Universidade do Minho
Escola de Engenharia
Departamento de Informática

Liu Chong

Lightweight Trustworthy High-Level Software Design

Programa de Doutoramento em Informática das Universidades do Minho, de Aveiro e do Porto

Universidade do Minho

January 2022

UMinho | 2022

This work is financed by the ERDF – European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme and by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project POCI-01-0145-FEDER-016826.

FCT Fundação para a Ciência e a Tecnologia

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Liu Chong

Lightweight Trustworthy High-Level Software Design

Tese de Doutoramento
Programa de Doutoramento em Informática das Universidades do Minho, de Aveiro e do Porto

Trabalho realizado sob a orientação do
Professor Doutor Manuel Alcino Cunha
e do
Professor Doutor Nuno Moreira Macedo

January 2022

DIREITOS DE AUTOR E CONDIÇÕES DE UTILIZAÇÃO DO TRABALHO POR TERCEIROS

Este é um trabalho académico que pode ser utilizado por terceiros desde que respeitadas as regras e boas práticas internacionalmente aceites, no que concerne aos direitos de autor e direitos conexos.

Assim, o presente trabalho pode ser utilizado nos termos previstos na licença abaixo indicada.

Caso o utilizador necessite de permissão para poder fazer um uso do trabalho em condições não previstas no licenciamento indicado, deverá contactar o autor, através do RepositóriUM da Universidade do Minho.

Licença concedida aos utilizadores deste trabalho

Atribuição

CC BY

https://creativecommons.org/licenses/by/4.0/

ACKNOWLEDGEMENTS

I still remember when I walked out of Professor Alcino's office for the first time five years ago, at the beginning of my five-year doctoral journey. I am very grateful to Professor Alcino for choosing me as his PhD student and introducing me to the wonders of scientific research. These five years have been the most unforgettable and happiest time for me. This experience will be one of the most memorable moments of my life.

Foremost, I would like to express my sincere gratitude to my supervisors Professor Alcino and Professor Nuno for the continuous support of my studies and research. I am sure that without your patience, enthusiasm, and immense knowledge, this journey wouldn't be possible. I could not have imagined having better advisors.

Besides my supervisors, I would like to thank Professor Alexandre Santos, Professor Joaquim Macedo, and Professor António Duarte Costa, who gave me a lot of help during my studies at the University of Minho. I thank my Master's supervisor, Professor Liu Dayou – without his help I would never have the opportunity to study in Portugal as an exchange student and would never choose to pursue my PhD in Portugal.

I am very grateful to my country. During the COVID-19 epidemic, I received masks, disinfectants, and medicine from the embassy many times, which made me feel the warmth of home outside. I would like to thank my friends and my colleagues in Lab Room 207 who helped and colored my life in Portugal. Although we hadn't seen each other for a long time since the outbreak, we spent a lot of happy times together.

Finally, I would like to express my heartfelt thanks to all the professors who participated in the thesis evaluation and defense.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity.

I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

RESUMO

A modelação e análise formal de software é essencial para obter um projecto de software confiável antes da implementação. O Alloy, uma linguagem de especificação com uma ferramenta de análise automática, é uma abordagem popular para esta tarefa. Frequentemente, um projecto de software inclui muitas variantes com grande partilha de código. Em vez de considerar cada variante individualmente, o *desenvolvimento de software orientado à funcionalidade* procura desenvolver em conjunto toda a família de variantes, também designada por *linha de produtos de software* (LPS). Aspectos em comum são organizados em *funcionalidades*, implementando cada variante um sub-conjunto das mesmas. Existem duas abordagens típicas para implementação de uma LPS: abordagens composicionais, onde cada funcionalidade é implementada num módulo distinto, e abordagens anotativas, onde o código específico de cada funcionalidade é assinalado com uma anotação. A primeira é melhor para adicionar grandes blocos de código a uma funcionalidade, por exemplo uma nova classe, enquanto que a segunda suporta melhor pequenas extensões, tais como adicionar uma instrução a um método.

O primeiro objectivo desta tese é propor uma extensão anotativa para o Alloy, para suportar a *concepção formal de software orientada à funcionalidade*. A ideia é suportar pequenas extensões a um modelo mas sem os problemas de compreensão que derivam das típicas anotações `#ifdef` usadas na implementação de uma LPS. Para tal, permitimos cores de fundo para identificar os fragmentos associados com cada funcionalidade, dando origem à linguagem Colorful Alloy. Também propusemos uma técnica de análise amalgamada para verificar toda uma família de variantes de uma só vez. O segundo objectivo é propor uma técnica para migração de um conjunto de variantes, possivelmente desenvolvidos com a abordagem *clone-and-own*, para um único modelo em Colorful Alloy. Para tal, propusemos um catálogo de refatorações e mostramos como podem ser usadas para iterativamente migrar modelos Alloy clonados para um único modelo "colorido". Também desenvolvemos uma técnica de migração que automatiza todo este processo. Este trabalho foi avaliado com recurso a vários casos de estudo de LPSs, desenvolvidos quer pro-activamente com Colorful Alloy, quer com a abordagem clone-and-own com Alloy normal. Esta avaliação mostrou que a técnica de análise amalgamada pode aumentar consideravelmente a eficiência da verificação de uma família de variantes, quando comparada com a análise variante a variante. Também mostrou que a técnica de migração (incluindo a completamente automática) pode reduzir significativamente a quantidade de código clonado, o que em princípio permitirá simplificar a compreensão de um projecto de uma LPS.

Palavras-chave Concepção formal de software, Alloy, linhas de produtos de software, variabilidade, refatoração, clone-and-own.

ABSTRACT

Formal modeling and analysis is essential to achieve a trustworthy software design prior to its implementation. Alloy, a specification language with a lightweight model finder, is a popular approach to accomplish this task. Frequently, a software design encompasses many variants with a large commonality between them. Instead of considering each variant individually, *feature-oriented software development* tackles at once the whole family of variants, also known as a *software product line* (SPL). Commonalities are organized as *features*, with each variant implementing a particular set of features. SPL implementation techniques mainly fall into two groups: compositional approaches, which implement features in distinct modules, and annotative approaches, that wrap feature-specific code with annotations. The former is well suited to coarse-grained feature extensions, such as adding a new class, while the latter is better for fine-grained extensions, like adding a statement in a method.

The first goal of this thesis is to propose an annotative extension to Alloy, to support formal *feature-oriented software design*. Our aim was to easily support fine-grained extensions to a model, but without the comprehension obstacles of the typical `#ifdef` annotations used in SPL implementation. To that end, we added support for background colors to identify the fragments associated with each feature, ending up with the Colorful Alloy language. We also proposed an amalgamated analysis technique that can verify a full family of design variants at once. Our second goal was to propose a technique for migrating a collection of variants, possibly developed with the *clone-and-own* approach, into a single managed Colorful Alloy design. To achieve this, we proposed a catalog of refactorings and showed how they can be used to iteratively merge cloned Alloy models into a colorful model. We also proposed a one-step merging strategy that automates this technique. We evaluated our work with several SPL case studies that were either developed proactively directly with Colorful Alloy or using clone-and-own using normal Alloy. The evaluation showed that the amalgamated analysis strategy can considerably speed-up the verification of a full family of design variants, when compared to the iterative and separate analysis of each variant. It also showed that the clone merging technique (including the automatic one) can substantially reduce the amount of cloned code, which in principle simplifies the understanding of an SPL design.

Keywords Formal software design, Alloy, software product lines, variability, refactoring, clone-and-own.

CONTENTS

1 Introduction 1

2 Formal Software Design with Alloy 6

2.1 Alloy by Example 6

2.1.1 E-commerce Example 7

2.1.2 Signature and Field Declaration 8

2.1.3 Type System 9

2.1.4 Exploring Scenarios 11

2.1.5 Specifying Constraints 13

2.1.6 Verifying Assertions 15

2.1.7 Modularization 17

2.2 Formal Presentation of the Language 21

2.2.1 Formal Syntax 21

2.2.2 Formal Semantics 24

2.2.3 Type Inference 27

2.2.4 Analysis 29

2.3 Refactoring Alloy Models 32

2.3.1 Laws for Signatures 33

2.3.2 Laws for Fields 34

2.3.3 Laws for Formulas 36

2.4 Alloy Extensions 37

3 Feature-oriented Software Design 44

3.1 Feature Modeling 45

3.1.1 Specifying Feature Models 45

3.1.2 Analyzing Feature Models 49

3.2 Modeling in Feature-oriented Design 55

3.2.1 Ad-hoc Approaches 57
3.2.2 Composition-based Languages 61
3.2.3 Annotation-based Languages 71
3.3 Analysis in Feature-oriented Design 76
3.4 Supporting Clone-and-own 81
3.4.1 Migrating Clones into an SPL 82
4 Colorful Alloy 88
4.1 The Background Color Approach 89
4.2 Colorful Alloy Syntax 93
4.3 An Example of Proactive SPL Design 95
4.4 Type Checking Rules 99
4.5 Semantics 104
4.6 Analysis 106
5 Merging Cloned Alloy Models with Colorful Refactorings 113
5.1 Migrating Code Clones into an SPL with Refactoring 114
5.2 Refactoring Rules for Colorful Alloy 117
5.3 Migrating Clones into a Colorful Alloy Model 129
5.4 Automatic Merging Strategy 133
6 Implementation and Evaluation 138
6.1 The Colorful Alloy Analyzer 138
6.2 Proactive Case Studies 142
6.2.1 E-commerce 142
6.2.2 GrandpaFamily 143
6.2.3 Alloy4Fun 144
6.2.4 Graph 148
6.2.5 Vending Machine 149
6.2.6 Bestiary 151
6.2.7 Comparison with Compositional Approaches 151
6.3 Evaluating the Clone Migration Strategy 152

6.3.1 Extractive Case Studies 153
6.3.2 Clone Migration Results 160
6.4 Evaluating Colorful Analysis 164
7 Conclusion 167
Bibliography 169
A Colorful Examples 181

LIST OF FIGURES

Figure 1 An instance of the single-variant e-commerce example. 12
Figure 2 An instance of the single-variant e-commerce example improved with a customized theme. 13
Figure 3 An instance of the single-variant e-commerce example considering facts. 16
Figure 4 A counterexample for an expected assertion of the single-variant e-commerce example. 17
Figure 5 Single-variant e-commerce model in Alloy. 20
Figure 6 Concrete syntax of the Alloy language. 21
Figure 7 Syntax of the core Alloy language. 25
Figure 8 Semantics of formulas. 27
Figure 9 Semantics of relational operators. 27
Figure 10 Inference rules for bounding types. 29
Figure 11 Simplified e-commerce specification in Alloy. 30
Figure 12 Kodkod problem corresponding to the simplified e-commerce. 30
Figure 13 Transition system of a vending machine. 37
Figure 14 The pre- and post-state of the cancel operation. 40
Figure 15 The pre- and post-state of the cancel operation in the Electrum Analyzer. 43
Figure 16 Feature diagram typical notation. 46
Figure 17 Cross-tree constraints. 46
Figure 18 Feature diagram of the e-commerce platform. 47
Figure 19 E-commerce FM encoded in TVL. 49
Figure 20 Language categorization based on variability representation. 56
Figure 21 E-commerce catalog SPL encoded in Clafer. 60
Figure 22 Feature diagram of the vending machine example. 61
Figure 23 Vending machine SPL encoded in Electrum. 62
Figure 24 Base vending machine model in fSMV. 64
Figure 25 Feature Cancel of the vending machine in fSMV. 65

Figure 26 Feature MultiSelection of the vending machine in fSMV. 66
Figure 27 Derivative feature for Cancel and MultiSelection of the vending machine in fSMV. 67
Figure 28 Base model of the e-commerce catalog in FeatureAlloy. 67
Figure 29 Feature Thumbnails of the e-commerce catalog in FeatureAlloy. 68
Figure 30 Feature Categories of the e-commerce catalog in FeatureAlloy. 68
Figure 31 Derivative feature for Categories and Thumbnails of the e-commerce catalog in FeatureAlloy. 69
Figure 32 Core class diagram for e-commerce. 70
Figure 33 Class diagram Δ-Model for Categories. 70
Figure 34 Class diagram Δ-Model for Multiple. 70
Figure 35 Vending machine in fPromela with Free, Cancel and MultiSelection features. 74
Figure 36 Annotated class diagram for the e-commerce catalog. 75
Figure 37 Annotated activity diagram for the vending machine. 76
Figure 38 FTS of a vending machine. 78
Figure 39 Projected transition systems for the vending machine. 80
Figure 40 SMV model for the amalgamated verification of the vending machine example. 81
Figure 41 Clone Alloy model of an e-commerce platform with thumbnails. 84
Figure 42 Possible result of merging clones with the approach proposed by (Rubin et al., 2015). 85
Figure 43 Excerpt of Berkeley DB with background colors (Feigenspan et al., 2013). 90
Figure 44 CIDE screenshot (Kästner et al., 2008). 91
Figure 45 FeatureCommander screenshot (Kästner et al., 2008) 93
Figure 46 Concrete syntax of the Colorful Alloy language (additions w.r.t. the Alloy syntax are colored red). 94
Figure 47 Feature diagram of the colorful e-commerce specification. 96
Figure 48 E-commerce specification in Colorful Alloy. 97
Figure 49 Collecting a typing context from declarations. 100
Figure 50 Type rules for kernel paragraphs. 102
Figure 51 Type rules for kernel expressions. 103
Figure 52 Paragraph projection. 104

Figure 53 Expression projection. 105
Figure 54 E-commerce example projection to variant ①,②. 106
Figure 55 Paragraph translation into the amalgamated model with variability. 109
Figure 56 Expression translation into the amalgamated model with variability. 110
Figure 57 Amalgamated translation of the e-commerce model from Fig. 48 (except commands). 111
Figure 58 Amalgamated translation of the e-commerce model from Fig. 48 (commands). 112
Figure 59 Clone migration process of (Fenske et al., 2017) 115
Figure 60 E-commerce base model (variant ❶❷❸). 130
Figure 61 Clone introducing categories (variant ①❷❸). 130
Figure 62 Part of the initial migrated e-commerce colorful model. 131
Figure 63 E-commerce specification obtained with the automatic merging strategy. 137
Figure 64 The e-commerce in the Colorful Analyzer, showing the editor and the visualizer. 139
Figure 65 Automatic merge strategies. 141
Figure 66 Contextual refactoring menu Remove Feature. 142
Figure 67 Feature diagram of the e-commerce specification. 143
Figure 68 Feature diagram of the GrandpaFamily specification. 144
Figure 69 GrandpaFamily specification in Colorful Alloy. 145
Figure 70 Alloy4fun specification in Colorful Alloy. 146
Figure 71 Feature diagram of the Alloy4fun specification. 147
Figure 72 Feature diagram of the Graph specification. 148
Figure 73 Feature diagram of the Vending Machine example. 149
Figure 74 Snippet of Vending Machine specification in Colorful Alloy. 150
Figure 75 Feature diagram of the Bestiary specification. 151
Figure 76 Feature diagram of the Grandpa specification. 153
Figure 77 A snippet of variant ❶❷ of the GrandPa specification. 154
Figure 78 A snippet of variant ①❷ of the GrandPa specification. 154
Figure 79 A snippet of variant ①② of the GrandPa specification. 155
Figure 80 Feature diagram of the Ring Election specification. 156
Figure 81 A snippet of Ring Election specification, variant ❶. 157
Figure 82 A snippet of Ring Election specification, variant ①. 158

Figure 83	Feature diagram of the AdressBook specification.	158
Figure 84	A snippet of *AddressBook1h.als* specification, variant ❶❷.	159
Figure 85	A snippet of *AddressBook2e.als* specification, variant ①❷.	159
Figure 86	A snippet of *AddressBook3b.als* specification, variant ①②.	159
Figure 87	Feature diagram of the Hotel specification.	160
Figure 88	A snippet of the Hotel variant ❶❷.	160
Figure 89	A snippet of the Hotel variant ①❷.	161
Figure 90	A snippet of the Hotel variant ❶②.	161
Figure 91	A snippet of the Hotel variant ①②.	162
Figure 92	E-commerce specification in Colorful Alloy.	181
Figure 93	Graph specification in Colorful Alloy.	182
Figure 94	Vending Machine specification in Colorful Alloy (part 1).	183
Figure 94	Vending Machine specification in Colorful Alloy (part 2).	184
Figure 95	Bestiary specification in Colorful Alloy.	185
Figure 96	OwnGrandPa specification in Colorful Alloy.	186
Figure 97	Ring Election specification in Colorful Alloy.	187
Figure 98	AddressBook specification in Colorful Alloy.	188
Figure 99	Hotel specification in Colorful Alloy (part 1).	189
Figure 99	Hotel specification in Colorful Alloy (part 2).	190

LIST OF TABLES

Table 1 A bestiary of binary relations. 23

Table 2 Rules for translating FMs to propositional formulas. 52

Table 3 Evaluation results. 163

Table 4 Evaluation of the amalgamated and iterative approaches for the proactive examples. 164

Table 5 Evaluation of the amalgamated and iterative approaches for the extractive examples. 165

1

INTRODUCTION

Nowadays, computers have become increasingly powerful and essential in our life. Therefore, rigorous software design techniques for dependable systems become an imperative requirement in the software developing process. A simple straightforward approach to develop a new system is to write the requirements (usually in natural language) that specify the desired behavior of the system, and then attempt to implement it directly into code by the programmer. Finally, the functionality of the system can be tested during and after the coding to ensure that the generated program meets the specification and is free of errors or bugs. However, this process is not easy to achieve, especially for some safety-critical systems. First of all, a complete and unambiguous specification is difficult to have; second, programmers must fully and correctly understand the content of the requirements and handle all possible ambiguities; third, the testing must be exhaustive, that is, all possible states of the system must be rigorously tested. However, for some safety-critical systems such as medical devices or aerospace systems, we have to fully ensure its correctness, otherwise their operation may result in severe consequences. Furthermore, testing some large and complex systems can be time-consuming, and due to the size of the system, exhaustive testing may not even be possible in practice. Therefore, in addition to exhaustive testing, techniques based on mathematical analysis should be used to ensure the correct implementation of requirements. In particular, formal software development methods, including specification, verification, and validation, are mathematical techniques to identify errors and discrepancies in the early phases of the software development process.

Specifically, formal methods allow us to formally check that an implementation (or a formal model of an implementation) of a system (or parts of a system) meets the expected requirements (specified in some formal language). Among the myriad of the proposed formal methods, lightweight ones – which rely on automatic analyses to verify (often partial) specifications – have become increasingly popular, since they bring the power of fast verification and validation to most software developers. That is the case of model checkers like NuSMV (Cimatti et al., 2000) or SPIN (Holzmann, 1997), for verifying temporal logic properties of (behavioral) designs (modeled as transition systems), or model finders like Alloy (Jackson,

2012), more geared towards verifying first-order properties of structural designs specified at a high level of abstraction (using simple mathematical concepts like sets and relations). Given its popularity, lightweight analysis, and suitability for high-level design, Alloy will precisely be the focus of this thesis.

