and the Dolly tool can be used. The ASTGen does not use a model checker to apply constraints to its generation process. Because it, the only concern is the support to multi-languages by the tool.

Our proposed implementation does not use the bounded-exhaustive strategy.

Because it, the generation process constructs the test data/program directly from the patterns extracted from the metaprogram under test or other artifact. This simplify the generation process and consequently reduce the cost of it. It is important to note that the case of negative pattern requirements(those that defines that the pattern does no match) necessary to a criterion is more complex to be solved by a direct construct test data from the pattern requirement.

7.3 Concolic Testing

With concolic testing [Sen et al., 2005] a similar effect of selecting the right test cases could be achieved as our approach can. Basically, the concolic testing is based on symbolically simulating a program for a given test input and using a SMT solver or theorem prover to generate the next input which will cover a different execution path than the previous test input did for a program under test. Despite it will be possible, in theory, to apply the same technique to the metaprograms, it is necessary to have support by the SMT Solver to grammar and pattern matching to it and implement a simulation engine for the metalanguage used for this approach reach similar results. Otherwise, our generation approach simpler and lightweight. This because it is not necessary to have an efficient SMT Solver and neither the simulation engine. In our approach, presented in the instantiation shown in Chapter 6, just needs just a pattern extraction tool and a sentence generator.

— 8 —

Conclusions

In this thesis, we aiming contribute to the testing process of the metaprograms. As shown, these programs have some particular characteristics that create new challenges to their testing. The pattern matching mechanism is a valuable resource, and many metaprogramming languages provide it. We provide a systematic approach for the verification process of the pattern-based metaprograms that use the pattern matching. This approach is based on the pattern-based test design process presented in Chapter 4. The test design process defines a framework that can be instantiated to conduct the test process for these kinds of metaprograms. The pattern-based coverage criteria were proposed, in Chapter 5, to give a systematic approach to generate the test datasets used during the test design process. They also provide a different level of rigor to the test design process.

Taking into consideration the Hypotheses presented in Section 1.3, we achieved the following results:

Hypothesis 1: *The test design based on pattern information from pattern-based metaprograms contributes to the quality of the testing of these programs.*

This hypothesis was investigated using three primary concerns: (1) the cost of using the test design process, (2) the quality of test set produced using the pattern information, and (3) the flexibility allowed when the criteria are used. These concerns a defined by three research questions:

– *Research Question 1: Can the test design based on pattern information from pattern-based metaprograms contribute to the quality of the testing of these programs?*
  Yes. The process of generation of a test set using the pattern information was presented in Section 6.2 for the Rascal language. In the presented case study, the generation was semi-automatic using the LGen tool as grammar-based sentence generator and adapting these sentences to be executed by the test case format used by Rascal. Additionally, in the Section 6.3 is presented the result related to

the usage of the pattern coverage criterion and its contribution to the quality of the pattern-based metaprogram.

– *Research Question 2: Is the cost of the generation of a test set using the pattern information to the test of a pattern-based metaprogram acceptable?*
  Yes. The cost of use of LGen tool was related to the modeling a form to use the tool with the pattern information. We also added some features to the tool to improve the generation process. The overall cost was not elevated, but some points can be improved. An example is the extension of the tool with the support to use the information from patterns.

– *Research Question 3: Is the quality of the generated test set using the pattern information to the test of a pattern-based metaprogram acceptable?*
  Yes. This question was investigated by a validation process of the pattern-based coverage criteria done in Section 6.3. During the validation process, the results obtained indicate that using the pattern information by the proposed coverage criteria presents a certain level of quality to the test sets analyzed. The results can be improved by using an improved implementation of the test data generation process, as discussed early.

– *Research Question 4: Does the pattern-based coverage criteria family allow a flexibility of rigor on the testing of pattern-based metaprograms?*
  Yes. The relation among the coverage criteria are analyzed, and the subsumption relation was shown in Section 5.2. Based on this, it is possible to use a specific coverage criterion to achieve a certain level of rigor with a cost associated with this choice.

This work presents evidence that sustains the Hypothesis 1. This claim is based on the results obtained for each of the research questions detailed previously. So, we can conclude that pattern information can be used to the test of pattern-based metaprograms. The test design process and the coverage criteria proposed here are no perfect and can be improved by a future work detailed in the next section.

Hypothesis 2: *The combination of pattern-based and grammar-based testing contributes to increase the quality of the testing of pattern-based metaprograms.*

This hypothesis claims about the combination of different coverage criteria, pattern-based and grammar-based, to improve the testing of pattern-based metaprograms. The research question used to guide the investigation about it is discussed below.

– *Research Question 5: What is the gain on quality, if any, for the use of the test sets from pattern-based and grammar-based criteria together in relation to the use of each of them individually?*
  The grammar-based test sets can be used to improve the overall testing process for the pattern-based metaprograms. These test sets in our case study can be used to represent the situations where fault programs are given to the metaprogram under test. This kind of situation is not treated by the pattern-based test design or criteria. Consequently, the grammar-based test set can contribute to the overall quality of the testing process.