In order to suit the wide variety of (different) customer requirements, many systems are actually developed as a family of systems, many times with only slight differences in the implementation between different family members. Using conventional formal methods, each family member of those systems would be designed and analyzed individually. The engineers conducting the design have to perform tedious and repeated work, especially when a new functionality is added that affects all family members. Therefore, such classic single variant formal methods are not suitable for the development of software families. In this context, techniques for designing, analyzing, and implementing a full family of systems at once are mandatory. A common paradigm that is adopted when developing such large-scale software systems is *feature-oriented software development* (Apel and Kästner, 2009), a paradigm that organizes software around the key concept of *feature*, a unit of functionality that satisfies some of the requirements and that originates a configuration option. If the implementation is properly decomposed, it is possible to deliver many variants of the system just by selecting the desired features. The set of all those variants is usually called a *software product line* (SPL). Ideally, the design of SPLs should already explicitly take features into account, and formal methods should be adapted to support such *feature-oriented design* (Apel et al., 2010). In fact, even when developing a single software product, it is still convenient to explicitly consider features and multi-variant analysis during design to support the exploration of different implementation alternatives. In feature-oriented design, the set of all product line features together with their relationships should be specified in a feature model, and the system itself should be modeled in feature-aware extensions of common modelling formalisms such as transition systems. Having proper language and tool support for feature-oriented design is a key enabling technology for developing a wide variety of software systems of high quality in a fast, consistent, effective, and comprehensive way.

The key for effective variability modeling is to design a system family exploiting its commonalities and efficiently express and manage its variability. Most general-purpose programming and modeling languages can already somehow support feature-oriented design with their standard constructs in an *ad-hoc* fashion, a technique that is often cumbersome and error-prone for realistic SPLs. In addition to these ad-hoc techniques, proper feature-oriented programming languages fall into one of two categories: *compositional* approaches, which implement features as distinct modules and have some sort of module composition technique to generate a specific variant; and *annotative* approaches, which implement features with explicit (or sometimes implicit) annotations in the source code, that dictate which fragments will be present in

a specific variant. The former are well suited to support coarse-grained feature extensions, for example adding a complete new class to implement a particular feature, but are not good for fine-grained extensions, for example adding a sentence to a method or changing the expression in a conditional, to affect the way a code fragment works with different features (Kästner et al., 2008). Annotative approaches are much better suited for such fine-grained variability.

Unfortunately, explicit support for feature-oriented design in formal methods, providing a uniform formalism for feature, architectural and behavioral modeling, as advocated for SPL engineering (Schaefer and Hähnle, 2011), is still scarce. Support for features in model checking has been proposed, namely a compositional approach for the SMV modeling language of NuSMV (Plath and Ryan, 2001; Classen et al., 2014) and an annotative approach for the Promela modeling language of SPIN (Cordy et al., 2013). A compositional approach has also been proposed to explicit support features in Alloy (Apel et al., 2010). However, modeling and specifying in Alloy is usually done at high levels of abstraction, and adding a feature can require only minimal and very precise changes, for instance, adding one new relation to the model or changing part of the specification of a desired property, and such compositional approach is not well suited for these fine-grained extensions.

This thesis addresses precisely this problem. As such, the first main goal of our work is to propose an annotative approach to add explicit support for features to Alloy and its Analyzer. This involves developing a new language extension as well as adapting the existing or proposing novel analysis techniques. A classic annotative approach for source code is the use of `#ifdef` and `#endif` C/C++ compiler preprocessor directives to delimit the code fragments that implement a specific feature. Unfortunately, such annotation style obfuscates the code and makes it hard to understand and maintain, leading to the well-known `#ifdef` hell (Feigenspan et al., 2013). To alleviate this problem, while retaining the advantages of annotative approaches, Kästner et al. (2008) proposed to annotate code fragments associated with different features with different background colors, which was later shown to clearly improve SPL code comprehension and be favored by developers (Feigenspan et al., 2013). Inspired by this colorful approach, we developed a colorful extension to Alloy and its Analyzer, denoted *Colorful Alloy*, that allows users to annotate model and specification fragments belong to distinct features with different background colors, and run analysis commands to verify either a particular variant, or several variants at once. To the best of our knowledge, this is the first color-based annotative approach formal method for feature-oriented design focusing on structural requirements.

Evolution is an important software development activity, as the original design usually does not comply with new requirements. One approach widely used by software developers when developing new software

variants is the *clone-and-own approach*, where new variants are implemented by copying code from existing variants and then adapting it to fit the new requirements. Since the cost to maintain the clones and synchronize changes in replicas increases rapidly with the number of clones, developers may benefit from migrating (by merging) such variants into a single proper SPL implementation, where the common parts of the clones are factored out and implemented only once. The resulting code provides significant benefits for management and requires less effort in subsequent maintenance and design evolutions.

Many techniques have been proposed to migrate product variants into managed SPLs, as detailed in the survey conducted by Assunção et al. (2017). However, most of these techniques work at the code level, and only a few have been proposed specifically for design models. The second main goal of our work is to propose a technique for migrating legacy Alloy models developed with a clone-and-own approach into a single SPL Colorful Alloy model. We achieve this by a step-by-step automated refactoring technique. A refactoring is a kind of transformation that changes the structure of the source code while preserving its external behavior. However, classical refactoring is not well-suited for feature-oriented development, since both the set of possible variants and the behavior of each variant must be preserved (Schulze et al., 2012), and refactoring laws are typically too coarse-grained to be applied in this context, focusing on constructs such as entire functions or classes. In this thesis, we propose a catalog of variability-aware refactoring laws for Colorful Alloy, covering all model constructs – from structural declarations to axioms and assertions – and granularity levels – from whole paragraphs to formulas and expressions. Then, we show how these refactorings can be used to migrate a set of legacy Alloy clones into a colorful SPL using an approach similar to the one previously proposed by Fenske et al. (2017) for Java code. To simplify this process we also proposed a one-step fully automatic merging strategy that composes a sequence of refactorings.

The organization of this thesis is as follows.

Chapter 2 gives an overview of formal software design with Alloy. It starts by showing an application of the Alloy language in the design of a catalog structure of an e-commerce site. Then it gives a formal presentation of the language and, finally, presents some work on Alloy refactoring and extensions.

Chapter 3 presents a literature review on feature-oriented software design, including an overview about feature modeling, a collection of approaches that can be used to model and analyse feature-oriented systems, and some approaches for migrating clones into an SPL.

Chapter 4 formally presents the Colorful Alloy language, including its syntax, typing rules, and semantics. It also presents the multi-variant analysis technique developed for this new language extension.

Chapter 5 presents a catalog of variability-aware refactoring laws and shows how they can be used to migrate a set of cloned Alloy variants into a Colorful Alloy model.

Chapter 6 presents the implementation and evaluation of this work. We first present a description of the implementation of Colorful Alloy language, analysis, as well as the catalog of refactoring rules in the so-called Colorful Analyzer. The evaluation includes a qualitative analysis where we show how Colorful Alloy was used to develop several case-studies, and a quantitative analysis focused on evaluating the effectiveness of the proposed automatic analysis and clone migration techniques.

Chapter 7 summarizes the research in this thesis and identifies possible directions for future work.

Appendix A presents the Colorful Alloy models of the case studies used in the evaluation.

The work in this thesis originated two publications: the first (Liu et al., 2019) was accepted at the 5th International Symposium on Dependable Software Engineering Theories, Tools and Applications (SETTA), and describes a preliminary version of the Colorful Alloy language and analysis technique that are presented in Chapter 4; the second (Liu et al., 2020) was accepted at the 23rd Brazilian Symposium on Formal Methods (SBMF) and describes a preliminary version of the catalog of variability-aware refactoring laws and the clone migration technique presented in Chapter 5. The latter paper won the 2nd place for best paper award at the symposium and was invited to be submitted as a journal version to a special issue of Science of Computer Programming (Elsevier). We have recently submitted this paper, which presents the final version of the Colorful Alloy language and of the clone migration technique.

2

FORMAL SOFTWARE DESIGN WITH ALLOY

A good software design is key to achieve a high-quality system that meets all the expected requirements. In particular, it is very important to have clear specifications for the structure and behavior of a software system prior to its implementation. Among the various approaches proposed currently to help software developers reason about a software design, those that combine simple but formal specification languages with automatic analysis tools, that allow users to quickly explore different design alternatives and verify model assumptions, have more potential to be widely adopted. Alloy, which consists of a lightweight declarative modeling language and an automatic analysis tool, the Alloy Analyzer, is a prime example of such formal approaches and is becoming increasingly popular for validating software designs in the early stages of development.

In this chapter, we present the Alloy language and its Analyzer in detail, as well as how to use them for software design. Specifically, Section 2.1 illustrates the application of Alloy in the design of the catalog structure of a simple e-commerce system, including how to model the catalogue structure, how to specify its constraints, how to use the Analyzer to simulate different scenarios and check expected properties, as well as how to structure an Alloy model for reuse and better understanding. Section 2.2 gives a formal presentation of the Alloy language, including its syntax, core semantics, type inference mechanism, as well as the analysis procedure implemented in its Analyzer. Section 2.3 presents some work on refactoring Alloy specifications and, finally, Section 2.4 briefly presents some Alloy extensions and variants that have been developed to address some of its shortcomings.

2.1 Alloy by Example

Alloy is a lightweight declarative modeling language that provides a simple syntax to introduce sets and relations, as well as a powerful relational logic syntax to express constraints. A model written in Alloy can be automatically analyzed by the Alloy Analyzer, and the obtained results are displayed through a user-friendly

visualizer. In this section, we will step through the use of the Alloy language as well as its Analyzer by an example of an e-commerce platform, namely by modeling the catalog structure of such a system.

2.1.1 *E-commerce Example*

Nowadays, with the rapid development of computer network and communication technology, people can not only conduct face-to-face traditional transactions on tangible goods, but also through websites offering a wide range of products, supported by an advanced logistics and distribution system, and a secure payment system to complete transactions. This increasingly popular way of using the Internet to buy and sell goods or services is called *e-commerce* and it has begun to become an important part of everyday life.

There are many types of e-commerce systems, all of which provide a web interface, generally a HTML page, which displays the *catalog* of *products* offered by the company. Buyers can view a company's entire product offer through this catalog. The main purpose of the catalog is to advertise the products and thus attract the attention of the customer, so each product is typically illustrated by a set of *images* and catalogues are illustrated by *thumbnails*. An e-commerce platform can implement several variants of this basic catalogue structure, to support different needs from different e-commerce sites. For example, some sites organize products in *categories* listing together similar products, in which case, a catalog is organized as a set of categories, each containing some products. Also, in some sites categories may be organized in a *hierarchy*, and in some each product may belong to more than one category. This means that an e-commerce platform can be designed as an SPL, where each variant supports a different set of features. In this example a possible feature is having categories or not, and when this feature is present, two additional optional features are available: hierarchical categories and multiple categories per product. In this chapter we will focus on the design of a single variant of this system, the one having all the aforementioned features. Later in this thesis, we will see how the proposed Alloy extension allows us to develop all the five variants in a single model. In the following sections, we will detail how to model this simple example from scratch using the Alloy language, and how to analyze it with the Alloy Analyzer.

This example is an adaptation of a similar one proposed by Czarnecki and Pietroszek (2006) also in the context of SPL design.

2.1.2 *Signature and Field Declaration*

An Alloy model typically declares some *signatures*, each with an arbitrary number of *field* declarations. Signatures introduce sets of elements (also known as *atoms* in Alloy) and fields capture the relations between them. Everything in Alloy can be viewed as a *relation*, which is a set of tuples that relates atoms to each other. The *size* of a relation is the number of tuples it contains and the *arity* of a relation is the length of its tuples. Note that all tuples of a relation must have the same length. Relations with arity one, two, three are called unary, binary, and ternary, respectively. Those with an arity greater than two are called multi-relations. A set in Alloy is just a unary relation. In particular, signatures are unary relations.

A *signature* is declared with the keyword **`sig`** followed by a user-defined name and a body enclosed in braces. For example, the following code fragment introduces four signatures, which capture the main concepts of an e-commerce framework, namely the catalogs and the respective categories, the products provided by the company, and the images illustrating these products.

```
sig Catalog {}
sig Category {}
sig Product {}
sig Image {}
```

Each signature declaration can include an (optional) multiplicity constraint to limit the number of atoms in the respective set. For example, we could restrict `Product` to be non-empty to capture the fact that an e-commerce site must have some products for sale.

```
some sig Product {}
```

The **`some`** qualifier before the **`sig`** keyword states that signature `Product` has at least one atom. In addition to **`some`**, the **`lone`** or **`one`** qualifiers can also be used in signature declarations, indicating that at most one or exactly one atom must be present in the respective set.

A signature declaration can also introduce a number of fields separated by commas, each field relating the atoms of the enclosing signature with other atoms in the universe of elements. For instance, to declare fields to relate products with the respective images and catalogs, the declaration of `Product` could be changed as follows.

```
sig Product {
    images: set Image,
    category: some Category }
```

This declaration now introduces two binary relations, `images` and `category` that, respectively, associate each product with its set of images and categories. The **`set`** multiplicity imposed on `images` means that `images` relates each `Product` to any number of atoms from `Image` (possibly zero). The **`some`** quantifier imposed on `category` means that `category` relates each `Product` to at least one `Category`.

To capture the relation between catalogues and their thumbnails, and the hierarchy between categories, we introduce fields `thumbnails` and `inside` in the declarations of `Catalog` and `Category`, respectively.

```
sig Catalog {
    thumbnails: set Image
}
sig Category{
    inside: one Catalog+Category
}
```

Notice that `inside` relates each category with either one category or a catalog, to capture the fact that a category can be inside a catalog or inside another category. In this field declaration, + denotes set union, and is one example of the relational logic operators that will be introduced in Section 2.1.5 for specifying constraints on our model.

2.1.3 *Type System*

The image of a product may have a variety of formats, such as JPG or PNG. Alloy's type system allows users to specify a hierarchy between signatures. Similarly to other languages, a signature can be introduced as an extension of another signature (declared with an **`extends`** keyword). Sibling extension signatures represent disjoint subsets of the parent signature. The parent signature can optionally be marked as abstract (with the keyword **`abstract`**). An abstract signature has no elements other than those that belong to its extensions. For example, to introduce the different image formats in our running example, we could refine the declaration of images as follows.

```
abstract sig Image {}
sig Jpg,Png extends Image {}
```

By declaring `Image` as abstract, each atom of `Image` must belong to either `Jpg` or `Png`.

The format of images could be captured alternatively with an enumerated data type `Format`, that specifies the available formats, and a binary relation `format` that associates each image with its format. To declare an enumerated data type in Alloy, we declare it as a normal abstract signature, and extend it with as many singleton signatures as its different values. A singleton signature contains exactly one atom, and thus captures a possible value of the enumerated data type. The alternative specification of image formats would be as follows.

```
abstract sig Format {}
one sig Jpg,Png extends Format {}
sig Image {
    format : one Format
}
```

Besides extension, a signature can also be declared by inclusion (using the keyword **`in`** instead of **`extends`**), indicating the signature is an arbitrary subset of its parent signature. For instance,

```
sig Product { ... }
sig onSale in Product {}
```

declares `onSale` as a subset signature of `Product`, to capture products that are on sale. Unlike extension signatures, sibling subset signatures are not necessarily disjoint and cannot be extended.

Two non-overlapping signatures may have overloaded fields, declared with the same name. When an overloaded field is used in formulas and expressions, the Alloy type system will use context information (such as the type of the values to which it is being applied) to disambiguate which concrete field is being referred. If it cannot disambiguate, an error message will be generated to report an ambiguous reference. This is just an example of the possible type errors detected by Alloy's type system, which will be presented with more detail in Section 2.2.3.

For example, since `Product` and `Catalog` are disjoint, we could have used the same name `images` to associate a product with its images and a catalog with its thumbnails, as follows.

```
sig Product {
    images: set Image,
    category: some Category
}
sig Catalog {
    images: set Image
}
```

In the remaining of this section we will however still use the name `thumbnails` for the latter, to make the model more clear.

2.1.4 *Exploring Scenarios*

Having modeled the catalog structure of our e-commerce platform with several signature and field declarations, we could now perform some *scenario exploration* with the goal of validating our model. One way to achieve this is to instruct the Alloy Analyzer to depict a possible *instance* of our model with a **`run`** command. The command

```
run {} for 2
```

instructs the Analyzer to find any possible instance of our model (the empty body between brackets imposes no additional constraints on the desired instance), specifying a finite scope limiting the search for instances. Here the scope limits the top-level signatures (those that do not extend or are included in others) to have a maximum of 2 distinct atoms. The scope in the command can be omitted, in which case the Analyzer will use the default scope of 3. Alloy not only allows users to set a default scope for all top-level signatures (as shown in the above command) but also a more specific scope for particular signatures. Of course, these two settings can be mixed, in which case the explicit scope on a signature will override the default scope. As we have seen, the command `run {} for 2` instructs the analyzer to find possible instances where top-level signatures contain no more than two atoms. If the command is changed to `run {} for 2 but 3 Product`, the Analyzer will also consider instances where `Product` has up to 3 atoms. Scopes can further be constrained with the **`exactly`** keyword, meaning that the specified scope is now an exact bound. For example, the command `run {} for 2 but exactly 3 Product` will instruct the Analyzer to consider only instances with exactly three products.

Running the above command could return the instance shown in Fig. 1. The output can be displayed in various ways, namely, as text, a table, or a graph, which can be selected by corresponding buttons in the toolbar of the visualizer. Here the instance is depicted as a graph, where atoms are shown as nodes inside boxes and relations as arrows connecting those nodes. In the case of higher arity relations, arrows would connect the first and last atoms of its tuples, being the remaining atoms shown as labels. Figure 1 shows an instance with one product (`Product`) that is on sale, belongs to two different categories (`Category0` and `Category1`), which in turn belong to the same catalog (`Catalog1`). This product has a PNG image that is also the thumbnail of the respective catalog (`Image0`). There is another empty catalog

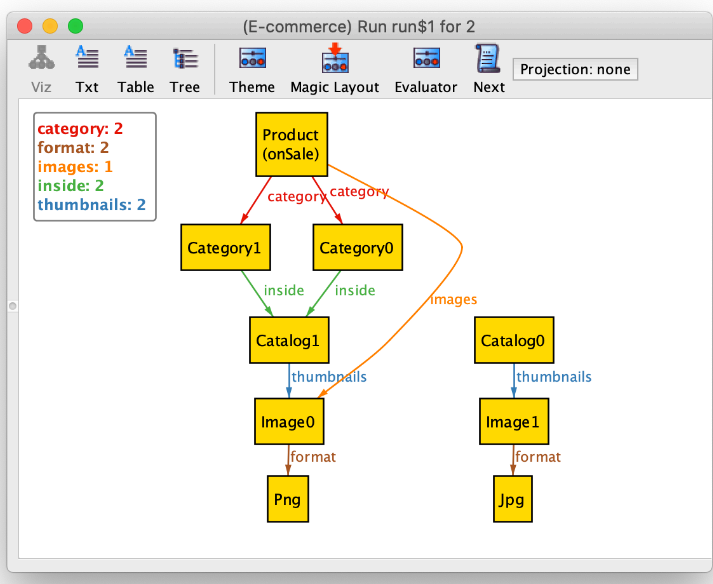


Figure 1: An instance of the single-variant e-commerce example.

(`Catalog0`) that however contains a thumbnail image (`Image1`): this clearly denotes a problem with our model, as catalogs should only have thumbnails that are images of the products they contain.

The visualizer allows users to ask for more instances by clicking on the *Next* button. Note that the Alloy analyzer runs a symmetry breaking procedure during the analysis, which avoids the generation of isomorphic instances (equal modulo renaming of atoms): this not only significantly improves the efficiency of analysis, but also considerably lowers the cognitive burden needed to understand scenarios, as the user is not encumbered with lots of instances that only differ in irrelevant details.

To further simplify the understanding of scenarios, the appearance of an instance can be customized in the *Theme* menu of the visualizer. As shown in Fig. 2, distinct signatures can be configured with different shapes and colors to highlight the important information. Moreover, relations can be configured to be shown as an attribute of atoms, instead of using arrows (as is the case of `format`) and some signatures can even be hidden (as is the case of `Jpg` and `Png`) to avoid cluttering visualizations with redundant or irrelevant information.

By exploring scenarios with **`run`** commands we identified a problem with our model. This problem cannot be avoided by changing only signature declarations. Additional constraints are needed in the model to eliminate this error, which motivates us to explore how constraints are specified in Alloy.

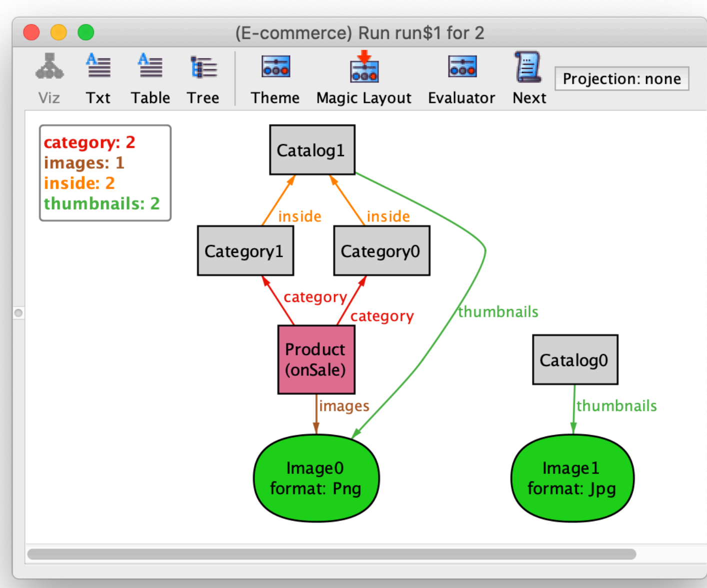


Figure 2: An instance of the single-variant e-commerce example improved with a customized theme.

2.1.5 *Specifying Constraints*

Additional constraints in Alloy are expressed with *relational logic*, a logic that combines the quantifiers of the first-order logic with the operators of the relational calculus. Since all structures in Alloy are represented as relations, all structural properties can be expressed with a small set of simple but powerful operators. In addition to the declared signatures and constraints, there are three pre-defined constants, representing the set of all atoms (**`univ`**), the empty set (**`none`**), and the binary identity relation that relates every atom to itself (**`iden`**).

The operators in Alloy fall into two categories: set operators and relational operators. The standard set operators have the conventional meaning, and can be applied to relational expressions of any arity. The *union* of two sets (denoted by +) returns a new set that contains all the tuples that are in at least one of them; the *intersection* (&) returns a new set with the tuples contained in both sets; and set *difference* (–) determines the tuples that belong to the first but not to the second. To compare sets we can use the binary operators **`in`**, that returns true if the first set is a subset of (or equal to) the second one, and = that returns true only when the two sets are the same. It is also possible to check the cardinality of a set with the multiplicity unary operators **`no`**, **`lone`**, **`some`**, and **`one`**, with the obvious meaning.

Among the relational operators we have the *product* (→) of two relations, that returns all the tuples that result from the combination (via concatenation) of all tuples from the first relation with all tuples of the

second one. The *transpose* unary operator (~) can be applied to a binary relation and reverses the order of the elements in all its tuples. A common use of transpose is the constraint `~r in r`, indicating that the relation `r` is symmetric. Given a set and a relation, we can filter the tuples of the latter that start with an atom in the former with the *domain restriction* operator (`<:`). Likewise, given a relation and a set, we can filter the tuples of the former that end with an atom in the latter with the *range restriction* operator (`:>`). For example, in our running example, `onSale <: images` would return the relation that associates `onSale` products with their images, and `inside :> Catalog` would restrict relation `inside` to categories directly inside a catalog.

One of the most important relational operators in Alloy is *join* (`.`), also known as *composition*. When composing two relations, all the tuples from the first are joined with all tuples of the second. The join of two tuples succeeds if the last atom of the first matches the first atom of the second, and, in that case, the matching atoms are dropped and the resulting tuples are concatenated. Given a relation `r` with arity n and a relation `s` with arity m, their composition `r.s` is possible if $n + m - 2 > 0$, being the arity of the composed relation precisely $n + m - 2$. For example, `images.format` is the binary relation that associates products with the formats of their images. It is possible to compose a set (possibly a singleton representing a scalar) with a relation of arity higher than 1. For example, `onSale.category` is a set that contains the categories of all products that are on sale and `format.Jpg` is the set of all images with format JPG.

The *transitive closure* `^r` of a binary relation `r` is the smallest relation on that contains `r` and is transitive. A binary relation `r` is transitive if `r.r in r` (hence, `^r = r + r.r + r.r.r + ...`). For example, the expression `^inside` associates a category with all the categories or catalogs that directly or indirectly contain it. The *reflexive transitive closure* of a binary relation `*r` is the smallest relation that contains `r` and is both transitive and reflexive, that is `*r = ^r + iden`.

As presented above, atomic formulas are either inclusion (or equality) tests (with **`in`** and =) or cardinality checks (with **`no`**, **`lone`**, **`some`**, or **`one`**). These atomic formulas can be combined with the standard boolean operators, namely conjunction (**`and`**, &&), disjunction (**`or`**, ||), implication (**`implies`**, ⇒), equivalence (**`iff`**, ⇔), and negation (**`not`**, !). It is also possible to write quantified expressions as

$$Q\,x : e \mid \phi$$

where $Q \in \{$ **`all,some,no,lone,one`** $\}$ is a quantifier, x is a variable, e is a unary expression denoting a set, and ϕ is a constraint that contains the variable x. In particular, expression **`all`** $x : e \mid \phi$ means that constraint ϕ is satisfied for every element in e. Multiple variables can be quantified together.

For instance **some** x, y **:** e | ϕ means that there are some values in e for variables x and y that make ϕ hold.

The *let* expression allows one to factor out repeated subexpressions, thus making a specification shorter. The let expression

$$\textbf{let}\ x = e \mid A$$

is equivalent to expression A with each occurrence of variable x replaced by the assigned expression e. A let expression is mostly used to simplify complex expressions or give meaningful names for some subexpressions.

In Alloy, constraints that must hold in the model are expressed inside a **fact** paragraph. Constraints in different lines of a fact are implicitly conjoined. In order to easily distinguish the constraints related to different requirements or relations, they can be split in different facts and given different names, although the Analyzer will ignore these names during the analysis. In terms of the analysis, it is no different from having a single fact that contains every constraint, and the order in which facts appear is also irrelevant.

We can now solve the problem that was identified during scenario explanation, by adding a fact stating that all thumbnails in a catalog must be images of products contained in it.

```
fact Thumbnails {
    all c:Catalog | c.thumbnails in (category.(^inside).c).images
}
```

In fact `Thumbnails`, the expression `c.thumbnails` retrieves all thumbnails in catalog `c`, and expression `(category.(^inside).c).images` retrieves all images of the products in `c`, by first determining all categories inside catalog `c` with expression `^inside.c`, then all products inside all these categories by navigating backwards with relation `category`, and finally all images of these products by navigating forward with relation `images`.

Rerunning the *run* command can yield an instance as shown in Fig. 3, that no longer has the identified problem, and where a single product belongs to two categories inside two different catalogs.

2.1.6 *Verifying Assertions*

To validate the design of our e-commerce system we should now verify some expected properties. Properties that are expected to hold should be enclosed in a named assertion paragraph, declared with the **assert** keyword. For example, the assertion

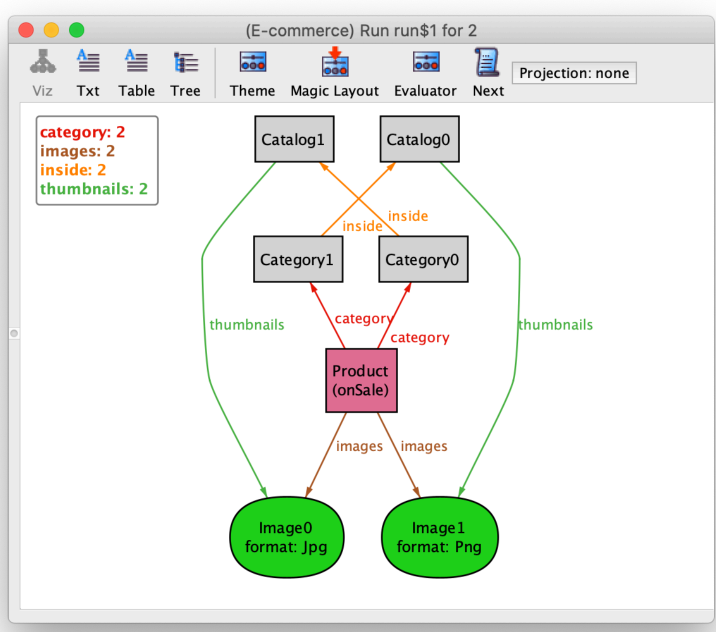


Figure 3: An instance of the single-variant e-commerce example considering facts.

```
assert AllCataloged {
    all p:Product | some (p.category.^inside & Catalog)
}
```

formulates the expectation that every product is contained inside a catalog. In this assertion, `p.category` determines all categories that directly contain product `p`, while `p.category.^inside` determines all categories or catalogs that might contain those (directly or indirectly): by intersecting with `Catalog` we require that at least one of these is a catalog.

An assertion can be verified with a **`check`** command, that instructs the analyzer to find a counterexample for the specified property. Similarly to **`run`** commands, a scope that limits the number of atoms that can be contained in top-level signatures can be given after the name of the assertion to be verified, with a default of 3 if none is given. By issuing command **`check`** `AllCataloged` **`for`** `2`, a counter-example is found, meaning our assertion is not valid. The counterexample is shown in Fig. 4, and depicts a category that is contained in itself. This counterexample exposes a problem with our specification. To fix this problem we can add the fact

```
fact Acyclic {
    all c:Category | c not in c.^inside
}
```

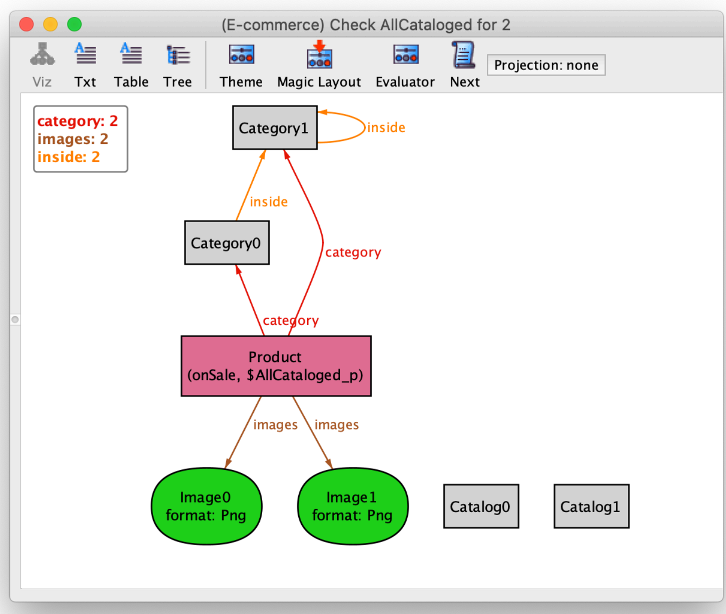


Figure 4: A counterexample for an expected assertion of the single-variant e-commerce example.

that enforces the `inside` relation to be acyclic.

Rerunning the **`check`** command gives no counterexample. Notice however that the assertion is being checked considering only a maximum of 2 atoms for each signature. To increase the confidence in the analysis, we can execute the command with a larger scope, for example **`check`** `AllCataloged` **`for`** `8`. Now the Analyzer takes more time to finish the verification but still does not find a counterexample. Although we can not conclude that the assertion is valid, as the analysis is always bounded, it gives us some assurance concerning the validity of the desired property.

Actually, there is no fundamental difference between running a predicate and checking an assertion in the analysis process. A **`check`** command is equivalent to a **`run`** command that searches for an instance that satisfies the negation of the constraints in an assertion.

2.1.7 *Modularization*

There may be some expressions that one might want to factor out of constraints, for example to improve readability and reusability when they occur repeatedly in the model. To support that, Alloy allows the declaration of auxiliary predicates and functions.

A predicate (declared with **pred** keyword) declares a reusable constraint. Its declaration consists of the name of the predicate, some optional argument declarations inside square brackets, and a block with formulas. A function (declared with **fun**) declares a reusable expression. Its declaration consists of the name of the function, some optional argument declarations inside square brackets, a return type after a colon, and a relational expression inside a block. The result of a predicate is a Boolean value, while the result of a function is a value of its return type. Predicates and functions are called by providing an expression for each parameter.

In our running example we can declare an auxiliary function that calculates the catalogs of a product as follows.

```
fun catalog [p : Product] : Catalog {
    p.category.^inside & Catalog
}
```

The function receives a `Product` as parameter and returns a set of catalogs as a result. Using this function, the assertion `AllCataloged` could be redefined as follows.

```
assert AllCataloged {
    all p: Product | some catalog[p]
}
```

A typical usage for predicates is to encode different specific scenarios, for example to act as (kind of) unit tests for a model (examples that the user knows to be valid instances). For example, the following predicate `Scenario` will only be true in instances where at least one product has images. A **run** command can then directly ask for an instance satisfying the predicate. If the predicate had parameters, the command would automatically instantiate them when searching for a satisfying instance.

```
pred Scenario {
    some Product.images
}
run Scenario for 2
```

In addition to predicates and functions, Alloy also has a module system that allows the modularization and reuse of models. A module describes a model that can be used as a submodel, imported by another with an open statement before all paragraphs. There are a number of utility modules, pre-packaged with the Analyzer, that provide common operations. In particular, module `util/relations` declares common

operations for computing the domain and the range of a binary relation, as well as a collection of typical constraints one might want to impose on those. For instance, it contains a predicate `acyclic` that constraints a relation `r` to be acyclic over a given set. This predicate is declared in the `util/relations` module as follows.

```
pred acyclic[r: univ→univ, s: set univ] {
    all x: s | x !in x.^r
}
```

Here we see that Alloy allows predicates and functions to receive as parameters arbitrary relations and sets, and not just singletons. The constant **`univ`** represents the universe of all atoms, and thus a value of type **`univ→univ`** is any binary relation. By declaring parameter `s` to be of type **`set univ`** we state that `s` is any subset of **`univ`**, that is, an arbitrary set. Using this module, the fact `Acyclic` could be expressed as follows.

```
open util/relation
fact Acyclic {
    acyclic[inside,Category]
}
```

The `util/ordering` module is one of the most popular modules pre-packaged with Alloy. This is a parameterized module that can be used to impose a total order over a signature. It provides functions for capturing the total order structure, such as the first element (`first`), the last (`last`), the predecessor relation (`prev`), or the the successor relation (`next`), and predicates for reasoning about its elements, for example to check whether a given element precedes another (`lt`). For example, we could have used this module to impose a total order on a signature that represents the star ratings that customers give to the products.

```
open util/ordering[Rating]
some sig Rating {}
sig Product {
    rating : one Rating
}
```

The final model of the running example described in this section is shown in Fig. 5. This model has no ratings and includes an extra restriction, requiring all catalogues to display at least one thumbnail of all on sale products contained in them.

```
open util/relation

sig Product {
    images: set Image,
    category: some Category
}
sig onSale in Product {}

sig Image {
    format: one Format
}

abstract sig Format {}
one sig Jpg, Png extends Format {}

sig Catalog {
    thumbnails: set Image
}

sig Category {
    inside: one Catalog+Category
}

fact Thumbnails {
    all c:Catalog | c.thumbnails in (category.(^inside).c).images
}

fact OnSaleThumbnails {
    all p:onSale, c:catalog[p] | some p.images & c.thumbnails
}

fact Acyclic {
    acyclic[inside,Category]
}

pred Scenario {
    some Product.images
}
run Scenario for 2

fun catalog [p : Product] : Catalog {
    p.category.^inside & Catalog
}
assert AllCataloged {
    all p:Product | some catalog[p]
}
check AllCataloged for 2
```

Figure 5: Single-variant e-commerce model in Alloy.

```
spec         ::= [ moduleDecl ] import* paragraph*
moduleDecl   ::= module qualName [ [ name,+ ] ]
import       ::= open qualName [ [ qualName,+ ] ] [ as name ]
paragraph    ::= sigDecl | factDecl | funDecl | predDecl | assertDecl | cmdDecl
sigDecl      ::= [ abstract ] [ mult ] sig name,+ [ sigExt ] { decl,* } [ block ]
sigExt       ::= extends qualName | in qualName [ + qualName ]*
mult         ::= lone | some | one
decl         ::= [ disj ] name,+ : [ disj ] expr
factDecl     ::= fact [ name ] block
assertDecl   ::= assert [ name ] block
funDecl      ::= fun name [ [ decl,* ] ] : expr block
predDecl     ::= pred name [ [ decl,* ] ] block
expr         ::= const | qualName | @name | this | unOp expr | expr binOp expr
                | expr arrowOp expr | expr [ expr,* ] | expr [ ! | not ] compareOp expr
                | expr ( ⇒ | implies ) expr else expr | let letDecl,+ blockOrBar
                | quant decl,+ blockOrBar | ( expr ) | block | { decl,+ blockOrBar }
const        ::= none | univ | iden | Int
unOp         ::= ! | not | no | mult | set | # | ~ | * | ^
binOp        ::= || | or | && | and | ⇔ | iff | ⇒ | implies | + | − | & | ++ | <: | :> | .
arrowOp      ::= [ mult | set ] → [ mult | set ]
compareOp    ::= in | =
letDecl      ::= name = expr
block        ::= { expr* }
blockOrBar   ::= block | | expr
quant        ::= all | no | sum | mult
cmdDecl      ::= [ check | run ] [ qualName ] ( qualName | block )  [ typeScopes ]
typeScopes   ::= for number [ but typeScope,+ ] | for typeScope,+
typeScope    ::= [ exactly ] number qualName
qualName     ::= [ this/ ] ( name/ )* name
```

Figure 6: Concrete syntax of the Alloy language.