8.1 Future Work

This thesis has come to an end, but there are some opportunities for future work. Bellow, we present some points that can be research and improved in the future:

1. The pattern-design test design and criteria provide a conceptual framework to improve the test of pattern-based metaprograms using patterns. It is essential to implement a software environment that supports this framework and conduct to the right use of it. A software implementation that uses and improves the steps presented in the case study can bring the pattern-based testing to real metaprograms projects. There are some challenges when trying to implement this software. First, the software should be language agnostic as possible. This characteristic because the pattern-based testing is not related to a specific metaprogramming language and can be used in many of them with direct support to the patterns or not. Second, the test set generation based on the proposed pattern criteria. For this goal, the use of a better grammar-based tool can help in this direction. Another important point is the generation of concrete test cases for a specific metaprogram. This process was conducted ad-hoc here and must be improved to this kind of software implementation.

2. The proposed coverage criteria family had as reference some coverage criteria that uses equivalence partition technique. However, other criteria can be used to define a pattern equivalent to them. These criteria can be investigated, and their data can be compared with those proposed in this work. An example is the *Base Choice Coverage* [Ammann and Offutt, 2008]. It is an intermediary criterion between the All Combinations and Each Choice criteria concerning the criteria subsumption.

3. The coverage criteria validation was conducted using the LGen tool to generate the sentences during the elaboration of the test sets related to the pattern-based

coverage criteria. We can evaluate the use of another tool like the YouGen [Hoffman et al., 2011] or XTextGen [Härtel et al., 2014] to this task. We believe that these tools can improve the sentence generation process to the pattern-based criteria, to generate a complete test set to evaluate better the grammar-based criteria and consequently improve the comparison of their results with the pattern criteria.

4. The validation process related to the proposed coverage criteria can be improved by adding more functions and kind of patterns from Rascal Type Checker to evaluate it. This addition can improve the data about the pattern-based criteria and give us more information about the exact behavior of these criteria during the testing of a pattern-based metaprogram.

5. During the case study realization we realize that pattern information can be used to improve the grammar-based test data generation. The LGen tool provides a flexible framework to generate sentences based on a context-free grammar. This tool can be extended to implement the pattern-based coverage criteria. As discussed in Chapter 7, there is no theoretical research and implemented tool with this feature implemented in the grammar-based community. This kind of implementation can support the use of the pattern-based test design and criteria in metaprograms projects.

6. During the theoretical foundation about the pattern matching presented in Chapter 3, we presented a formalization of the patterns based on term trees. Besides this formalization been a well know and used one, it defines only the variable feature for the pattern language definition. To improve the abstraction capacity of the pattern formalism used in the coverage criteria definitions, we can investigate the formalization using the lambda calculus defined by Jay [2009] to achieve this improvement goal. Additionally, with this new formalization, we can investigate the possibility of using the patterns to describe some semantic characteristics of the metaprogram's input program aiming the improvement of the metaprograms' testing process.

Bibliography

Connor Adsit and Matthew Fluet. An efficient type- and control-flow analysis for system f. In *Proceedings of the 26Nd 2014 International Symposium on Implementation and Application of Functional Languages*, IFL '14, pages 3:1–3:14, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3284-2. doi: 10.1145/2746325.2746327. URL http://doi.acm.org/10.1145/2746325.2746327.

B.B. Agarwal, S.P. Tayal, and M. Gupta. *Software Engineering and Testing*. Computer science series. Jones & Bartlett Learning, 2010. ISBN 9781934015551.

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools*. Pearson Education, Boston, MA, USA, 2nd edition, 2007. ISBN 0-321-48681-1.

Igor N. Aizenberg. *Computational Intelligence: Theory and Applications International Conference, 6th Fuzzy Days Dortmund, Germany, May 25–28 1999 Proceedings*, chapter Neural Networks Based on Multi-valued and Universal Binary Neurons: Theory, Application to Image Processing and Recognition, pages 306–316. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999. ISBN 978-3-540-48774-6. doi: 10.1007/3-540-48774-3_36.

M. Allamanis, H. Peng, and C. Sutton. A convolutional attention network for extreme summarization of source code. *ArXiv e-prints*, February 2016.

Miltiadis Allamanis and Charles Sutton. Mining idioms from source code. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 472–483, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3056-5. doi: 10.1145/2635868.2635901.

Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 2008. ISBN 0521880386, 9780521880381.

J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 402–411, New York, NY, USA, 2005. ACM. ISBN 1-58113-963-2. doi: 10.1145/1062455.1062530.

Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, New York, NY, USA, 1998. ISBN 0-521-77920-0.

Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, and Antoine Reilles. Tom: Piggybacking rewriting on java. In Franz Baader, editor, *Term Rewriting and Applications*, volume 4533 of *Lecture Notes in Computer Science*, pages 36–47. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-73447-5. doi: 10.1007/978-3-540-73449-9_5.

Franco Bazzichi and Ippolito Spadafora. An automatic generator for compiler testing. *IEEE Transactions on Software Engineering*, SE-8(4):343–353, July 1982. ISSN 0098-5589.

Boris Beizer. *Software Testing Techniques*. Van Nostrand Reinhold Co., New York, NY, USA, 2 edition, 1990. ISBN 1850328803.

Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. ISBN 0387310738.

Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Marian Vittek. Elan: A logical framework based on computational systems. *Electronic Notes in Theoretical Computer Science*, 4(0):35 – 50, 1996. ISSN 1571-0661. doi: http://dx.doi.org/10.1016/S1571-0661(04)00032-5. URL http://www.sciencedirect.com/science/article/pii/S1571066104000325.

Abdulazeez S. Boujarwah and Kassem Saleh. Compiler test case generation methods: a survey and assessment. *Information & Software Technology*, 39(9):617–625, 1997.

Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/xt 0.17. a language and toolset for program transformation. *Science of Computer Programming*, 72(1–2):52 – 70, 2008. ISSN 0167-6423. doi: http://dx.doi.org/10.1016/j.scico.2007.11.003. Special Issue on Second issue of experimental software and toolkits (EST).

Timothy A. Budd. Mutation analysis: Ideas, exemples, problems and propects., 1981.

Mihai Christodorescu and Somesh Jha. Static analysis of executables to detect malicious patterns. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, page 12, Berkeley, CA, USA, 2003. USENIX Association.

Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. *SIGPLAN Not.*, 35(9):268–279, September 2000. ISSN 0362-1340. doi: 10.1145/357766.351266.

Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. The maude 2.0 system. In Robert Nieuwenhuis, editor, *Rewriting Techniques and Applications (RTA 2003)*, number 2706 in Lecture Notes in Computer Science, pages 76–87. Springer-Verlag, June 2003.

David M. Cohen, Siddhartha R. Dalal, A. Kajla, and G. C. Patton. The automatic efficient test generator (AETG) system. In *Software Reliability Engineering, 1994. Proceedings., 5th International Symposium on*, pages 303–309, November 1994. doi: 10.1109/ISSRE.1994.341392.

David M. Cohen, Siddhartha R. Dalal, Jesse Parelius, and Gardner C. Patton. The combinatorial design approach to automatic test generation. *IEEE Software*, 13(5):83–88, September 1996. ISSN 0740-7459. doi: 10.1109/52.536462.

David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. The AETG system: An approach to testing based on combinatorial design. *IEEE Trans. Softw. Eng.*, 23(7):437–444, July 1997. ISSN 0098-5589. doi: 10.1109/32.605761.

Alain Colmerauer and Philippe Roussel. History of programming languages—ii. In Thomas J. Bergin, Jr. and Richard G. Gibson, Jr., editors, *History of Programming languages—II*, chapter The Birth of Prolog, pages 331–367. ACM, New York, NY, USA, 1996. ISBN 0-201-89502-1. doi: 10.1145/234286.1057820.

S. D. Conte, H. E. Dunsmore, and V. Y. Shen. *Software Engineering Metrics and Models*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1986. ISBN 0-8053-2162-4.

Lee Copeland. *A Practitioner's Guide to Software Test Design*. Artech House, Inc., Norwood, MA, USA, 2004. ISBN 158053791X.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, England, 3rd edition, 2009. ISBN 9780262033848.

Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative programming - methods, tools and applications*. Addison-Wesley, 2000. ISBN 978-0-201-30977-5.

Jacek Czerwonka. Pairwise testing, 2016. URL `http://www.pairwise.org/`. Accessed: 2016-08-08.

O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured Programming*. Academic Press Ltd., London, UK, UK, 1972. ISBN 0-12-200550-3.

Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. Automated testing of refactoring engines. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 185–194, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-811-4. doi: 10.1145/1287624.1287651. URL http://doi.acm.org/10.1145/1287624.1287651.

Márcio Eduardo Delamaro, José Carlos Maldonado, and Mario Jino. *Introdução ao Teste de* Software. Elsevier Editora Ltda, Rio de Janeiro, RJ, Brasil, 2007. ISBN 978-85-352-2634-8.

R.A. DeMillo, D.S. Guindi, W.M. McCracken, A.J. Offutt, and K.N. King. An extended overview of the mothra software testing environment. In *Software Testing, Verification, and Analysis, 1988., Proceedings of the Second Workshop on*, pages 142–151, Jul 1988. doi: 10.1109/WST.1988.5369.

Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978. ISSN 0018-9162. doi: http://dx.doi.org/10.1109/C-M.1978.218136.

A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific Publishing Co., 1996.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-63361-2.

Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov. Test generation through programming in udita. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 225–234, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-719-6. doi: 10.1145/1806799.1806835. URL http://doi.acm.org/10.1145/1806799.1806835.

Keith Golden and Wanlin Pang. *Constraint Reasoning over Strings*, pages 377–391. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003. ISBN 978-3-540-45193-8. doi: 10.1007/978-3-540-45193-8_26. URL http://dx.doi.org/10.1007/978-3-540-45193-8_26.

John B. Goodenough and Susan L. Gerhart. Toward a theory of test data selection. *SIGPLAN Not.*, 10(6):493–510, 1975. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/390016.808473.

Frederick M. Goodman. *Algebra: Abstract and Concrete*. SemiSimple Press, Iowa City, IA, USA, 2rd edition, 2012.

James Gosling, Bill Joy, Guy L. Steele Jr, and Gilad Bracha. *The Java Language Specification*, volume 1. Addison Wesley, 3rd edition, 2005. ISBN 9780321246783.

Kenneth V. Hanford. Automatic generation of test cases. *IBM Systems Journal*, 9(4): 242–257, 1970.