2.2 Formal Presentation of the Language

Up to this point, we have presented a rough idea of the Alloy language with the simple e-commerce catalog example. In this section we will give a formal presentation of this language in terms of its syntax, semantics, type inference, and analysis.

2.2.1 *Formal Syntax*

The syntax of Alloy is presented in Fig. 6 using standard BNF operators: `x*` means zero or more repetitions of symbol `x`; `x+` means one or more repetitions of `x`; `x | y` represents a choice of `x` or `y`; `[x]` denotes an optional `x`. In addition, `x,*` means zero or more occurrences of the `x` separated by commas, and `x,+` means one or more occurrences of such comma-separated `x`. The `name` symbol represents a string identifier. Similar to other programming languages, Alloy identifiers may include the alphabetic characters, numbers (except in the first character), underscores and quotation marks, and are case sensitive. Alloy's reserved keywords are displayed in bold type.

An Alloy model consists of one or more files, one of which contains a main module that either directly or indirectly imports (through the use of **open**) modules in the other files. A module consists of an optional module header, possibly some imports, and a set of paragraphs, each either a signature, predicate, or function declaration, a fact, an assertion, or an analysis command. The paragraphs can appear in the module in any order. There is no requirement of declaration before the use.

The module header provides the relative path and filename of the model and can only appear in the first line of the model. Similar to the Java language, the path is relative to the directory that the importing module is located. Every model that wishes to use a module must have an explicit import statement following the optional header (and before any signatures, paragraphs, or commands), with a simple **open** statement. A model can import multiple modules, each of which requires a separate **open** statement that provides the respective paths and names, instantiation of its parameters (if any), and an optional alias. The order of those import statements does not matter. Each parameter of the imported module must be instantiated by a signature, which can be any declared signature or one of the predefined signatures such as **Int** (a signature containing all integers) or **univ**. Each imported module can be referenced in the model either by its alias (if given) or by its module identifier. A module can be imported more than once but must declare aliases if their parameters are instantiated differently.

A declaration introduces one or more variables (on the left of the colon) with a bounding expression (on the right side of the colon) constraining their value. Declarations can be used inside signature declarations for declaring fields, to introduce quantified variables, in the definition of sets (or relations) by comprehension, and to declare the arguments of predicates and functions. The keyword **disj** stands for disjoint, indicating that the variables or declared relations are somehow disjoint (their intersection is empty). For example, `sig A {disj r,s : e}` declares two disjoint binary relations `r` and `s`, while the declaration `sig A {r: disj e}` declares that the `r` values of different members of signature `A` are disjoint. To give an example of a relation defined by comprehension consider the following expression that defines a binary relation that maps each product to the respective JPG images:

```
{p : Product, i : Image | i in p.images and i in Jpg}
```

Expressions can denote either Boolean expressions (formulas) or relational expressions. Formulas are used, for example, in facts, assertions, and predicates. Signature declarations can also be followed by a so called *signature fact*, an implicitly universally quantified formula that somehow restricts the atoms of that signature. In signature facts, **this** denotes the implicitly universally quantified variable, any mention of a field declared in the signature is implicitly projected over **this**, and if one wishes to override this

Table 1: A bestiary of binary relations.

Property	Multiplicity constraint
Entire	`r in A → some B`
Simple	`r in A → lone B`
Surjective	`r in A some → B`
Injective	`r in A lone → B`
Function	`r in A → one B`
Injection	`r in A lone → one B`
Surjection	`r in A some → one B`
Bijection	`r in A one → one B`

behavior and refer to all fields, its name should be preceded by the special marking @. For example, fact `Thumbnails` could have been declared as a signature fact of `Catalog` as follows:

```
sig Catalog {
    thumbnails: set Image
} {
    thumbnails in (category.(^inside).this).images
}
```

Atomic formulas can either be inclusion (**in**) or equality (=) checks between relational expressions of equal arity, or multiplicity checks over unary relational expressions (with **lone**, **some**, **one**, or **no**). It is also possible to check the multiplicity of both endpoints of a relational expression with a special arrow notation: formula `r in A m → n B`, where `m` and `n` are multiplicity checks, is a shorthand notation for `all a : A | n a.r` and `all b : B | m r.b`. This notation can be used to quickly restrict the shape of a binary relation, as presented in Table 1. For example, a relation r that maps each element from a set A to at most one element of B is known as *entire* and if it maps to at least one element of B is known as *simple*. A relation that maps at least one element of A to each element of B is *surjective* and one that maps at most of one element of A to each element of B is *injective*. A relation is a *function* if it is both simple and entire. An injective function is called an *injection* and a surjective function is called a *surjection*. A function is a *bijection* if is both injective and surjective.

Atomic formulas can be composed with the standard logical operators and with quantifiers. There are two forms of each logical operator: a shorthand (!, ||, &&, ⇒, ⇔) and a verbose form (**not**, **or**, **and**, **implies**, **iff**). The two form are completely interchangeable.

As seen in the previous section, relational expressions are built with both set and relational operators. The operators +, −, & are the standard set operators of union, difference, and intersection, respectively.

As for relational operators we have product (->), transpose (~), domain and range restriction (`<:` and `:>`), transitive and reflexive closure (`^` and `*`), and join (`.`). There exists an alternative notation for (dot) join, known as *box join*: expression `p[s]` is the same as `s.p`, but box join has lower precedence that dot join, hence, `p.q[s]` is equal to `(p.q)[s]`. The override operator (++) is used to replace some tuples in a relation: the expression `p ++ q` is the union of `p` and `q` after removing from `p` the tuples that starts with an element in the domain of `q`.

Alloy has limited support for integers. Signature **`Int`** contains all the integers present in the universe. Integers are represented in two's complement notation, and the scope of **`Int`** in a command sets the numbers of bits used in the representation. Module `util/integer` provides some arithmetic functions and predicates to work with integers. For example `add` is a binary function that adds two integers. All arithmetic operations work in the usual way for two's complement representation, namely with wrap-around semantics. The cardinality operator (`#`) computes the number of tuples in a relation. There is also a special quantifier **`sum`** that can be used to sum the integers somehow associated with all the atoms of a unary expression. Integers should be used very carefully in Alloy due to the wrap-around semantics. For example, if the scope for **`Int`** is not chosen correctly the operator `#` can return the wrong cardinality of a set. Fortunately, when developing abstract models, integers can usually be replaced by structures with less semantics, such as signatures with total orders (imposed with module `util/ordering`). An alternative semantics for integers with the goal of preventing arithmetic overflows has been proposed in (Milicevic and Jackson, 2014), but it is also rather cumbersome. As such, in this thesis we will not address integers in Alloy.

2.2.2 *Formal Semantics*

After expanding predicates, functions and modules, an Alloy model basically consists of a collection of relation declarations (signatures and fields inside a signature) and a collection of facts and analysis commands. For a given command, the meaning of a model is a set of instances, each of which assigns a value to each of the declared relations, makes the facts true, and makes the formula in the command true or false, depending if the command is a run or a check, respectively. In practice, a check command can be replaced by a run command with the negated assertion, and instead of putting assertions in facts, they can just be conjoined in the command. Hence, an Alloy model can essentially be regarded just as a set of signatures and field declarations together with one run command containing the constraint one wishes to make true.

```
alloyModule ::= sigDecl* cmdDecl
sigDecl     ::= sig name [ sigExt ] { decl,* }
sigExt      ::= [ abstract ] extends name | in name [ + name ]*
decl        ::= name : [set] expr
form        ::= expr in expr | not expr | expr and expr | all decl | form
expr        ::= name | const | unOp expr | expr binOp expr  | { decl,+ | form }
const       ::= none
unOp        ::= ~ | ^
binOp       ::= + | & | − | . | →
cmdDecl     ::= run { form } for typeScope,+
typeScope   ::= [ exactly ] number name
```

Figure 7: Syntax of the core Alloy language.

In this section, we will explain the semantics of the Alloy language in terms of such simplified core language, where a model consists only of declarations and a single run command. This core language also does not include some operators that can be defined with others, and does not allow multiplicity restrictions in declarations. It is much smaller than the full Alloy language, but it captures its essential semantics, and it is quite easy to translate the full language into it, as we will briefly explain below. The syntax of the core language is presented in Fig. 7.

To translate an Alloy model to this core language, we can start by in-lining all imported modules, and expand all predicate and function definitions at the point they are called. Signature facts can be converted to normal facts by quantifying universally. Multiplicities in declarations can also be trivially replaced by facts. For example, declaration

```
some sig Product {}
```

is equivalent to

```
sig Product {}
fact {some Product}
```

And the multiplicity in the declaration of `category` in signature `Product`

```
sig Product {category: some Category}
```

can be replaced with

```
sig Product {category: set Category }
fact {all p:Product | some p.category }
```

All the facts can then be eliminated and the respective assertions conjoined with the formula in the run command. Operators not included in the core syntax can be replaced by their definition. For the logic operators we have the usual definitions, for example, ϕ **or** ψ can be replaced by **not (not** ϕ **and not** ψ) and **some** x : Φ | ϕ by **not all** x : Φ | **not** ϕ. Likewise for the missing relational constants and operators, for example, **univ** can be replaced by the union of all top-level signatures, **iden** can be defined by comprehension as {x, y : **univ** | x=y}, and A <: Φ can be replaced by Φ & (A → **univ**). In the core syntax we also assume the run command to have a scope for every type signature, namely every top level (not extending or included in other) or extension signature, but not inclusion signatures, which cannot be given a scope. Obtaining such scopes from the original scope in command involves some non-trivial computations which are detailed in (Jackson, 2012, Appendix B).

An instance M can be viewed as an assignment of a set of tuples of atoms to all declared relational variables (signatures and fields). A valid instance M should respect all declarations and scopes, and satisfy the formula ϕ inside the run command. The satisfaction of a formula ϕ in a model M is denoted by $M \models \phi$, and the value of a relational expression Φ in a model M is denoted by $[\![\Phi]\!]_M$.

An instance M respects the declaration of a type signature A with scope k if $|M[A]| \leq k$, or $|M[A]| = k$ if the scope is exact. All tuples in the valuation of a signature must be of arity one, that is $\forall t \in M[A] \cdot |t| = 1$. If A is a top-level signature then for any other top-level signature B we have $M[A] \cap M[B] = \emptyset$. If A is extended by signatures A_1 to A_n then we have $M[A_1] \cup \ldots \cup M[A_n] \subseteq M[A]$, or $M[A_1] \cup \ldots \cup M[A_n] = M[A]$ if A is abstract, and for all $1 \leq i < j \leq n$ we have $M[A_i] \cap M[A_j] = \emptyset$. For an inclusion signature B contained in A we have that $M[B] \subseteq M[A]$. An instance M respects the declaration of field r declared inside signature A with bounding expression Φ if $M[r] \subseteq [\![A \rightarrow \Phi]\!]_M$.

The definition of $M \models \phi$ is presented in Fig. 8 and the definition of $[\![\Phi]\!]_M$ is presented in Fig. 9. In these definitions x is an identifier (a variable, signature, or field), ϕ and ψ are formulas, Φ and Ψ are relational expressions, and a and b are atoms. Given an instance M, $M \oplus x \mapsto X$ represents the instance where the value of x is set to X, and all other variables keep the same value. Given the informal presentation of the language given before we believe all definitions are self-explanatory. To simplify, comprehension semantics is presented just for the case of unary relations.

$$
\begin{array}{lll}
M \models \Phi \ \mathtt{in}\ \Psi & \text{iff} & [\![\Phi]\!]_M \subseteq [\![\Psi]\!]_M \\
M \models \mathtt{not}\ \phi & \text{iff} & M \not\models \phi \\
M \models \phi \ \mathtt{and}\ \psi & \text{iff} & M \models \phi \wedge M \models \psi \\
M \models \mathtt{all}\ x : \Phi \mid \phi & \text{iff} & \forall \langle a \rangle \in [\![\Phi]\!]_M \cdot M \oplus x \mapsto \{\langle a \rangle\} \models \phi
\end{array}
$$

Figure 8: Semantics of formulas.

$$
\begin{array}{lll}
[\![x]\!]_M & = & M[x] \\
[\![\mathtt{none}]\!]_M & = & \emptyset \\
[\![\sim\Phi]\!]_M & = & \{\langle a_2, a_1 \rangle \mid \langle a_1, a_2 \rangle \in [\![\Phi]\!]_M\} \\
[\![\hat{}\Phi]\!]_M & = & [\![\Phi]\!]_M \cup [\![\Phi.\Phi]\!]_M \cup [\![\Phi.\Phi.\Phi]\!]_M \cup \cdots \\
[\![\Phi{+}\Psi]\!]_M & = & [\![\Phi]\!]_M \cup [\![\Psi]\!]_M \\
[\![\Phi\&\Psi]\!]_M & = & [\![\Phi]\!]_M \cap [\![\Psi]\!]_M \\
[\![\Phi{-}\Psi]\!]_M & = & [\![\Phi]\!]_M \setminus [\![\Psi]\!]_M \\
[\![\Phi.\Psi]\!]_M & = & \{\langle a_1, \ldots, a_{n-1}, b_2, \ldots, b_m \rangle \mid \langle a_1, \ldots, a_n \rangle \in [\![\Phi]\!]_M \wedge \langle b_1, \ldots, b_m \rangle \in [\![\Psi]\!]_M \wedge a_n = b_1\} \\
[\![\Phi{\rightarrow}\Psi]\!]_M & = & \{\langle a_1, \ldots, a_n, b_1, \ldots, b_m \rangle \mid \langle a_1, \ldots, a_n \rangle \in [\![\Phi]\!]_M \wedge \langle b_1, \ldots, b_m \rangle \in [\![\Psi]\!]_M\} \\
[\![\{x : \Phi \mid \phi\}]\!]_M & = & \{\langle a \rangle \mid \langle a \rangle \in [\![\Phi]\!]_M \wedge M \oplus x \mapsto \langle a \rangle \models \phi\}
\end{array}
$$

Figure 9: Semantics of relational operators.

2.2.3 *Type Inference*

As we have seen in Section 2.1.3, the type system of Alloy supports subtyping, through signature extension, and overloading of relation names. Alloy includes a type inference mechanism to detect type errors. The goal of this mechanism, first proposed by Edwards et al. (2004), is to detect *irrelevant expressions*, expressions that can be replaced by an empty relation without affecting the value of its enclosing constraint. This notion of type error is quite different from the one in programming languages, but arguably more adequate for a formal modelling language. An example of a trivial irrelevant expression is `Product.format` in our running example. This expression is always empty in any possible valid instance, since products never appear as first atoms in the tuples contained in binary relation `format`. To clarify the difference between Alloy's type errors and those of a typical object-oriented programming language, let's assume that our running example had the following additional signature, capturing products sold in bundles.

```
sig Bundle extends Product {
    products: some Product
}
```

Likewise in a programming language, in Alloy we can access a field from any atom in a subtype. For example, given a bundle `b` we can determine its images as `b.images`. However, unlike in a programming

language, in Alloy we can also access a field from any atom in a supertype. For example, expression Product.products does not raise a type error, since it is not irrelevant, as some products might be bundles and thus this expression may denote a non-empty set.

In order to simplify the type-inference mechanism, the type of an expression in Alloy is defined in a canonical form, which eliminates subtype comparisons from the type system. This is achieved by flattening the type hierarchy, and expressing types in terms of the so-called *atomic types*, which includes all types signatures that are not supertypes (those not extended by others) plus for each non abstract supertype, a special atomic type known as its *remainder type*, with the same name prefixed with $\$$, that contains all its atoms that do not belong to any of its subtypes. For example, the remainder type of Product, denoted by $Product, contains all products that are not bundles.

Following Alloy's motto that "everything is a relation", types in Alloy are relations defined with the atomic types as atoms, i.e. sets of tuples of atomic types. The type A of an expression Φ is in a sense an upper-bound on its value: any tuple that appears in $[\![\Phi]\!]_M$ must belong to the Cartesian product of the atomic types in one of the tuples in its type. For example, in our running example, the type of Format is {<Jpg>,<Png>}, the type of Product is {<Bundle>,<$Product>} and the type of images is {<Bundle,Image>,<$Product,Image>}.

The type inference mechanism presented in (Edwards et al., 2004) works in two-phases: by proceeding bottom-up in the abstract syntax tree a first approximation of the type, denoted *bounding type*, is computed; then this type is refined in a second top-down phase in order to compute the so-called *relevance type*. The fact that an expression Φ has bounding type A is denoted by $\Gamma \vdash \Phi : A$, being Γ a typing context containing the types of all declared signatures and relations. The inference rules for bounding types are shown in Fig. 10. After type inference any expression (except **none**) that has an empty type is reported as erroneous. Notice how, due to the fact that types are relations, the same relational operators of the expressions being typed are used to compute their types. To give some examples, expression Bundle.images is not irrelevant since its type is {<Image>}. However Product.format is irrelevant because Product has type {<Bundle>,<$Product>}, format has type {<Image,Jpg>,<Image,Png>}, and composing both types yields an empty type.

The presentation of the inference rules for relevance types is omitted, since currently the Alloy Analyzer seems to not implement them fully. The resolution of overloading is also based on this type inference mechanism. An overloaded relation is represented as the union of all relations that share the same name. After type inference only one of them should have a non-empty type (only one should be relevant), otherwise an ambiguity error is reported. This type system is sound but not complete: expressions that reported

$$\frac{\Gamma[x] = A}{\Gamma \vdash x : A} \qquad \frac{}{\Gamma \vdash \textbf{none} : \emptyset} \qquad \frac{\Gamma \vdash \Phi : A}{\Gamma \vdash \sim\Phi : \sim A} \qquad \frac{\Gamma \vdash \Phi : A}{\Gamma \vdash \hat{}\Phi : \hat{}A}$$

$$\frac{\Gamma \vdash \Phi : A \quad \Gamma \vdash \Psi : B}{\Gamma \vdash \Phi + \Psi : A + B} \qquad \frac{\Gamma \vdash \Phi : A \quad \Gamma \vdash \Psi : B}{\Gamma \vdash \Phi \& \Psi : A \& B} \qquad \frac{\Gamma \vdash \Phi : A \quad \Gamma \vdash \Psi : B}{\Gamma \vdash \Phi - \Psi : A - B}$$

$$\frac{\Gamma \vdash \Phi : A \quad \Gamma \vdash \Psi : B}{\Gamma \vdash \Phi . \Psi : A . B} \qquad \frac{\Gamma \vdash \Phi : A \quad \Gamma \vdash \Psi : B}{\Gamma \vdash \Phi \rightarrow \Psi : A \rightarrow B} \qquad \frac{\Gamma \vdash \Phi : A \quad \Gamma \oplus x \mapsto A \vdash \phi}{\Gamma \vdash \{x : \Phi \mid \phi\} : A}$$

$$\frac{\Gamma \vdash \Phi \& \Psi : A}{\Gamma \vdash \Phi \texttt{ in } \Psi} \qquad \frac{\Gamma \vdash \phi}{\Gamma \vdash \texttt{not } \phi} \qquad \frac{\Gamma \vdash \phi \quad \Gamma \vdash \psi}{\Gamma \vdash \phi \texttt{ and } \psi} \qquad \frac{\Gamma \vdash \Phi : A \quad \Gamma \oplus x \mapsto A \vdash \phi}{\Gamma \vdash \texttt{all } x : \Phi \mid \phi}$$

Figure 10: Inference rules for bounding types.

erroneous are indeed irrelevant, however, it is possible to write an irrelevant expression which is not identified as an error.