Johannes Härtel, Lukas Härtel, and Ralf Lämmel. Test-data generation for xtext. In *Software Language Engineering*, pages 342–351. Springer, 2014.

Cleverton Hentz. Geração automática de testes a partir de descrições de linguagens. Master's thesis, Programa de Pós-graduação em Sistemas e Computação da Universidade Federal do Rio Grande do Norte, Natal, RN, 2010.

Cleverton Hentz, Jurgen J. Vinju, and Anamaria M. Moreira. Reducing the cost of grammar-based testing using pattern coverage. In Khaled El-Fakih, Gerassimos Barlas, and Nina Yevtushenko, editors, *Testing Software and Systems*, volume 9447 of *Lecture Notes in Computer Science*, pages 71–85. Springer International Publishing, 2015. ISBN 978-3-319-25944-4. doi: 10.1007/978-3-319-25945-1_5. URL `http://dx.doi.org/10.1007/978-3-319-25945-1_5`.

Robert M. Hierons, Kirill Bogdanov, Jonathan P. Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, Kalpesh Kapoor, Paul Krause, Gerald Lüttgen, Anthony J. H. Simons, Sergiy Vilkomir, Martin R. Woodward, and Hussein Zedan. Using formal specifications to support testing. *ACM Comput. Surv.*, 41(2):9:1–9:76, February 2009. ISSN 0360-0300. doi: 10.1145/1459352.1459354.

Mark Hills. *Streamlining Control Flow Graph Construction with DCFlow*, pages 322–341. Springer International Publishing, Cham, 2014. ISBN 978-3-319-11245-9. doi: 10.1007/978-3-319-11245-9_18. URL `http://dx.doi.org/10.1007/978-3-319-11245-9_18`.

Daniel Hoffman, David Ly-Gagnon, Paul Strooper, and Hong-Yi Wang. Grammar-based test generation with YouGen. *Software: Practice and Experience*, 41:427–447, 2011. ISSN 1097-024X. doi: 10.1002/spe.1017. URL `http://dx.doi.org/10.1002/spe.1017`.

Christoph M. Hoffmann and Michael J. O'Donnell. Pattern matching in trees. *J. ACM*, 29(1):68–95, January 1982. ISSN 0004-5411. doi: 10.1145/322290.322295. URL http://doi.acm.org/10.1145/322290.322295.

W. E. Howden. Methodology for the generation of program test data. *IEEE Trans. Comput.*, 24(5):554–560, 1975. ISSN 0018-9340. doi: http://dx.doi.org/10.1109/T-C.1975.224259.

P. Hudak, S. Peyton Jones, and P. Wadler (editors). Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2). *ACM SIGPLAN Notices*, 27(5), May 1992a.

P. Hudak, S. Peyton Jones, and P. Wadler (editors). Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2). *ACM SIGPLAN Notices*, 27(5), May 1992b.

Daniel Jackson. Alloy: A lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, April 2002. ISSN 1049-331X. doi: 10.1145/505145.505149. URL http://doi.acm.org/10.1145/505145.505149.

Vilas Jagannath, Yun Young Lee, Brett Daniel, and Darko Marinov. Reducing the costs of bounded-exhaustive testing. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, FASE '09, pages 171–185, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-00592-3. doi: 10.1007/978-3-642-00593-0_12. URL http://dx.doi.org/10.1007/978-3-642-00593-0_12.

Barry Jay. *Pattern Calculus: Computing with Functions and Structures*. Springer Publishing Company, Incorporated, 1st edition, 2009. ISBN 3540891846, 9783540891840.

Sarfraz Khurshid. *Generating Structurally Complex Tests from Declarative Constraints*. PhD thesis, Cambridge, MA, USA, 2004. AAI0806118.

Adam Kiezun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. Hampi: A solver for string constraints. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 105–116, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-338-9. doi: 10.1145/1572272.1572286. URL http://doi.acm.org/10.1145/1572272.1572286.

Damir Kirasić and Danko Basch. *Knowledge-Based Intelligent Information and Engineering Systems: 12th International Conference, KES 2008, Zagreb, Croatia, September 3-5, 2008, Proceedings, Part I*, chapter Ontology-Based Design Pattern Recognition, pages 384–393. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-85563-7. doi: 10.1007/978-3-540-85563-7_50.

Paul Klint. A Meta-Environment for generating programming environments. *ACM Trans. Softw. Eng. Methodol.*, 2(2):176–201, 1993.

Paul Klint. How to test a meta-program? invited talk at the Workshop on Scalable Language Specification (SLS 2013), 2013. URL `https://www.microsoft.com/en-us/research/video/scalable-language-specification-2013-5/`.

Paul Klint, Jurgen J. Vinju, and Tijs van der Storm. Language design for meta-programming in the software composition domain. In Alexandre Bergel and Johan Fabry, editors, *Software Composition, 8th International Conference, SC 2009, Zurich, Switzerland, July 2-3, 2009. Proceedings*, volume 5634 of *Lecture Notes in Computer Science*, pages 1–4. Springer, 2009. ISBN 978-3-642-02654-6.

Paul Klint, Jurgen J. Vinju, and Tijs van der Storm. Rascal tutor: Language and libraries, 2010. URL `http://tutor.rascal-mpl.org/Rascal/Rascal.html`. Accessed: 2017-05-10.

Paul Klint, Tijs van der Storm, and Jurgen Vinju. Easy meta-programming with rascal. In João Fernandes, Ralf Lämmel, Joost Visser, and João Saraiva, editors, *Generative and Transformational Techniques in Software Engineering III*, volume 6491 of *Lecture Notes in Computer Science*, pages 222–289. Springer Berlin / Heidelberg, 2011.

Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, pages 323–350, 1977. ISSN 1571-0661. doi: 10.1137/0206024.

A. S. Kossatchev and M. A. Posypkin. Survey of compiler testing methods. *Program. Comput. Softw.*, 31(1):10–19, 2005. ISSN 0361-7688. doi: http://dx.doi.org/10.1007/s11086-005-0008-6.

Ralf Lämmel. Grammar testing. In *Proc. of Fundamental Approaches to Software Engineering (FASE) 2001*, volume 2029 of *LNCS*, pages 201–216. Springer-Verlag, 2001.

Ralf Lämmel and Wolfram Schulte. Controllable combinatorial coverage in grammar-based testing. In M. Ümit Uyar, Ali Y. Duale, and Mariusz A. Fecko, editors, *TestCom*, volume 3964 of *LNCS*, pages 19–38, New York, NY, USA, 2006. Springer. doi: http://dx.doi.org/10.1007/11754008_2.

Davy Landman, Alexander Serebrenik, and Jurgen Vinju. Empirical analysis of the relationship between CC and SLOC in a large corpus of Javah methods. In *30th IEEE International Conference on Software Maintenance and Evolution, ICSME 2014*, 2014.

Yu Lei and Kuo-Chung Tai. In-parameter-order: a test generation strategy for pairwise testing. In *High-Assurance Systems Engineering Symposium, 1998. Proceedings. Third IEEE International*, pages 254–261, November 1998. doi: 10.1109/HASE.1998.731623.

Xavier Leroy. A formally verified compiler back-end. *J. Autom. Reason.*, 43:363–446, December 2009. ISSN 0168-7433. doi: 10.1007/s10817-009-9155-4. URL http://portal.acm.org/citation.cfm?id=1666192.1666216.

Huiqing Li. *Refactoring Haskell Programs*. PhD thesis, The University of Kent, 2006.

Rüdiger Lincke, Jonas Lundberg, and Welf Löwe. Comparing software metrics tools. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 131–142, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-050-0. doi: 10.1145/1390630.1390648. URL http://doi.acm.org/10.1145/1390630.1390648.

Thomas J. McCabe. A complexity measure. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, page 407, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

Melina Mongiovi, Gustavo Mendes, Rohit Gheyi, Gustavo Soares, and Márcio Ribeiro. Scaling testing of refactoring engines. In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, pages 371–380, 2014. doi: 10.1109/ICSME.2014.59. URL http://dx.doi.org/10.1109/ICSME.2014.59.

David Monniaux. *A Survey of Satisfiability Modulo Theory*, pages 401–425. Springer International Publishing, Cham, 2016. ISBN 978-3-319-45641-6. doi: 10.1007/978-3-319-45641-6_26. URL http://dx.doi.org/10.1007/978-3-319-45641-6_26.

Anamaria Martins Moreira, Cleverton Hentz, and Viviane de Menezes Ramalho. Application of a syntax-based testing method and tool to software product lines. In *7th Brazilian Workshop on Systematic and Automated Software Testing, SAST 2013*, 2013.

Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. Wiley Publishing, 3rd edition, 2011. ISBN 9781118031964.

Martin Odersky, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, Matthias Zenger, and et al. An overview of the scala programming language. Technical report, École Polytechnique Fédérale de Lausanne, 2004.

A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. An experimental determination of sufficient mutant operators. *ACM Trans. Softw. Eng. Methodol.*, 5(2):99–118, April 1996. ISSN 1049-331X. doi: 10.1145/227607.227610.

Shari Lawrence Pfleeger. *Software Engineering: Theory and Practice*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 2001. ISBN 0130290491.

Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages*. MIT Press, 2005. ISBN 9780262162289. URL https://mitpress.mit.edu/books/advanced-topics-types-and-programming-languages.

Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 5 edition, 2001. ISBN 0073655783.

Paul Purdom. A sentence generator for testing parsers. *BIT Numerical Mathematics*, 12 (4):366–375, 1972. ISSN 0006-3835.

Stefano Crespi Reghizzi. *Formal Languages and Compilation*. Springer London, April 2009. ISBN 9781848820500. URL https://link.springer.com/book/10.1007/978-1-84882-050-0.

G. Rozenberg and A. Salomaa. *Handbook of Formal Languages: Word, Language, Grammar*, volume 1 of *Handbook of Formal Languages*. Springer, 1997. ISBN 9783540604204.

Lt. Richard L. Sauder. A general test data generator for COBOL. In *Spring Joint Computer Conference*, volume 21, pages 317–324, San Francisco, CA, USA, 1962.

Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM. ISBN 1-59593-014-0. doi: 10.1145/1081706.1081750. URL http://doi.acm.org/10.1145/1081706.1081750.