2.2.4 *Analysis*

The Alloy Analyzer uses Kodkod (Torlak and Jackson, 2007), an efficient SAT-based relational model finder, as its analysis engine. An Alloy model is automatically translated into a Kodkod problem and then solved by an off-the-shelf SAT solver such as MiniSat[1] or Glucose[2]. SAT abbreviates *satisfiability*, the problem of determining whether there exists a model (an assignment of values to propositional variables) of a Boolean formula that makes it true.

A Kodkod problem consists of a universe declaration, a set of relation declarations, and a relational logic formula in which the declared relations are free variables. The universe declaration specifies the set of atoms that can be used to build models. Each relation declaration specifies its arity and two bounds: a lower bound, that contains tuples that *must* be present in the relation; and an upper bound that contains tuples that *may* be presented in the relation.

The translation of an Alloy model to a Kodkod problem is for the most part relatively straightforward, as the supported relational logic is the same. The problem formula is obtained by conjoining all facts (including explicit fact paragraphs and implicit facts in declarations) with the constraints in the run command to be analyzed (or with the negation of the constraints the command is an assertion). The interesting part is the

1 http://minisat.se
2 https://www.labri.fr/perso/lsimon/glucose/

```
sig Product {
    images: set Image
}
abstract sig Image {}
sig Jpg,Png extends Image {}

fact { all m : Image | m in Product.images }
run { some Product.images } for 2
```

Figure 11: Simplified e-commerce specification in Alloy.

```
{A1,A2,A3,A4}

Product :1 {} {<A1>,<A2>}
Jpg :1 {} {<A3>,<A4>}
Png :1 {} {<A3>,<A4>}
images :2 {} {<A1,A3>,<A1,A4>,<A2,A3>,<A2,A4>}

no Jpg & Png and
all p : Product | p.images in Jpg+Png and
images.univ in Product and
all m : Jpg+Png | m in Product.images and
some Product.images
```

Figure 12: Kodkod problem corresponding to the simplified e-commerce.

translation of the type hierarchy and scopes. We will illustrate this translation with a very simplified version of our e-commerce example, presented in Fig. 11. The resulting Kodkod problem is presented in Fig. 12.

With a scope of 2, a valid instance of the Alloy model could have at most 2 products and 2 images, but in some instances these could be all JPGs or all PNGs. If different atoms where declared for all atomic types we would need a total of 6 atoms, even if a valid instance could have at most 4. With larger scopes and larger type hierarchies this approach could lead to large inefficiencies. Instead, in the translation to Kodkod only 2 atoms are created for each top-level signature. In the case of `Image` the 2 atoms will be allowed in the upper-bound of both `Jpg` and `Png`, to allow all possible instances[3]. However, to respect the Alloy semantics, an implicit constraint will be added to ensure that the two signatures are disjoint. As such, the first of line of the Kodkod problem in Fig. 12 declares a universe of 4 elements. Then only the 3 atomic type signatures are declared, all with arity 1 and empty lower bound. Two of the atoms are chosen for the upper bound of `Product` and the other two for the upper bound of `Jpg` and `Png`. Then the binary relation `images` is declared. The first constraint in the problem ensures that `Jpg` and `Png` are disjoint. The second and third ensure that the range and domain of `images` are contained in `Image`

3 Note that the atom names in Kodkod are irrelevant and are renamed by the Alloy Analyzer when translating back to depict the instance.

and `Product`, respectively. Note that the upper bound of `image` is not sufficient to ensure this, as the final value of the signatures might be smaller then their upper bound. Finally we have the constraint in the fact and the constraint in the run command. Since signature `Image` is not declared it is replaced by the union of all its extensions.

The translation of a Kodkod problem to SAT starts by representing each relation r of arity k by a k-dimensional matrix $|n|^k$

$$r[i_1,...,i_k] = \begin{cases} \top & if\langle A_{i_1},...,A_{i_k}\rangle \in r_l \\ x_{i_1,...,i_k} & if\langle A_{i_1},...,A_{i_k}\rangle \in r_u \backslash r_l \\ \bot & otherwise \end{cases}$$

where n is the number of universe elements, r_l and r_u represent the lower and upper bound of r, respectively, $i_1,...,i_k \in \{0..n-1\}$, and $x_{i_1,...,i_k}$ is a fresh Boolean variable. Each element of the matrix represents a possible tuple in the relation: if the tuple is in the lower bound then it must be present, if it is not in upper bound then it cannot be present, otherwise a variable is created whose value will be determined by the SAT solver. Then, for each relational expression, the respective matrix is obtained by performing matrix operations on its sub-relation matrices. The relational operators have counter-parts in terms of matrix operations. For example, the union operator will translate to sum and the join operator is translated as the product of matrices. Finally, for atomic formulas, an inclusion check will be translated by the conjunction of element-wise implication, and a **some** multiplicity check by the disjunction of all elements in the matrix. Finally all Boolean connectives are included, universal quantifiers are expanded to conjunctions, and existential quantifiers are skolemized for efficiency reasons.

For example the matrix corresponding to the relational expression `Product.images` will be computed as follows.

$$\begin{bmatrix} p_1 & p_2 & \bot & \bot \end{bmatrix} . \begin{bmatrix} \bot & \bot & i_{1,3} & i_{1,4} \\ \bot & \bot & i_{2,3} & i_{2,4} \\ \bot & \bot & \bot & \bot \\ \bot & \bot & \bot & \bot \end{bmatrix}$$

In the vector representing `Product`, variables p_1 and p_2 will decide if atoms `A1` and `A2` are included. In the matrix representing `images` we have four variables to decide which of the possible 4 tuples are present. The formula corresponding to constraint **some** `Product.images` would be

$$(p_1 \wedge i_{1,3}) \vee (p_2 \wedge i_{2,3}) \vee (p_1 \wedge i_{1,4}) \vee (p_2 \wedge i_{2,4})$$

The resulting Boolean formula is finally passed to a SAT solver. If the solver finds no solution, the Analyzer will generate a message saying that no instance or counterexample is found, otherwise, the solution will be mapped back to an instance, to be graphically depicted.

In order to decrease SAT complexity, Kodkod performs a variety of optimizations before sending a formula to the SAT solver. The most significant one is symmetry breaking. Since atom names are meaningless, many instances would in fact be isomorphic (modulo renaming). Kodkod detects which atoms can be considered symmetric based on the declaration of bounds and generates a lex-leader symmetry breaking predicate that is conjoined to the problem's formula. This not only filters out isomorphic instances from users, but also improves the efficiency of analysis by reducing the search space. In our example problem, the universe can be partitioned into two sets of symmetric atoms, namely `{A1,A2}` and `{A3,A4}`. This means that, for example, an instance with no images and a single product `A1` can be generated, but the isomorphic instance with the single product `A2` will not.

2.3 Refactoring Alloy Models

Refactoring, an essential activity in programming, is a technique that enables code quality improvements (for example, improve readability or efficiency) without changing its external behaviour. Refactoring of design models is also important and could provide similar benefits. For Alloy in particular, a catalogue of refactoring bidirectional transformations has been proposed in (Gheyi and Borba, 2004; Gheyi, 2007). To support more application scenarios, such as refining a model by introducing additional signatures or relations, they use a more flexible notion of equivalence between the models (before and after applying the refactoring), based on bidirectional refinement. The semantics of the models is only compared for a relevant set of signatures and fields Σ. The value of entities not in this set is considered irrelevant and is not compared. Moreover, for elements in Σ that are not present in both models, a view function v must be defined, stating how the value of an element in one model can be interpreted using elements of the other.

Each refactoring law consists of two templates (patterns) of equivalent Alloy models (according to the above notion), on the left-hand and right-hand sides. A refactoring can be applied whenever one template of the law is matched with a model and some pre-conditions are met. In the following laws, `ps` represents a collection of paragraphs, `frms` stands for a set of formulas, `ds` denotes a set of relations, `n` refers to identifiers, and `exp` denotes a relational expression. After each law some pre-conditions are provided for its application to be possible. The symbol $(\rightarrow)$ indicates a pre-condition that is only required when applying the law from left to right, while $(\rightarrow)$ indicates a pre-condition that is required when applying the

law in the opposite direction. A pre-condition marked with ($\leftrightarrow$) is necessary in both directions. All the proposed laws were formalized and proved sound using the PVS theorem prover[4].

We will now present some of the refactoring laws proposed by Gheyi (2007).

2.3.1 *Laws for Signatures*

Law 1 says that we can add or remove an empty signature that is not used elsewhere. The proviso in the bottom of the law states that the name of the new signature can not be declared in the model. Notice that in this case the equivalence between the two models does not consider the value of the new signature.

Law 1 (Introduce signature)**.**

```
ps
```

$=_{\Sigma,v}$

```
ps
sig n {}
```

provided

($\leftrightarrow$) (1) n is not in Σ; (2) for all names in Σ that are not in the resulting model, v must have exactly one valid item for it;

($\rightarrow$) `ps` does not declare any signature named n;

($\leftarrow$) n does not appear in `ps`.

Law 2 allows us to remove the *abstract* quantifier, and replace it by a constraint. Notice that the number of sub-signatures has no constraint and can be any number greater than zero. Moreover, it can be applied when n extends another signature and there are no pre-conditions for its application.

Law 2 (Remove abstract qualifier)**.**

```
ps
abstract sig n { ds }
sig n1 extends n { ds1 }
    ...
sig nn extends n { dsn }
```

$=_{\Sigma,v}$

```
ps
sig n { ds }
sig n1 extends n { ds1 }
    ...
sig nn extends n { dsn }
fact { n = n1 + ··· + nn }
```

Law 3 introduces a generalization between two or more signatures. We can always remove (add) a parent signature that is not used in other places of the model by applying the law from right to left (left to right). If the new signature is to be considered in the equivalence then it must be defined accordingly in the view.

4 https://pvs.csl.sri.com

Law 3 (Introduce generalization)**.**

ps **sig** n_1 { ds_1 } … **sig** n_n { ds_n }	$=_{\Sigma,v}$	ps **sig** n {} **sig** n_1 **extends** n { ds_1 } … **sig** n_n **extends** n { ds_n } **fact** { n = $n_1 + \cdots + n_n$ }

provided

($\leftrightarrow$) If n belongs to Σ, v must contain the view n $\mapsto$ $n_1 + \cdots + n_n$;

($\rightarrow$) ps does not declare any paragraph named n;

($\leftarrow$) n does not appear in ps, ds_1, ..., ds_n.

Law 4 allows us to remove (or add) multiplicity qualifiers for signatures (possibly extending other signatures). No pre-conditions are required for this law. Note that this law together with Law 2 enables the removal (or adding) of multiplicity qualifiers from abstract signatures.

Law 4 (Remove signature multiplicity qualifier)**.**

ps mlt **sig** n { ds }	$=_{\Sigma,v}$	ps **sig** n { ds } **fact** { mlt n }

where

mlt $\in$ {**one**, **lone**, **some**}.

2.3.2 *Laws for Fields*

Law 5 enables the introduction of a new field named n_0 whose value is determined to be exp. Similarly, we can also remove fields that is not being used, by applying the law from right to left. The family of a signature is the set of signatures that extend or are extended by it direct or indirectly. The first pre-condition in ($\rightarrow$) is necessary because Alloy does not allow two overloaded fields in the same family.

Law 5 (Introduce relation)**.**

ps
sig n {
    ds
}
	$=_{\Sigma,v}$	ps
		sig n {
		    ds,
		    n_0 : **set** T
		}
		fact { n_0 = exp}

provided

($\leftrightarrow$) If n_0 belongs to Σ then n_0 does not appear in exp and v contains the view $n_0 \mapsto$ exp;

($\rightarrow$) The family of n in ps does not declare any field named n_0; T is a signature name declared in ps or is S; n_0 does not appear in exp or exp is n_0; exp is well-typed in the resulting model;

($\leftarrow$) n_0 does not appear in ps.

Similarly to signatures, the multiciplicity qualifiers in a field can be removed, and replaced with a fact by applying Law 6 from left to right.

Law 6 (Remove field multiplicity qualifier)**.**

ps
sig n {
    ds,
    n_0: mlt T
}
	$=_{\Sigma,v}$	ps
		sig n {
		    ds,
		    n_0 : **set** T
		}
		fact { **all** x:n | mlt x.n_0 }

where

mlt $\in$ {**one**, **lone**, **some**}.

Fields declared together can always be separated with Law 7, by applying it from left to right.

Law 7 (Separate field declarations)**.**

ps
sig n {
    ds,
    n_0, ⋯, n_k : **set** T
}
	$=_{\Sigma,v}$	ps
		sig n {
		    ds,
		    n_0 : **set** T,
		    ⋯
		    n_k : **set** T
		}

A final example of a refactoring law for relations is Law 8, which allows to replace an expression in a field declaration by its type.

Law 8 (Replace field expression)**.**

Left		Right

ps
sig n {
    ds,
    n_0 : **set** exp
}
	$=_{\Sigma,v}$	ps
		sig n {
		    ds,
		    n_0 : **set** T
		}
		fact { **all** x : n | x.n_0 **in** exp' }

provided

(↔) exp has the type T, which is a signature declared in ps or is n; exp′ replaces each reference to a field relation r of n (whether declared or inherited) not prefixed by @ by x.r, and every occurrence of **this** by x.

(←) n_0 does not appear in exp.

2.3.3 *Laws for Formulas*

Besides laws for signatures and relations, there are also some rules proposed for refactoring formulas.

Law 9 states that we can add or remove a formula that can be deduced from other formula specified in the fact of a specification. This law offers a way for remove redundant constraints in Alloy. Since the condition of Law 9 is not syntactic, an extra mechanism must be used to guarantee the its satisfaction. For example, we can use Alloy itself for checking this condition, using a check command to verify if the formula f holds.

Law 9 (Introduce formula)**.**

Left		Right

ps
fact n {
    frms
}
	$=_{\Sigma,v}$	ps
		fact n {
		    frms
		    f
		}

provided

(↔) f can be deduced from the facts in ps and frms.

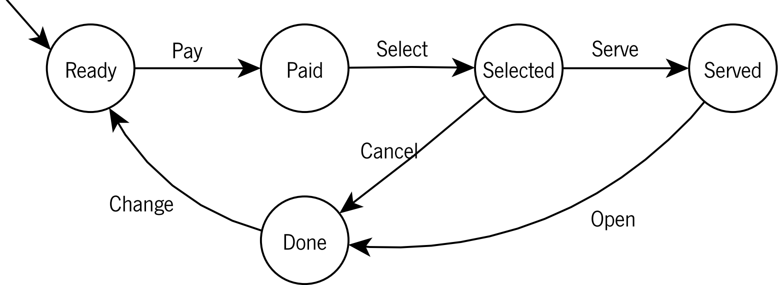


Figure 13: Transition system of a vending machine.

Law 10 allows the introduction of an empty fact provided there is no name conflicts. Since facts cannot be referred to by other paragraphs, the elimination of an empty fact is always possible. A similar law can be defined for introducing or removing empty signature facts.

Law 10 (Introduce empty fact)**.**

`ps`	$=_{\Sigma,v}$	`ps` **`fact`** `n {}`

provided

(→) `ps` does not declare any paragraph named `n`.

2.4 Alloy Extensions

In order to achieve distinct goals such as verifying behavioral requirements when designing software systems or improving scenario exploration, a number extensions to (or variants of) Alloy have been proposed. Among them, the Electrum (Macedo et al., 2016; Brunel et al., 2018) extension was proposed for the analysis of dynamic systems with rich configurations, and will soon be included in the official version 6 of the Alloy language. In plain Alloy, the value of all signatures and field relations is immutable. If one wants to model variable relations, it is necessary to introduce an explicit time signature, totally ordered to simulate an execution trace, and all relations whose value might change must include this signature as an extra column in their declaration.

As a simple example, let us consider a simple vending machine system adapted from the family of vending machine systems used to introduce feature transition systems (FTSs) (Classen et al., 2010). A basic vending machine system takes a coin, returns change, selects a soda, serves the soda, and eventually opens a compartment for the customer to take their soda. Three additional features were added to the

basic model: selling *Tea*; allowing buyers *Cancel* their purchase; offering *Free* drinks. Here we consider one of the variants of this system, just with the feature cancel, and its behavior is depicted in Fig. 13 with a transition system. After paying, the user can select one of the products in stock. Then if the selection is not canceled, the product is served, and the door opens, so that the user can collect the selected product. After collecting the product or canceling the selection, the change is returned to the user.

To model this vending machine in plain Alloy we need to declare singleton signatures for the possible states of the machine, and a signature for the possible products.

```
abstract sig State {}
one sig Ready, Paid, Selected, Served, Done extends State {}

sig Product {}
```

As mentioned above, to model variable relations and sets, we can to introduce an explicit `Time` signature, and inside it declare fields for all relations that are supposed to be mutable. By doing this, we allow them to have different values in different `Time` instants. In this case we would like to have four mutable relations: `state`, that captures the state the machine is in, `stock`, that contains the available products, `selection`, that contains the selected product, and `balance`, the amount of coins collected by the machine (assuming a price of one coin per product). A total ordering should also be imposed over `Time`, so that time instants form a trace. For `balance` the standard library `util/natural` is used: this library declares a signature `Natural` to represent natural numbers.

```
open util/ordering[Time] as time
open util/natural as nat

sig Time {
  state : one State,
  stock : set Product,
  selection : lone Product,
  balance : one Natural
}
```

Operations can be modeled with predicates that relate the value of mutable relations at a given `Time` instant (parameter `pre` below) with their value at the next one (`pos`). For example, operation `Select` can be specified as follows.

```
pred Select [pre,pos : Time, p : Product] {
  pre.state = Paid and p in pre.stock
  pos.state = Selected
  pos.selection = p
  pos.stock = pre.stock
  pos.balance = pre.balance
}
```

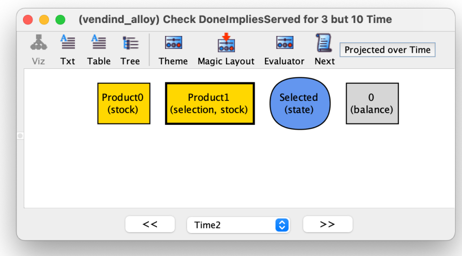
In the first line we have the pre-condition for this operation to occur: the machine should be in state `Paid` and the selected product should be in `stock`. Then we have the effect of the operation: the state will change to `Selected` and the product will be added to `selection`. Finally we have the frame-condition that specifies what does not change, in this case the `stock` and `balance`.