Tim Sheard. *Semantics, Applications, and Implementation of Program Generation: Second International Workshop, SAIG 2001 Florence, Italy, September 6, 2001 Proceedings*, chapter Accomplishments and Research Challenges in Meta-programming, pages 2–44. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001. ISBN 978-3-540-44806-8. doi: 10.1007/3-540-44806-3_2.

Gustavo Soares. Making program refactoring safer. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 521–522, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-719-6. doi: 10.1145/1810295.1810461. URL http://doi.acm.org/10.1145/1810295.1810461.

Gustavo Soares, Rohit Gheyi, and Tiago Massoni. Automated behavioral testing of refactoring engines. *IEEE Trans. Software Eng.*, 39(2):147–162, 2013. doi: 10.1109/TSE.2012.19. URL http://doi.ieeecomputersociety.org/10.1109/TSE.2012.19.

Ian Sommerville. *Engenharia de Software*. Pearson Addison Wesley, São Paulo, SP, Brasil, 6 edition, 2003. ISBN 85-88639-07-6.

Simon Thompson, Claus Reinke, and Huiqing Li. Refactoring functional programs. Technical Report GR/R75052/01, University of Kent, Kent, UK, 2006. URL https://www.cs.kent.ac.uk/projects/refactor-fp/FinalReport.pdf.

Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007. ISBN 0123725011, 9780080466484.

Mark G. J. van den Brand, Arie van Deursen, Jan Heering, H. A. de Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter A. Olivier, Jeroen Scheerder, Jurgen J. Vinju, Eelco Visser, and Joost Visser. The ASF+SDF Meta-environment: A component-based language development environment. In *Compiler Construction*, volume 2027 of *LNCS*, pages 365–370, Genova, Italy, 2001. Springer-Verlag. ISBN 3-540-41861-X.

Stewart N. Weiss. What to compare when comparing test data adequacy criteria. *SIGSOFT Softw. Eng. Notes*, 14(6):42–49, October 1989. ISSN 0163-5948. doi: 10.1145/70739.70742.

N. Wirth. The programming language oberon. *Softw. Pract. Exper.*, 18(7):671–690, July 1988. ISSN 0038-0644. doi: 10.1002/spe.4380180707. URL http://dx.doi.org/10.1002/spe.4380180707.

S. V. Zelenov and S. A. Zelenova. Generation of positive and negative tests for parsers. *Program. Comput. Softw.*, 31(6):310–320, 2005. ISSN 0361-7688. doi: http://dx.doi.org/10.1007/s11086-005-0040-6.

Hanlin Zhang, Yevgeniy Cole, Linqiang Ge, Sixiao Wei, Wei Yu, Chao Lu, Genshe Chen, Dan Shen, Erik Blasch, and Khanh D. Pham. Scanme mobile: A cloud-based android malware analysis service. *SIGAPP Appl. Comput. Rev.*, 16(1):36–49, April 2016. ISSN 1559-6915. doi: 10.1145/2924715.2924719.

Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/267580.267590.

— A —
The Rascal Implementations

A.1 Rascal Mutator

Listing A.1 presents an extract from the module “`ModuleMutator.rsc`” [1] that implements the mutation operators.

[1]The module is available on `https://git.io/vFVIl`.

Listing A.1: The mutator module implemented in Rascal.

```
private set[str] muOpers = {
 //"ABS",
 "COR1","COR2","COR3","COR4","COR5","COR6",
 "BSR1","BSR2","BSR3","BSR4","BSR5",
 "PWA","PWAR"
};

tuple[bool,PatternWithAction] muOperPattAction("PWA", (PatternWithAction) ‘<Pattern p> : <Statement _>‘) = <true,
(PatternWithAction) ‘<Pattern p> : throw "mutant! PWA: Remove action from pattern and action.";‘>
when !(p is concrete);

tuple[bool,PatternWithAction] muOperPattAction("PWAR", (PatternWithAction) ‘<Pattern p> =\> <Replacement _>‘) = <true,
(PatternWithAction) ‘<Pattern p> : throw "mutant! PWAR: Remove pattern rewrite.";‘>
when !(p is concrete);

default tuple[bool,PatternWithAction] muOperPattAction(str opId, PatternWithAction pa) = <false,pa>;

tuple[bool,Statement] muOperStm("BSR1", (Statement) ‘if (<{Expression ","}+ cond>) <Statement s>‘) = <true,
(Statement) ‘if (<{Expression ","}+ cond>)
              '  throw "mutant! BSR1: Remove if conditionals. Case 1.";‘>
when !(cond is concrete);

tuple[bool,Statement] muOperStm("BSR2", (Statement) ‘if (<{Expression ","}+ cond>) <Statement s> else <Statement t>‘) = <true,
(Statement) ‘if (<{Expression ","}+ cond>) throw "mutant! BSR2: Remove if conditionals. Case 2."; else <Statement t>‘ >
when !(cond is concrete);

tuple[bool,Statement] muOperStm("BSR3", (Statement) ‘if (<{Expression ","}+ cond>) <Statement s> else <Statement t>‘) = <true,
(Statement) ‘if (<{Expression ","}+ cond>) <Statement s> else throw "mutant! BSR3: Remove if conditionals. Case 3.";‘ >
when !(cond is concrete);

tuple[bool,Statement] muOperStm("BSR4", (Statement) ‘while (<{Expression ","}+ cond>) <Statement s>‘) = <true,
(Statement) ‘while (<{Expression ","}+ cond>) throw "mutant! BSR4: Remove while conditionals.";‘ >
when !(cond is concrete);

tuple[bool,Statement] muOperStm("BSR5", (Statement) ‘for (<{Expression ","}+ cond>) <Statement s>‘) = <true,
(Statement) ‘for (<{Expression ","}+ cond>) throw "mutant! BSR5: Remove for conditionals.";‘ >
when !(cond is concrete);
```

```
default tuple[bool,Statement] muOperStm(str opId, Statement stm) = <false,stm>;

tuple[bool,Expression] muOperExp("COR1", (Expression) `<Expression lhs> \>= <
      Expression rhs>`) = <true,
(Expression) `<Expression lhs> \<= <Expression rhs>`>;

tuple[bool,Expression] muOperExp("COR2", (Expression) `<Expression lhs> \<= <
      Expression rhs>`) = <true,
(Expression) `<Expression lhs> \>= <Expression rhs>`>;

tuple[bool,Expression] muOperExp("COR3", (Expression) `<Expression lhs> \< <
      Expression rhs>`) = <true,
(Expression) `<Expression lhs> \> <Expression rhs>`>;

tuple[bool,Expression] muOperExp("COR4", (Expression) `<Expression lhs> \> <
      Expression rhs>`) = <true,
(Expression) `<Expression lhs> \< <Expression rhs>`>;

tuple[bool,Expression] muOperExp("COR5", (Expression) `<Expression lhs> == <
      Expression rhs>`) = <true,
(Expression) `<Expression lhs> != <Expression rhs>`>;

tuple[bool,Expression] muOperExp("COR6", (Expression) `<Expression lhs> != <
      Expression rhs>`) = <true,
(Expression) `<Expression lhs> == <Expression rhs>`>;

default tuple[bool,Expression] muOperExp(str opId, Expression e) = <false,e>;
```

A.2 The Rascal Patterns Extraction Algorithms

Listing A.2 presents the module "`PatternExtraction.rsc`" used during the pattern extraction step.