Having specified the operations, it is necessary to axiomatize what are valid traces. In this case, a valid trace should begin with `state` set to `Ready`, no `selection`, some `stock`, and zero `balance`. Then all transitions between consecutive time instants should result from one of the specified operations. This behaviour can be specified in the following **`fact`**.

```
fact Behaviour {
  first.state = Ready
  no first.selection
  some first.stock
  first.balance = Zero
  all pre : Time, pos : pre.time/next {
    Pay[pre,pos] or (some p : Product | Select[pre,pos,p]) or Cancel[
   pre,pos] or
    Serve[pre,pos] or Open[pre,pos] or Change[pre,pos]
  }
}
```

Temporal properties can be specified by quantifying explicitly over the time signature. For example, to specify that every time the machine is in the `Served` state the balance must be strictly positive we could write the following assertion.

```
assert Balance {
  all t : Time | t.state = Served implies gt[t.balance,Zero]
}
check Balance for 3 but 10 Time
```



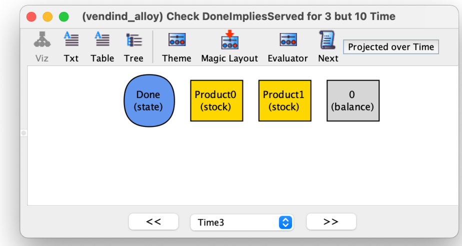


Figure 14: The pre- and post-state of the cancel operation.

This property is valid (here we are checking it with traces of size 10). As another example, to specify that every time the machine is in the `Done` state previously it must have been in the `Served` state, we could write the following assertion.

```
assert DoneImpliesServed {
  all t : Time | t.state = Done
      implies some u : t.*time/prev | u.state = Served
}
check DoneImpliesServed for 3 but 10 Time
```

Notice the use of `t.*time/prev` to compute all time instants that occurred up to instant `t`. Checking this property yields an obvious counter-example, where the user cancels the selection and thus is not served a product. To better understand trace counter-examples, we could use the projection feature of the visualiser, that allows the user to focus on a particular atom of a signature at a time. In this case, by projecting on the `Time` signature we can see one time instant at a time. Figure 14 shows the two consecutive time instants corresponding to the `Cancel` operation (with the theme already configured to hide irrelevant information).

Assertion `DoneImpliesServed` is an example of a safety property. Special care must be had to properly perform bounded model checking of liveness properties, as counter-examples must be infinite traces (Cunha, 2014). The need to explicitly model time and traces in plain Alloy ends up often being a tedious, time-consuming, and error-prone task, in particular if we need to verify liveness properties, where back loops must also be considered to represent infinite traces.

In contrast, Electrum offers an implicit notion of time and allows both signatures and fields to be declared as mutable using the keyword **`var`**. Instances (traces) are infinite (looping) sequences of time instants. The non-variable signatures and fields are still static, meaning that their value remains fixed during the evolution of the system. Facts and assertions can be specified using Linear Temporal Logic operators (including past ones), such as **`always`** or **`eventually`**, and expressions can be evaluated in the next

state by marking them with `'`. The Electrum Analyzer supports both bounded-model checking with a translation to Kodkod and complete model checking with a translation to SMV model checkers, such as NuSMV[5] or nuXmv[6].

In the vending machine example, instead of declaring them inside the explicit `Time` signature, the declaration of the three mutable sets could be done simply as follows.

```
var one sig state in State {}
var sig stock in Product {}
var lone sig selection in Product {}
var one sig balance in Natural {}
```

By using `'`, the specification of operations is also much simplified (and do not need to be explicitly parametrized with the time instants). For example, the `Select` operation could be specified as follows.

```
pred Select [p : Product] {
  state = Paid and p in stock
  state' = Selected
  selection' = p
  stock' = stock
  balance' = balance
}
```

The specification of the `Behaviour` fact is also much simpler, resorting to the **`always`** temporal operator to restrict the valid transitions.

```
fact Behaviour {
  state = Ready
  no selection
  some stock
  balance = Zero
  always {
    Pay or (some p : Product | Select[p]) or Cancel or
    Serve or Open or Change
  }
}
```

5 https://nusmv.fbk.eu
6 https://nuxmv.fbk.eu

Likewise, the specification of the desired properties can be simplified, due to the use of the temporal logic operators. In the case of the assertion `DoneImpliesServed` we could use both **`always`** and the **`once`** past operator, that checks if previously some formula was valid.

```
assert Balance {
  always (state = Served implies gt[balance,Zero])
}
check Balance for 3 but 10 steps
assert DoneImpliesServed {
  always (state = Done implies once state = Served)
}
check DoneImpliesServed for 3 but 10 steps
```

The Electrum Analyzer also depicts traces and counter-examples graphically, allowing the user to focus on any pair of consecutive states, which are depicted side by side, as show in Fig. 15. In this case the property was checked with the bounded-model checking engine with the trace length limited to 10 steps. This engine ensures the length of counter-examples to be minimal; these 4 steps suffice before the trace repeats itself indefinably.

Several other Alloy extensions and variants have been proposed to simplify the specification and analysis of dynamic systems. DynAlloy (Frias et al., 2005, 2007) is an Alloy variant that uses dynamic logic (instead of temporal logic) to specify behaviour, limited to the specification of safety properties, but with a strong emphasis on achieving an efficient verification mechanism. In (Near and Jackson, 2010) Alloy is extended with standard imperative programming constructs like sequential composition or loops: imperative specifications can be annotated with pre- and post-conditions, and analysis is limited to check if such an annotated specification can hold for some execution path or holds in all possible execution paths. In (Chang and Jackson, 2006) a model checking framework is proposed, where systems can also be modeled with mutable relations and imperative programming constructs, but where expected properties are expressed in CTL temporal logic and verified symbolically with a BDD-based technique. More recently, the DASH language has been proposed to specify systems with state machines (Serna et al., 2017). It uses Alloy syntax to declare and reason about system states, and to specify declarative transitions within a control state hierarchy.

A key distinctive feature of Alloy is its good support for scenario exploration, by supporting iteration to explore alternative instances and their graphical depiction for easier comprehension. Some techniques have been proposed to provide more control on how iteration over instances works, instead of the random iteration provided by standard Alloy. In particular, Aluminum (Nelson et al., 2013) is a tool built on top of

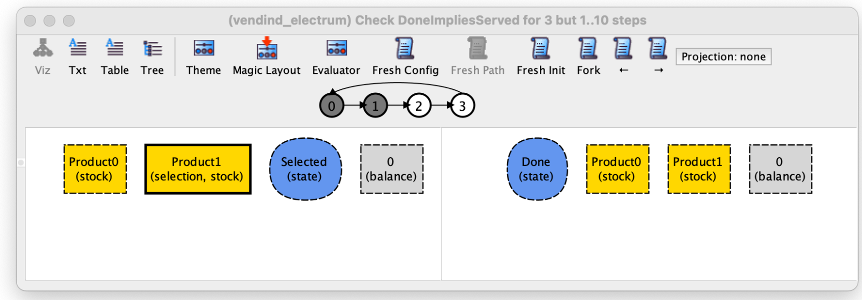


Figure 15: The pre- and post-state of the cancel operation in the Electrum Analyzer.

Alloy focused on generating minimal scenarios, instances that contain no more atoms than those strictly necessary (no atom can be removed without making the instance invalid). The work by Macedo et al. (2015) extended this idea of controlling instance generation, but provides more instance generation and iteration operations, for example, it provides an iteration operation that generates a new instance that is as different as possible from the previous one. Several improvements to the Alloy instance visualization mechanism have also been proposed. For example, Zaman et al. (2013) extended the visualizer with more shapes that enable a better depiction of entities in the same type family (sub- or super-types), and also with a new atom positioning mechanism that fixes their position in different projections, simplifying the comprehension of instances of dynamic models. More recently, an extension of the visualizer has been proposed that focus precisely on improving the visualization of instances from dynamic models (Couto et al., 2018): this work allows the use of different layout and transition managers to, respectively, decide how atoms are positioned and how transitions between states are depicted. Instead of just the standard Alloy tree layout, atoms can positioned with a circular or grid layout, for example.

3

FEATURE-ORIENTED SOFTWARE DESIGN

An *SPL* (Pohl et al., 2005) aims at constructing a family of similar software systems that share common functionalities. Instead of developing each system from scratch individually, an SPL represents the entire system family in a single code base and each variant (usually called *product*) is instantiated by configuring and assembling the different components. *Feature-oriented software development* (Apel and Kästner, 2009) is a paradigm proposed for the construction, customization, and synthesis of large-scale software systems. It relies on the core concept of *feature*, a unit of functionality that implements some requirements, represents some software decisions, or provides a potential configuration option. In this approach many different software systems can be generated sharing a set of common features and differing in other features, resulting in a popular paradigm for the development of SPLs.

The development of an SPL can usually be split in two stages: *domain engineering*, where commonalities and variability points are identified and developed, and *application engineering*, where concrete products are derived from the SPL. Domain engineering itself encompasses three different activities (Apel and Kästner, 2009): *domain analysis*, *domain design*, and *domain implementation*. In feature-oriented development the focus is domain engineering, so that application engineering is mostly automated through the selection of features. Feature-oriented domain analysis (Kang et al., 1990) is a key activity in this process, producing a *feature model* (FM) encoding the available features and their dependencies. These are typically represented as *feature diagrams* (FD), a graphical tree structure that describes features as atomic units of difference. Both domain design and implementation then consider features as first-class artefacts. This explicit introduction of features in the design and implementation of the SPL usually follows either a *compositional* or an *annotative* approach. Several *feature-oriented software design* techniques have been proposed to support domain design in feature-oriented development, including variability-aware specification languages and validation and verification techniques, which are the main focus of this thesis.

The engineering of an SPL (Krueger, 2001) can be *proactive*, where feature-oriented development is adopted from the start and features are an integral part of the system from the early stages of development.

More commonly, however, development starts without variability concerns, resulting in the development of multiple software variants through *clone-and-own*. As the cost of maintaining and synchronizing the clones increases, developers often find benefits in merging the clones into a single SPL. A common challenge in feature-oriented development is thus the migration of legacy systems into an SPL – including the extraction of FMs, and feature-oriented designs and implementations –, an *extractive* approach to SPL engineering. In fact, even after migration, development may not be fully compliant with feature-oriented development, and new clones may need to be integrated back into an existing SPL – a *reactive* approach to SPL engineering.

This chapter presents an overview of existing work on feature-oriented software design. Section 3.1 starts with a brief overview about feature modeling during domain analysis, including the automatic analysis of FMs. Section 3.2 then shows how systems can be modelled in feature-oriented design, including and a list of languages classified into three approaches to handling features: annotative, compositional, or ad-hoc. Section 3.3 describes the related analysis techniques for these modelling approaches, particularly the model checking of designs with variability. Lastly, Section 3.4 presents existing approaches to the migration of design clones into a feature-oriented SPL.

3.1 Feature Modeling

One of the main activities of feature-oriented domain analysis is feature modeling, the identification of the available features and their dependencies. The result of this analysis is an *FM*, an artefact that is central in feature-oriented development and in the related variability-aware analyses.

The section starts in Section 3.1.1 by presenting the standard FM formalism, and some popular extensions, as well as languages for their specification. It then provides an overview of existing techniques for their analysis in Section 3.1.2.

3.1.1 *Specifying Feature Models*

An FM is commonly specified with an FD (Kang et al., 1990; Czarnecki and Eisenecker, 2000), basically a tree that captures the set of valid products in a system family. Each node of the tree represents a feature, and the edges represent the hierarchical relations (or dependencies) between features. A child feature can only be selected if its parent is also selected. Different relationships between a parent feature and its child features impose additional restrictions, which can be categorized as *mandatory* or *optional* features, or *xor-* or *or-groups* of features. The typical graphical representation of these relationships is shown in Fig. 16.

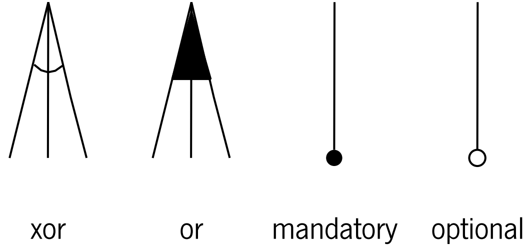


Figure 16: Feature diagram typical notation.

A mandatory relationship specifies that a feature must be selected in a product instance if its parent also is. An optional relationship indicates that a feature may be selected in a product instance if its parent also is. The xor- and or-group relationships restrict groups of child features of the same parent. A xor-group (also known as an alternative group), represented symbolically with an arc drawn through all the child options, stresses that exactly one child feature from that group must be included in a product. This means, only one child feature must be selected when its parent feature is included. An or-group allows one or more than one child features from the group to be selected in a product instance simultaneously. Note that any non-empty child feature may be included in the product instance if their parent features are included. To consider the whole relationship between a parent and its child features, different relations can be combined. These relationships are considered only when the parent feature is included in the description of a particular product. Otherwise, none of them has an effect on the selection of a product instance.

(a) feature A requires feature B.

(b) feature A excludes feature B.

Figure 17: Cross-tree constraints.

In order to have a more accurate description of the relationships between child features, support for cross-tree constraints has been added to the basic FMs, most commonly the *Requires* and *Excludes* relationships (Kang et al., 1990). This are often represented with dashed arrows. As shown in Fig. 17a: "feature A requires feature B" means that feature B must be part of the product instance if feature A is selected. Conversely, the constraint "feature A excludes B", shown in Fig. 17b, restricts the inclusion of features A and B, that is, features A and B cannot be in the same product instance.

Figure 18 shows an FD for a possible FM of the e-commerce platform running example, adapted from (Czarnecki and Pietroszek, 2006). The root feature Catalog denotes the concept being designed, the structure of an electronic catalog. Feature *Image* is mandatory – every catalog must have images

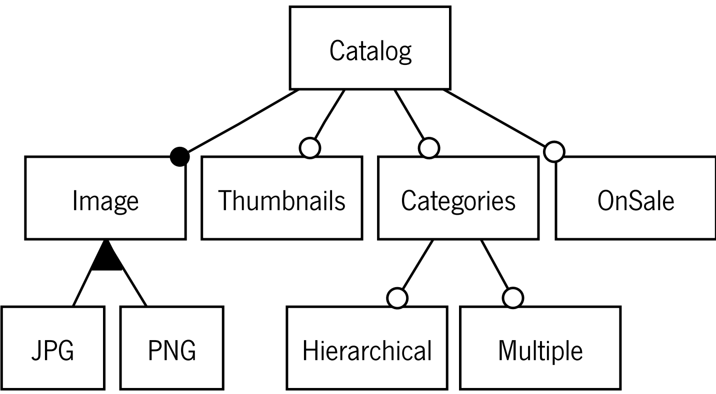


Figure 18: Feature diagram of the e-commerce platform.

representing each product. Different formats may be supported for these images, either as *PNG*, *JPEG*, or both: this is represented by an or-group below Image. An optional feature *Thumbnails* allows catalogs to be associated with a set of images selected from their products. Catalogs may additionally be structured in categories, denoted by optional feature *Categories* on the right-hand side. Two additional features for categories are supported as already discussed in Section 2.1: optional feature *Hierarchical* allows categories to be nested, while *Multiple* allows a product to be stored in multiple categories. Lastly, catalogues may also support products on sale, represented by the optional feature *OnSale*. The FD formally describes the set of products of this e-commerce system family. In this particular case, we have sixty valid configurations in total.

Besides these basic elements of an FM, several extensions have been proposed to include more information and to support different requirements that often arise in realistic software development scenarios.

Some authors (Czarnecki and Wasowski, 2007) allow FMs to be annotated with arbitrary propositional formulas over the presence of features, allowing more general cross-tree constraints not expressible in the basic FMs. To ease this specification of these constraints, besides the general operators, such as $\land$, $\lor$, $\neg$, $\rightarrow$, Batory (2005) additionally introduces a *choose* operator that may also be included in the expression. Specifically, $choose_{mn}(e_1 \cdots e_k)$ means that at least m and at most n of the expressions $e_1 \cdots e_k$ are true. The support of *reference* features has also been proposed to promote the reusage or modularization of FMs (Bednasch, 2002; Czarnecki et al., 2005). A *reference* allows to refer to other FMs directly or indirectly.

Other popular extensions introduce cardinalities in FMs. For instance, (Czarnecki et al., 2005) extend standard FMs with UML-like multiplicities (referred as cardinalities), namely *feature cardinalities* (Czarnecki et al., 2002) and *group cardinalities* (Riebisch et al., 2002). A feature cardinality describes the relationship

between a child feature and its parent with a sequence of intervals $[n..m]$, where n is the lower bound and m is the upper bound. These intervals determine the number of instances of a feature that may be part of a family member. A specification of a feature can have more than one interval. For example, $[0..2][6..6]$ shows that one can take 0, 1, 2 or 6 times of the corresponding feature. In particular, the mandatory and optional relationships can be generalized as $[1,1]$ and $[0,1]$ respectively. A group cardinality is an interval $\langle n,m\rangle$, with the lower bound n and the upper bound m, which represents the number of sub features that can be selected from a parent feature. Therefore, a xor-group can be changed to $\langle 1,1\rangle$ group cardinality, and an or-group to $\langle 1..n\rangle$, where n is the number of features in the group. For example, the or-group with PNG and JPEG in Fig. 18 can be labeled with group cardinality $\langle 1,2\rangle$, meaning that a product has to select at least one and at most two features in the group Thumbnails. The optional features Hierarchical and Multiple would be annotated with feature cardinalities $[0..1]$.

Another typical extension is the ability to introduce *feature attributes* (Czarnecki et al., 2002, 2005), so that a diagram can be more succinctly represented and have improved readability. The attributes of a feature can be introduced by associating it with a type which can either be a primitive data type, such as integer, float, Boolean, string, or a reference type that allows the user to model graph-like connections between features. A feature can have several attributes each of which can be modeled by a distinct subfeature of the given feature, in which case the hierarchy in the diagram increases rapidly and quickly leads to a very large diagram. With feature attributes, the developers can reorganize the tree structure to a more concise and intuitive diagram.

Since FMs are ubiquitous in all activities of feature-oriented software development, considerable efforts have been made to develop interchangeable formats for FMs (Sepúlveda et al., 2012; Eichelberger and Schmid, 2013). Some of these have have been developed mostly with tool support in mind, so that they are managed through a graphical interface and are not expected to be human-readable, such as XML formats provided by FeaturePlugin (Antkiewicz and Czarnecki, 2004), FAMA (Benavides et al., 2007) or FeatureIDE (Thüm et al., 2014). However, graphic representations have poor scalability due to cluttering in large systems. Thus, various human-readable text-based languages have been proposed for feature modeling, including FDL (Van Deursen and Klint, 2002), GUIDSL (Batory, 2005), SXFM (Mendonça et al., 2009a), VSL (Abele et al., 2010), and TVL (Classen et al., 2011a) (see, for instance, (Eichelberger and Schmid, 2013) for a detailed comparison).