Listing A.2: Pattern extraction module.

```
@bootstrapParser
module PatternExtraction

import util::Reflective;
import lang::rascal::\syntax::Rascal;
import ParseTree;
import IO;
import Map;
import Set;

import MetaPattern;

anno loc node@location;

list[node] getPrefixPatterns(loc m) =
  [ implode(#node, p) | p <- extractPatterns(m) ];

set[str] filterTypedVars(set[node] i) =
  { name | /"typedVariable"(_,name) <- i };

set[node] filterCallOrTree(set[node] i) =
  { p | p:"callOrTree"(_,_,_) <- i };

private tuple[int total,int filtered, list[str] patts] countPattTree(Tree t, set
    [str] lblSet) {
  tuple[int total,int filtered, list[str] patts] ret = <0,0,[]>;
  //println("countPattTree: begin <lblSet>");
  list[str] lst = [];
  top-down-break visit (t) {
    case Pattern p : {
      if (!isEmpty(lblSet) && appl(prod(/label(l,_),_,_),_) := p && l in lblSet)
          {
        //println("countPattTree: Patt <p>");
        ret.filtered += 1;
        lst += "<p>";
      } else if ( isEmpty(lblSet) ) lst += "<p>";
      ret.total += 1;
    }
  }
  ret.patts = lst;
  return ret;
}

list[Pattern] extractPatterns(loc lm) {
  if (Module m := parseModule(lm)) {
    return [ p | /Pattern p := m];
  }

  throw "could not parse module <lm>";
}

public map[str, list[Tree]] getPatternTree(loc lt) {
  Tree m = parseModule(lt);
  return getPatternTree(m);
}

private list[Pattern] extractPattTree(Tree t, set[str] lblSet) {
  list[Pattern] ret = [];
  top-down-break visit (t) {
    case Pattern p : {
      if (isEmpty(lblSet) ||
          ( appl(prod(/label(l,_),_,_),_) := p && l in lblSet ) ) {
        ret += p;
      }
    }
  }
  }
  return ret;
}

alias aGPRet = map[str , tuple[str fSig, list[Tree] patts]];

public aGPRet getPatternTree(Tree t, set[str] lblFSet) {
  aGPRet r = ();

  str nm,sig;

  for (/FunctionDeclaration f := t) {
    Tree st;
    if (f has signature) {
      nm = "<f.signature.name>";
      sig = "<f.signature>";
    }

    try {
      if (f has expression) {
        st = f.expression;
      }

      if (f has body) {
        st = f.body;
      }

      if (nm in r) {
        r[nm].patts += extractPattTree(st,lblFSet);
      } else {
        r += (nm:<sig, extractPattTree(st,lblFSet)>);
      }
    } catch e : println("getPatternTree exception: <e>");
  }

  return r;
}

public set[str] defLblFilter = {"callOrTree","concrete","descendant"};

public aGPRet getPatternTree(Tree t) = getPatternTree(t,defLblFilter);

public map[str, int] getPatternStats(loc lt) {
  Tree m = parseModule(lt);
  return getPatternStats(m);
}

public map[str, int] getPatternStats(Tree t, bool quiet, set[str] lblFSet = {"
    callOrTree","concrete","descendant"}) {
  //str nm;
  map[str, int] r = ();
  int fNum = 0;
  int TotNum = 0;
  tuple[int,int] ret;

  int fCount = 0;
  int fPatt = 0;
  int fWOPatt = 0;
  str fName;
  set[str] pSet = {};

  for (/FunctionDeclaration f := t) {
    Tree st;
    fCount+=1;
    if (!quiet) {
      println("<f.signature.name>(<f@\loc.begin.line>,<f@\loc.end.line>) LOC: <(
          f@\loc.end.line - f@\loc.begin.line)+1>");
    }
    }
    /*
    if (f has signature) {
      fName = "<f.signature.name>";
    }
    */

    try {
      if (f has expression) {
        st = f.expression;
      }

      if (f has body) {
        st = f.body;
      }

      ret = countPattTree(st,lblFSet);

      fNum += ret.filtered;
      TotNum += ret.total;
      pSet += toSet(ret.patts);

      //Get info about functions
      if (ret.total >0) {
        fPatt+=1;
      } else {
        fWOPatt+=1;
      }
    //println("<nm>::<ret.filtered>");

    } catch e : println("getPatternStats exception: <e>");
  }

  //println("getPatternStats: <fNum>, <TotNum>");
  assert fNum<=TotNum : "getPatternStats: Filtred pattern is grater than total
      patterns.";

  return ("filtered":fNum,
          "unfiltered":TotNum-fNum,
          "uniquePatts":size(pSet),
          "total":TotNum,
          "func": fCount,
          "funcPatt": fPatt,
          "funcWOPatt": fWOPatt);
}

private loc sMSample = |cwd:///MSample.rsc|;
private loc sMcCabeM3 = |cwd:///McCabeM3.rsc|;

void main() {
  println("Checking the files...");
  if (!(exists(sMSample) && isFile(sMSample))) {
    throw "Invalid File!";
  }

  Tree m = parseModule(sMSample);
  map[str,int] stats = getPatternStats(m);
  println("Stats from sMSample");
  for (e <- stats) {
    println("<e>: <stats[e]>");
  }


  list[Pattern] l1 = extractPatterns(sMSample);
  println("List Patt from sMSample - <size(l1)>");
  int i = 1;
  for (e <- l1) {
    println("Patt Index: <i>, \'<e>\'");
    i = i + 1;
  }

  println("List Patt from sMcCabeM3");

  m = parseModule(sMcCabeM3);
  stats = getPatternStats(m);
  println("Stats from sMcCabeM3");
  for (e <- stats) {
    println("<e>: <stats[e]>");
  }

  println("Test tree--Initial Tree");
  //rprintln(m);
  map[str, list[Tree]] patts = getPatternTree(sMcCabeM3);
  println("getPatternTree::Result");
  for (e <- patts) {
    print("<e>::");
    println(["<p>" | p <- patts[e]]);
    //implode(#node, p) | p <- extractPatterns(m)
  }
}

test bool tstBasic1() = size(extractPatterns(sMSample)) == 1;
test bool tstBasic2() = size(extractPatterns(sMcCabeM3)) > 0;
test bool tstBasic3() = size(getPatternTree(sMSample)) == 1;
test bool tstBasic4() = size(getPatternTree(sMcCabeM3)) > 0;
test bool tstBasic5() = size(getPatternStats(sMcCabeM3)) == 3;
test bool tstBasic6() = getPatternStats(sMcCabeM3)["total"] == 13;
```

A.3 The Patterns Language Syntax of Rascal

Listing A.3: The Rascal's patterns language definition.

```
syntax Pattern
  = \set                  : "{" {Pattern ","}* elements0 "}"
  | \list                 : "[" {Pattern ","}* elements0 "]"
  | qualifiedName        : QualifiedName qualifiedName
  | multiVariable        : QualifiedName qualifiedName "*"
  | splice               : "*" Pattern argument
  | splicePlus           : "+" Pattern argument
  | negative             : "-" Pattern argument
  | literal              : Literal literal
  | \tuple                : "\<" {Pattern ","}+ elements "\>"
  | typedVariable        : Type type Name name
  | \map                  : "(" {Mapping[Pattern] ","}* mappings ")"
  | reifiedType          : "type" "(" Pattern symbol "," Pattern definitions ")"
  | callOrTree           : Pattern expression "(" {Pattern ","}* arguments
    KeywordArguments[Pattern] keywordArguments ")"
  > variableBecomes      : Name name ":" Pattern pattern
  | asType               : "[" Type type "]" Pattern argument
  | descendant           : "/" Pattern pattern
  | anti                 : "!" Pattern pattern
  | typedVariableBecomes: Type type Name name ":" Pattern pattern
    ;
```

The Listing A.3 presents an extract from the module "`Rascal.rsc`" [2] that define the Rascal syntax using the language itself. The module version used is `4e9cd4e` and the code extract start at line 860 of the original code.

[2]The Rascal syntax module is available on `https://git.io/vQs3a`.