As an example, we briefly present TVL (Text-based Variability Language), which supports most extensions of FMs and is used as an input language for FMs for some of the analyses that will be presented in Section 3.1.2. TVL is a light and comprehensive feature modelling language. It is a text-based human-

```
root Catalog group allOf {
  Image group someOf {
    PNG,
    JPG
  },
  opt Thumbnails,
  opt Categories group allOf {
    opt Hierarchical,
    opt Multiple
  },
  opt OnSale
}
```

Figure 19: E-commerce FM encoded in TVL.

readable feature modelling language with comprehensive constructs that is usually used as an auxiliary language for the variability modelling languages to specify the differences and common features during the verification process. It has a C-like syntax and a formal semantics which allows the engineer to model FMs intuitively and easily without concerns about ambiguities.

The root feature of an FM is specified by the keyword **root** followed by the name of the root feature. The decomposition relation is introduced by the **group** keyword, which is followed by the decomposition type. The and, or, and $alternative$ decomposition types were renamed to **allOf**, **someOf**, and **oneOf** respectively. The and-decomposition type can also specify cardinality. The cardinalities can be constants, natural numbers, or the asterisk character. If a feature is an optional feature, its name is preceded by the **opt** keyword. Each feature can declare its own children.

Figure 19 shows an example of the FM of the catalog structure of the e-commerce platform from Fig. 18 encoded in TVL. Cardinality-based decomposition relations and attributes can be easily modeled in the body of a feature in TVL Language. Furthermore, a feature can be extended easily since its just a name showing below the decomposition operator.

3.1.2 *Analyzing Feature Models*

The analysis of system families concerned to the verification and specification of family members has been paid a lot of attention in the research and practical fields. Extracting information from FMs manually is really time-consuming and error-prone. Hence, effective methods to provide automated analysis mechanisms have led to wide research interest since FMs were proposed. A set of operations, techniques, and tools has

been proposed in the literature. In this section, we give an overview of some of the most useful operations as well as the approaches for their automatic analysis.

Analysis Operations

Since the introduction of FMs, a number of operations have been proposed in the literature and play a significant role in the domain analysis step. Here, a brief review of some widely used operations will be given. To get more acquainted with the operations please see (Benavides et al., 2010). Analyses of FMs are categorized as correctness checking and configuration support (Mendonça et al., 2009b). Examples of correctness checking include the consistency checking and dead or common features finding, while examples of configuration support include finding optimal products or multi-step configuration.

Void Feature Model The *void feature model* operation is used to validate whether an FM is void or not. An FM is void if it represents no product instance. Usually, the major causes of a void FM are the wrong usage of cross-tree constrains. This operation is widely used in the debugging process especially in large-scale FMs (Benavides et al., 2010). In such models, the detection of this kind of problem is an error-prone and time-consuming task.

Number of Products This operation returns the number of valid product instances modelled by an FM. Obviously, the return number of a void FM is zero. The result of this operation shows the flexibility and complexity of a system family and may also help the developers to detect potential product instances.

Validating Products Most of the time, it is useful to check whether a product (also called specification in the literature) is valid or not when analyzing and managing system families. The validate operation, also referred as *product checking* or *product specification completeness*, determines whether a product belongs to the system family.

Finding Dead and Common Features A *dead feature* is a feature that does not belong to any of the valid products in a system family. Generally, dead features are caused by the wrong usage of cross-tree constraints between features. Notably, all features depicted in a void FM are dead features. A *common feature* is a feature that is shared by all product instances.

Identifying Atomic Sets An atomic set is a group of features that can be treated as a unit when analyzing an FM. The complexity of an FM is reduced since features of an atomic set always appear together. Obviously, features related by mandatory relations are always part of the same atomic set.

Finding all Family Members In an FM, a product represents a unique set of feature selection. Sometimes, the developer would like to find all valid products for the system family in order to identify new members that are not considered in the initial design step. The result of this operation is a set containing all valid products described in the FM.

Selecting Optimal Products This operation is used to find a product instance that fulfills a criterion decided by the user. The criterion is a function that determines how good a solution is for a particular product instance. In general, this operation is used in FMs extended with attributes with the parameters of the function constrained in the attribute field. This operation is well suited for CSP-based analysis approaches, as described later in this section.

Multi-step Configuration Multi-step configuration is a process to find a series of intermediate configurations that satisfy a series of configuration constraints and edge constraints. The set of edge constraints may include numerous types of constraints on the transition from one configuration to another. Configuration constraints must be satisfied at each step, such as the FM rules. This operation requires the initial configurations, desired final configurations, a number of steps for the configuration, and a function determining the cost to transition configurations through the steps. The result of this operation is an ordered list of configurations.

Automated Analysis Support

There is extensive work on supporting such FM analysis operations (see for instance (Benavides et al., 2010) for a survey). Most of the analyses concern the operations just described. Here we divide them into three groups according to the underlying logic paradigm. Particularly, we present SAT-based analysis approaches, CSP-based approaches, and other paradigms. The main idea of SAT and CSP approaches is to translate an FM to a property in the respective logic, and then automatically analyze that property using an off-the-shelf solver. Besides the binary decision operations, CSP-based methods can deal with operations yielding integer or set values, which cannot be handled natively by SAT-based approaches.

Table 2: Rules for translating FMs to propositional formulas.

Relationship	Graphical notation	Mapping to a formula
root	r	r
mandatory	p, c	$p \leftrightarrow c$
optional	p, c	$c \rightarrow p$
or-group	p, c_1, c_2, c_n	$p \leftrightarrow (c_1 \vee c_2 \vee ... \vee c_n)$
xor-group	p, c_1, c_2, c_n	$(c_1 \leftrightarrow (p \wedge \neg c_2 \wedge ... \wedge \neg c_n)) \vee$ $(c_2 \leftrightarrow (p \wedge \neg c_1 \wedge \neg c_3 \wedge ... \wedge \neg c_n)) \vee ... \vee$ $(c_n \leftrightarrow (p \wedge \neg c_1 \wedge \neg c_2 ... \vee \neg \wedge c_{n-1}))$

SAT-based Analysis SAT-based approaches have attracted considerable attention especially after Batory (2005) proposed the rules for translating FMs to propositional logic formulas. In particular, the features in the model are encoded as Boolean variables, while the relationships as well as the additional constraints, are translated to formulas expressed over these variables. Table 2 shows the translation of relations between a parent and its child features. Feature r represents the root feature of an FM and will be translated to a simple formula r that forces the respective variable to be true. The relationship between a parent feature and its child features are represented by the *implication* operator. More specifically, there is an implication from every child feature c to its parent p. However, only the *mandatory* child features are implied by its parent. An *or-group* is represented by an implication from the parent feature p to at least one of its child features $c_1, c_2, ..., c_n$, while in a *xor-group* the parent implicates exactly one of its child features. To obtain the semantics of an FM, all formulas for its individual relations are conjoined together, and then some of the operations described above can be implemented trivially by invoking a SAT solver (for example, determining if the FM is void or validating a product).

Some works use higher level modelling languages instead of SAT directly. For example, considerable work has been performed on analyzing FMs using Alloy (Sree-Kumar et al., 2016). The implementation of FMs needs to be made manually since there is no standard format to represent FMs in Alloy. In general, the possible features are described as a collection of singleton signatures and the relation and the cross-tree constraints among features are encoded using predicates following the semantics described

above. Regarding the automatic analysis, assert commands can be used for operations such as product validation or checking common features, while run commands can be used for operations such as finding all family members. This approach was followed, for example, in one of the Alloy encodings proposed in (Gheyi et al., 2006), the so-called R-theory of FMs. In this encoding, an FM is just a collection of feature names (to be instantiated with singleton signatures).

```
sig FM {
    features: set Name
}
sig Name {}
```

The semantics of the FM can then be encoded with predicates that enforce a particular restriction over a configuration (a set of feature names), for example:

```
pred mandatory(A,B:Name, config:set Name) {
    A in config ⇔ B in config
}
```

In this work another encoding was proposed, the so-called G-theory, where an FM includes an explicit representation of its relations and cross-tree constraints. For example, for capturing relations we have the following signature.

```
sig Relation {
    parent: Name,
    child: set Name,
    type: Type
}
abstract sig Type {}
one sig Optional, Mandatory, OrFeature, Alternative extends Type {}
```

Many operations, namely checking valid/invalid configurations, detecting dead and common features, or checking if a transformation is a refactoring, can be implemented with both encodings, but some can only be implemented with the G-Theory. However, the R-Theory is more efficient due to the much reduced number of signatures to encode a particular FM. Another work that used an encoding of FMs in Alloy very similar to the G-theory is (Ripon et al., 2012).

CSP-based Analysis A Constraint Satisfaction Problem (CSP) is a mathematical problem that consists of a set of variables, a domain for each variable, and a set of constraints restricting the values of the variables. The aim of a CSP is to choose a value for each variable so that all constraints are satisfied. Constraint programming is a powerful and flexible technique for dealing with CSPs, in particular those with lots of constraints.

In the research literature, there are some works (Benavides et al., 2005a,b, 2010) that propose the use of constraint programming for the automated analysis of FMs. When translating an FM to a CSP, the features are mapped to variables with `{True,False}` or `{0,1}` domain, depending on the solver that will be selected. The relations between features are translated to different kinds of constraints over feature variables. Let us suppose p is the parent of child features $c_1, ..., c_n$, and the domain of the feature variables is `{0,1}`. The mandatory relation between p and c_1 will be encoded as $c_1 = p$, while the optional relation between c_1 and p will be mapped to the conditional expression $\text{if } (p = 0) \text{ then } (c_1 = 0)$. An or-group relation in the children will be transformed into expression $\text{if } (p > 0) \text{ then } (1 \leq c_1 + ... + c_n \leq n)$ else $c_1 = ... = c_n = 0$, and a xor-group relation would be encoded with constraint $\text{if } (p > 0) \text{ then } (c_1 + ... + c_n = 1)$ else $c_1 = ... = c_n = 0$. In many real-life development scenarios, we do not want to find any solution but a good solution. Constraint programming is more effective that SAT for finding an optimal solution for a particular requirement. The criterion to assess the quality of a solution can be modeled as an objective function based on the variables described in the feature and attributes. As such, this approach is better fitted for dealing with FMs extended with attributes. After the transformation, an off-the-shelf CSP solver, such as IBM's CPLEX Optimizer[1], can be used to conduct an automatic analysis.

Most of the analysis operations for FMs can be performed by constraint programming (Benavides et al., 2007). A detailed presentation of such analyses, for example computing the number of products, validation of an FM, or finding valid products can be found in (Benavides et al., 2005b). The result for the same operation can be different if different solvers are chosen. The performance may be improved if several solvers are working together (Benavides et al., 2007). In this context, a multi-platform and extensible framework, FAMA (Benavides et al., 2007) was proposed. It allows the integration of different solvers (including SAT solvers) to optimize the analysis process. FAMA can be configured to use the most efficient available solver for each operation during execution time. The configuration parameters are set based on the priority between each operation for the available solver. Usually, the performance of analysis using SAT

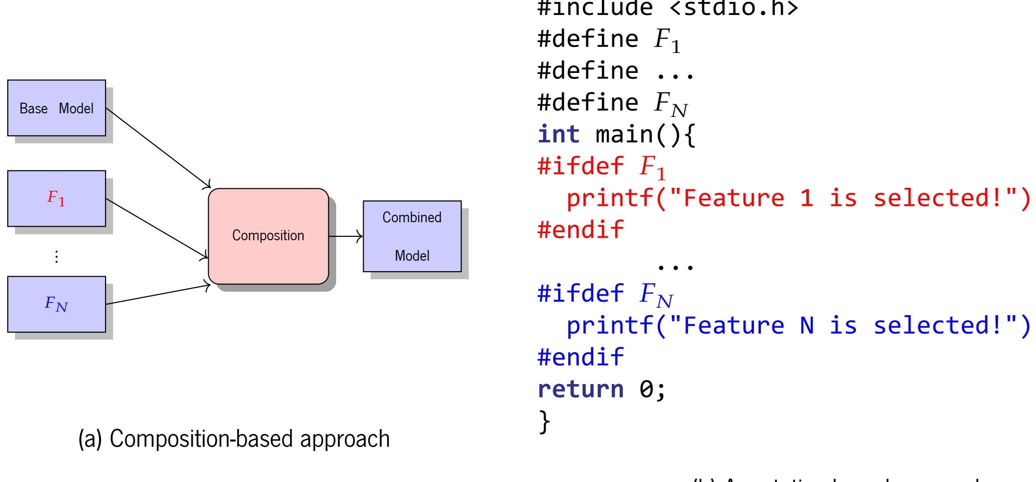
1 https://www.ibm.com/analytics/cplex-optimizer

solvers is slightly better than of CSP, but, as mentioned above, the latter can handle FMs extended with attributes and finding optimal solutions according to criteria that maximize or minimize attribute values.

Other Kinds of Analysis There are some works that use neither SAT nor CSP to perform automatic analysis of FM. For example, in (Segura, 2008) an algorithm is proposed to simplify FMs by calculating atomic sets. The idea is reduce the number of variables that are needed for its analysis, by replacing each feature variable by a new variable that identifies the atomic set it belongs to. This algorithm proceeds by recursively checking all the child features of the FMs. A child feature is added to the current atomic set if it is mandatory. Otherwise, a new atomic set including the child feature is created and added to the collection of atomic sets. A set of atomic sets is returned when all the features are checked. In (Wang et al., 2007) a method is proposed that uses description logics to analyse FMs. Description logics are a family of knowledge representation languages enabling the reasoning within knowledge domains by using specific logic reasoners. It consists of a set of concepts, a set of roles, and set of individuals (or instances). The problem is solved by a solver providing facilities for consistency and correctness checking. In this method, *Web Ontology Language* (OWL) ontologies were used to capture the relationships among the features. The analysis of an FM can then performed with off-the-shelf OWL reasoning engines, which can for example be used to check for inconsistencies of feature configurations. As a final example, in (Hemakumar, 2008) a method is described to statically detect conditional dead features in a system family. A contradiction in an FM is a specification error. Finding such errors is important, both for designers and customers. An FM is k-contradiction-free where $0 < k \leq n$, if every selection of k features does not expose a contradiction (i.e. they are compatible). The FM is proven to be contradiction-free if n is the number of selectable features for the user.

3.2 Modeling in Feature-oriented Design

Specification languages are textual computer languages that define the system models as well as the system properties by a systematic set of rules and frameworks, and are used to model the system under development during design. General-purpose specification languages are often sufficiently rich to support the explicit encoding of features and variability points. However, as the complexity of the system increases, managing variability may become impractical, and languages have been developed or adapted to natively support feature-oriented design when developing a family of similar systems. This also applies to formal specification languages if designs are expected to be validated and verified (see, for instance, the survey by



(a) Composition-based approach

(b) Annotation-based approach

Figure 20: Language categorization based on variability representation.

Benduhn et al. (2015)). Likewise SPL implementation, feature-oriented design approaches also typically fall in two groups depending on how variability is represented: composition- and annotation-based approaches.

Composition-based approaches model a system with a base model and a set of delta models describing distinct feature units and specifying the additional variants of the system. As show in Fig. 20a, a variant modeled in such kind of language is derived by compositing the base model with a set of valid features ($F_1, \cdots, F_n$) one by one using a composition tool. Hence, the order in which these features are composed matters for generating the final model, since the behaviour specified by one feature maybe rewritten by another that appears later in the composition list. This approach allows the modular development of features and provides a clear mapping from the features to their implementations promoting feature traceability. Therefore, when a developer wants to debug errors that occur in a particular feature, they only needs to work with the code associated with that feature, rather than walk through the entire code. This kind of approach is best suited for coarse granularity variability, such as changing the entire method of a class. Fine grained variability, such as small changes in the body of a method, are often very difficult or impossible to achieve due to the conceptual limitations (Kästner et al., 2008).

In annotation-based approaches all variants are superimposed in a single code unit and variability implemented by annotating certain parts of the code with the variant they belong to. During variant derivation, all code that belongs to deselected features or invalid feature combinations is removed from the source code. A typical example of this kind of approaches is the explicit usage of `#ifdef` and `#endif` annotations of the C/C++ preprocessors which surrounds related feature code fragments as shown in Fig. 20b (code belonging to distinct features is presented in different colors). In contrast to composition-based approaches, annotation-based approaches add feature annotations in the actual lines of the source

code, which naturally supports fine-grained variability. However, it poorly supports feature traceability, since a feature is not encapsulated in a single code unit but scattered in the code as feature annotations, which can lead to maintainability issues. Although tools such as CIDE (Kästner et al., 2008) provide virtual views and navigation support of feature traceability at the tool level, feature traceability in such approaches is still a matter of tool support (Kästner and Apel, 2008). Although the composition-based approach offers an explicit mapping between features and their implementations, which brings a straight forward trace and easy maintenance of the features, in practice the annotative approaches are more popular (Le et al., 2013). The industry is very careful when adopting a composition-based approach because it will have too much impact on the existing code base and the development process. Annotative approaches, by contrast, can be adopted more quickly because they require only lightweight tools which do not change much of the code or development process (Kästner and Apel, 2008).

This section reviews existing work on feature-oriented modelling of SPLs. The focus is on formal high-level specification languages, although we also explore some graphical conceptual modelling languages. We briefly explore how general-purpose modelling languages can be used to model multiple variants in an *ad hoc* manner in Section 3.2.1. Then we explore composition- and annotation-based modelling languages for feature-oriented design in Section 3.2.3 and Section 3.2.2, respectively.

3.2.1 *Ad-hoc Approaches*

Some general-purpose modelling languages are sufficiently expressive to support some kind of feature-oriented design. The main advantage of such approaches is that they inherit all existing support for a well-established language, including subsequent analyses using existing model checkers or theorem provers. The caveat is that the implementation of the FM must be performed manually, which is cumbersome and error-prone for realistic SPLs. Moreover, it also limits the ability to perform more advanced feature-aware analyses since features are not first-class entities in this context.

Some authors explore the decomposition capabilities of general-purpose modelling languages to simulate the implementation of a composition-based approach. For structural modelling, (Shiraishi, 2010) proposed a variability modeling approach for the Architecture Analysis and Design Language (AADL), a language for modeling system architectures. AADL supports the specification of several types of components, such as process or threads, as well as their communication through ports. The interface of a component is given by a *type* definition, and its internal specification by an *implementation*. Components can be *extended* or *refined*, and the authors explore how these mechanisms can be used to model variability. Considering

approaches focused on modeling behavior, Gondal et al. (2010) proposed using a feature configurator tool for Event-B to compose features selected by the user. Here, each feature is mapped directly to a plain Event-B machine. For ASM, Batory and Börger (2008) proposed to use its own mathematical transition rules in the form of *If Condition then Updates* to describe new behaviour and states of the machine. Such extensions can be composed in several ways, such as conservative extensions that introduce new behavior in a module by increments (for example, add exception handling), parallel additions that add extra behavior with the same guard (condition) of the original, or introductions that add new elements.

Alternatively the whole SPL can be represented in a superimposed model and somehow relate elements of the model – either structural or behavioural – to presence conditions. These presence conditions may not be explicitly present in the model but rather implicit in supporting tools. This is the case of the approach proposed by Calder and Miller (2006), where Promela is used to model a system with variability to analyse feature interactions, using an *ad hoc* tool to project transitions/variables of specific variants. Another approach is to encode the FM in the general-purpose modelling language alongside the superimposed system model and explicitly implement the presence conditions based on the valuation of the FM, a technique known as *lifting* (Post and Sinz, 2008). This is followed, for instance, in (ter Beek and de Vink, 2014a,b), where the authors explore whether the mCRL2 model checker can be used to model and verify SPLs. Here, encoding of an SPL involves two state machines, one encoding the initial feature selection, and another the behavior of the family whose transitions depend on the initially selected product.

This is also the approach followed by Clafer (CLAss, FEature, Reference) (Bak et al., 2016), firstly proposed as a class modeling language and then refined to include behaviour modeling (Juodisius et al., 2018). Clafer unifies basic modeling constructs - such as classes or features - into a single construct named *clafer*. The format of the model, especially the nesting space, is very important in Clafer, since it represents inheritance. For example, if clafers represent features, the fact that clafer B is nested in A means that B is a subfeature of A. A multiplicity annotation follows the declaration of a clafer, for instance `1..1` meaning exactly one (the default by omission), `0..*` (or `*`) meaning zero or more, or `0..1` (or `?`) zero or one. Group cardinalities can also be assigned to restrict the number of child instances, for instance `0..*` (or **`opt`**, the default), or `1..1` (or **`xor`**) representing a xor-group. A clafer can be defined as a subset of another clafer by either arrow notation `->`, denoting a subset, or by a colon `:`, denoting a partition. Constraints to express dependencies among clafers or restrict numerical and textual values are modeled in square brackets in the context of the clafer being restricted. Assertions can be defined with the same constraint language, if marked with the keyword **`assert`**. Clafer uses plain Alloy as its verification backend

(although the supported expression language is more restricted), supporting bounded analysis over the defined models.

As an example, consider a simplified version of the e-commerce catalog SPL already presented, assuming the FM from Fig. 18 but without feature Hierarchical. A possible specification in Clafer is depicted in Fig. 21. The first clafer `Catalog` represents the FM, and by default will have exactly one instance. Clafer `Image` is marked with group cardinality **`xor`** to denote a xor-group, while the other children features are marked as optional with multiplicity `?`. The remainder clafers denote the structure of the system, with a clafer declared for each element type with multiplicity `*`, allowing any number of instances (except for `Format`, which encodes some kind of enumeration type). To model structural variability points, the presence of elements is controlled by associating restrictions over features, such as **`no`** `Categories => ` **`no`** `category` that forces `category` to be empty when there is no `Categories` instance (i.e., the Categories feature is not selected). Features are simply treated as any other element of the model, and as the model becomes more complex it may not be easy to identify variability points. The `Thumbnails` restriction – that all images from a catalog are present in one of its products – is added as a top-level constraint. Note that the constraint language is a restricted version of Alloy, e.g., you cannot navigate backwards, hence the more verbose encoding of this constraint. Transitive closures are also not supported, so it is unclear how the feature Hierarchical would be implemented. Lastly, the `AllCataloged` assertion is encoded, which specification depends on whether we have a configuration with Categories.

Although the initial proposal was purely structural, an extension has been proposed to support modeling behavioural models with variability (Juodisius et al., 2018). They introduce a temporal operator `-->` to constraint state transitions, namely that if the condition on the left of `-->` holds in one state, then the condition on the right must hold in the next state. Likewise structural constraints, variability in behaviour is controlled by presence conditions in the constraints. Lastly, assertions can also be written using LTL operators such as **`always`**, **`never`**, **`sometime`** or **`always`**.

A similar approach can be employed with Alloy and Electrum, as shown in (Macedo et al., 2016) for the latter. One possible approach is to encode each feature as a singleton signature, and then an extra signature, here `Configuration`, contains a selection of features. Consider, as an example, the vending machine SPL, for which Fig. 23 depicts a possible specification in Electrum. The corresponding FD is shown in Fig. 22. The base variant allows the selection and serving of a single product. The base model is extensible by introducing three independent features: *Cancel*, allowing to cancel the selection before serving products; *MultiSelection*, allowing the selection of several products in one interaction; and *Free*, allowing free drinks. What's more, feature MultiSelection is only allowed when free drinks are not allowed and a

```
Catalog
  xor Images
    JPG
    PNG
  Thumbnails ?
  OnSale ?
  Categories ?
    Multiple ?

Product *
  catalog -> Catalog *
  [ Categories => no catalog ]
  category -> Category *
  [ no Categories => no category ]
  [ no Multiple => lone category ]
  images -> Image *
  onSale ?
  [ no OnSale => no onSale ]
  assert [ if Categories then some catalog else some category.
      inside ]

Image *
  format -> Format

abstract Format
Jpg : Format ?
[ JPG <=> one Jpg ]
Png : Format ?
[ PNG <=> one Png ]

Category *
  inside -> Catalog
[no Categories => no Category]

Catalog *
  thumbnails -> Image *
  products -> Product *
  [ no Thumbnails => no thumbnails ]

[ Thumbnails => (all c : Catalog | all i : Image | i in c.
    thumbnails => (some p : Product | i in p.images && (if
    Categories then c in p.category.inside else c in p.catalog)))
    ]
```

Figure 21: E-commerce catalog SPL encoded in Clafer.

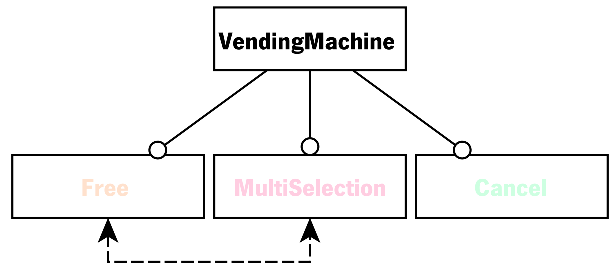


Figure 22: Feature diagram of the vending machine example.

restriction can be imposed over signature `Configuration` to forbid `Free` and `MultiSelection` from being selected simultaneously. Features can then be tested to be in `Configuration` and the structural and behavioural properties adapted accordingly. For instance, when `MultiSelection` is not selected, there can be at most one product selected at anytime. Regarding behaviour, one can, for instance, forbid the application of certain predicates with pre-conditions over configurations, as in `Cancel` when feature Cancel is not selected. Moreover, the behaviour of the predicate can itself depend on the selected configuration, as in `Cancel` where the succeeding `state` depends on whether MultiSelection is selected or not. Lastly, assertions themselves can also be conditioned by the configuration.

3.2.2 *Composition-based Languages*

In composition-based specification languages, the base model and FMs are specified separately and later integrated using an automatic tool. Most composition-based approaches extend existing languages with some support for feature modules that can then be combined in a single product after configuration. For formal approaches, such is the case of fSMV (Plath and Ryan, 2001; Classen et al., 2014) for the SMV input language of symbolic model checkers, FeatureAlloy (Apel et al., 2010) for the Alloy lightweight specification language, or ProFeat (Chrszon et al., 2018) for the Prism language for probabilistic model checking, supported by TVL-inspired FMs. Approaches for conceptual modelling include FORML (Shaker et al., 2012), where each feature module encodes a state-machine, and the approach proposed by Greenyer et al. (2012) where each feature module encodes a modal sequence diagram (a variant of live sequence charts).

Some compositional approaches explore concepts of delta modelling for feature-oriented design, which allows for finer granularity modifications. One of the first applications of delta modelling was in the context of model-driven development with UML (Schaefer, 2010). Lochau et al. (2014) propose DeltaCCS, whose implementation extends a Maude implementation of CCS. Sabouri and Khosravi (2013) propose an extension for the actor-based Rebeca language. Haber et al. (2011) proposed Δ-MontiArc, an extension for

```
open util/natural

abstract sig Feature {}
one sig Cancel, MultiSelection, Free extends Feature {}
sig Configuration in Feature {}
fact FeatureModel {
  Free+MultiSelection not in Configuration
}
abstract sig State {}
one sig Ready, Paid, Selected, Served, Done extends State {}
var one sig state in State {}
sig Product {}
var sig stock, selection in Product {}
var one sig balance in Natural {}
fact {
  MultiSelection not in Configuration implies always lone selection
}
pred Select [p : Product] {
  state in (Free in Configuration implies Ready else Paid)+
           (MultiSelection in Configuration implies Selected else none)
  p in stock – selection
  state' = Selected
  stock' = stock
  selection' = selection + p
  balance' = balance
}
pred Cancel [] {
  Cancel in Configuration
  state = Selected
  state' = Free in Configuration implies Ready else Done
  stock' = stock
  selection' = none
  balance' = balance
}
...
fact Behaviour {
  state = Ready
  no selection
  some stock
  balance = Zero
  always {
    (some p : Product | Select[p]) or Pay or Cancel or
    Serve or Open or Change or Nop
  }
}
assert DoneImpliesServed {
  always (state = Done implies once state = Served)
}
assert PaidImplesServed {
  Free not in Configuration implies
  always (state = Paid implies eventually state = Served)
}
```

Figure 23: Vending machine SPL encoded in Electrum.

the MontiArc ADL. Clarke et al. (2010) propose an extension to the ABS language, with FMs provided in a language inspired by TVL. Others have explored aspect-oriented modelling in the context of feature-oriented design. In the approach proposed by Noda and Kishi (2008), each aspect represents a feature and is comprised by a class diagram and a state diagram, restricted by rules between them to enforce the FM.

In the remainder of this section we present in more detail three composition-based approaches that we feel are most relevant to the work developed in this thesis, namely fSMV for dynamic modelling, FeatureAlloy for structural modelling, and Δ-modelling for conceptual modelling.

fSMV

The fSMV language, initially proposed by Plath and Ryan (2001) and later adapted by Classen et al. (2014), is a composition-based feature extension of the symbolic modeling language SMV (McMillan, 1993). In fSMV, a complex system can be considered as a base system plus a number of feature modules. The base system and its features are modeled as different textual units. Specifically, the base system is modeled in pure SMV, while a feature is described in a new structure named *feature construct* that overwrites the existing behavior when superimposed over another SMV module. In the initial proposal (Plath and Ryan, 2001), the features are integrated automatically by means of a tool, called SFI (SMV Feature Integrator), that compiles into pure SMV so that verification can be performed by the plain SMV model checkers. In (Classen et al., 2014), it is translated into a dedicated model checking algorithm (see Section 3.3). Ideally, this superimposition process should be restricted by an FM, but this has not been fully implemented for fSMV (Classen et al., 2014).

Essentially, an SMV model consists of a section **VAR** with set of variable declarations defining the state space and a set of assignments that specify the evolution of the variables. For each variable, in section **ASSIGN** there can be an **init** assignment that defines its initial value and a **next** function constraining the value in the next state based on its current value or the value of other variables. Initial states and transitions can also be restricted by constraints with **INIT** and **TRANS** sections. Like other modeling languages, modules (labeled with keyword **MODULE**) are used to encapsulate elements. Specifications to be model checked are then encoded as LTL or CTL in constructs **LTLSPEC** or **CTLSPEC**, respectively.

As an example, we encode a simplified version of the vending machine SPL in fSMV (with a single type of product, with infinite stock, so the current selection amounts to a count of items). The base model, shown in Fig. 24, contains a variable `State` with the valid states of the vending machine as an enumerated type. The **init** assignment defines the system's initial value (initially, the system is `Ready`). The **next**

```
MODULE Main

VAR
  State : {Ready, Paid, Selected, Served, Done};

ASSIGN
  init(State) := Ready;
  next(State) := case
                   State = Ready    : Paid;
                   State = Paid     : Selected;
                   State = Selected : Served;
                   State = Served   : Done;
                   State = Done     : Ready;
                 esac;
```

Figure 24: Base vending machine model in fSMV.

assignment defines the transition relation between, defining how the next state depends on the current state.

A feature in fSMV is declared with the keyword **FEATURE**. There are three main sections of the feature construct, defined by the keywords **REQUIRE**, **INTRODUCE** and **CHANGE**. The **REQUIRE** section describes what entities are required to be present in the base model in order to encode the specific feature. A collection of modules and variables may be involved in this section, and all preexisting modules and variables that are used in the **INTRODUCE** and **CHANGE** sections must be described in the **REQUIRE** section. The **INTRODUCE** section introduces new modules, variables, assignment clauses or specifications that will be used in the feature. These are directly added to the SMV code when conducting the integration. The **CHANGE** section specifies the changes of the system behavior in terms of features. It includes a number of **TREAT** or **IMPOSE** clauses, associated with an application condition. **TREAT** replaces all variable occurrences with a different expression when a certain condition holds, otherwise it remains the same. **IMPOSE** statements deal with assignments of variables, changing the update of a variable when a certain condition holds, otherwise preserving the original behaviour.

Let us consider feature Cancel which adds a transition from state `Selected` back to `Done`, representing the cancellation of the selection and the return of the inserted change. A feature in fSMV can only update existing assignments by introducing a new guarded expression, which can override existing ones if the guard overlaps. A possible encoding of Cancel as an fSMV feature is shown in Fig. 25. The **CHANGE** section introduces a new non-deterministic transition when the system is in state `Selected`, either to `Served`, as originally, or to `Done` when cancelled. In the merged model **`next`**`(State)` is changed so that state `Selected` always enters that assignment, leaving the original transition to only `Served` unreachable. A

```
FEATURE Cancel

REQUIRE
  MODULE Main
  VAR State : {Ready, Paid, Selected, Served, Done};

INTRODUCE
    CTLSPEC EF !(State = Served)

CHANGE
  MODULE Main
  IF State = Selected THEN
    IMPOSE next(State) := {Done,Served};
```

Figure 25: Feature Cancel of the vending machine in fSMV.

new specification only relevant for products where Cancel is selected – checking if there are paths where the drink is never served – is also added in the **INTRODUCE** section.

As another example, feature MultiSelection in fSMV is depicted in Fig. 26. This feature introduces a new variable `cnt` – counting the number of selected items – as well as assignments for its initial state and transitions. In this case, the increment of `cnt` occurs in a transition from `Selected` to itself when the user requests an additional item, is reset when leaving state `Served`, and is left unchanged otherwise. The transition of `State` when in `Selected` is also updated accordingly, allowing the non-deterministic transition either to `Served` or back to `Selected`.

The main advantage of fSMV – as other composition-based approaches – is that features are developed independently, promoting the separation of concerns, modularity and maintainability. Unfortunately, such approaches also have some drawbacks. Some are related to the granularity and nature of the refinement operations usually provided by such approaches. In fSMV, for instance, existing expressions in assignments cannot be updated (e.g., by adding a new possible state as in our example), meaning that the user must always fully specify the update for the intended guard. Another limitation is that the provided operations are mostly incremental, so that existing sections (such as variable declarations) cannot be removed.

Another issue is related to the feature interaction problem, which arises when features interfere with each other in ways that are not easy to predict. In particular, transitions affected by several features are hard to model in fSMV's composition-based design. In fact, this is patent in our example: both Cancel and MultiSelection changes introduce a new assignment when `State` is in `Selected`, so in products with both features selected, depending on the order of composition, it will either end with the non-deterministic choice between `Served` and `Selected` or `Served` and `Done`, although the expected behaviour would be the choice between those 3 states. In composition-based approaches with the level of granularity of fSMV,

```
FEATURE MultiSelection

REQUIRE
  MODULE Main
  VAR State : {Ready, Paid, Selected, Served, Done};

INTRODUCE
  VAR cnt : [0..5]
  ASSIGN
    init(cnt) := 1;
    next(cnt) :=
        case
            State = Selected & next(State) = Selected & cnt<5 : cnt+1;
            State = Served                                    : 1;
            true                                              : cnt;
        esac;

CHANGE
  MODULE Main
  IF State = Selected THEN
    IMPOSE next(State) := {Selected,Served};
```

Figure 26: Feature MultiSelection of the vending machine in fSMV.

this problem must be addressed through the creation of a new *derivative* feature that must be additionally merged when both other features are selected. This is depicted in Fig. 27, fixing the `State` transition. This approach will undermine the reusability characteristic of composition-based approaches and as the number of feature interactions grows, it may become unmanageable.

FeatureAlloy

FeatureAlloy is a feature extension of the lightweight modelling language Alloy proposed by Apel et al. (2010). Like fSMV, FeatureAlloy is a composition-based approach that takes advantage of the distinguishing characteristics of feature-oriented software development and establishes a clear mapping between features and their design. It separates the features of a complex system, making them explicit in the design phase.

The elements that are to be introduced or changed by a feature (signatures, predicates, functions, or assertions) are encapsulated and modeled separately from the base system. These features combine with the base model or other features by means of a tool, called FeatureHouse, in order to produce a final model in plain Alloy. Assertions can then be checked using the standard Alloy Analyzer. The composition is achieved by recursively superimposing and merging the features selected by a user. FMs are not explicitly considered during superimposition, and the authors argue that such assertions capture depedences that would not have been present in FM regardless.

```
FEATURE CancelMultiSelection

REQUIRE
  MODULE Main
  VAR State : {Ready, Paid, Selected, Served, Done};
      cnt   : [0..5];

CHANGE
  MODULE Main
  IF State = Selected THEN
    IMPOSE next(State) := {Served,Done,Selected};
```

Figure 27: Derivative feature for Cancel and MultiSelection of the vending machine in fSMV.

```
module ecommerce

sig Product {
    catalog : some Catalog,
    images : set Image
}

sig Image {}
sig Catalog {}
```

Figure 28: Base model of the e-commerce catalog in FeatureAlloy.

FeatureAlloy combines new features by means of refinement. Elements in the refining module are added to the base module or overwrite the existing elements. Specifically, signatures are refined by adding new fields (or overriding existing ones) while paragraphs (except assertions) are refined by overriding. All matches are determined by the elements' names. Assertions are not allowed to be replaced by subsequent features since the authors advocate their use to detect semantic dependences and feature interactions. We will explain in detail FeatureAlloy by encoding some features of the e-commerce catalog running example. The base model, shown in Fig. 28, organizes the system in catalogs, images, and products assigned to some catalogs and with possibly images associated.

Figure 29 depicts a module that refines the base model by introducing feature Thumbnails. Signature `Catalog` has a new field `thumbnails` which will be added to the `Catalog` signature from the base model. Fact `Thumbnails` did not exist in the base model, and is simply added as a new fact in the superimposed model.

Figure 30 shows instead the module for feature Categories, where catalogs are now comprised by categories rather than individual products. A new signature `Category` is introduced, as well as a new field `category` associating each product with **`some`** category. However, in this feature products are

```
refines module ecommerce

sig Catalog {
    thumbnails : set Image
}

fact Thumbnails {
    all c:Catalog | c.thumbnails in (catalog.c).images
}
```

Figure 29: Feature Thumbnails of the e-commerce catalog in FeatureAlloy.

```
refines module ecommerce
sig Product {
    catalog : set Catalog,
    category : some Category
}
fact NoCatalogs {
    no catalog
}

sig Category {
    inside : one Catalog
}
```

Figure 30: Feature Categories of the e-commerce catalog in FeatureAlloy.

no longer associated directly to catalogs, which would ideally result in the removal of field `catalog` of `Product`. This is however impossible in FeatureAlloy, where refinements do not allow the removal of fields. A workaround is to override the existing field with another with a looser multiplicity **`set`**, and then add an extra fact `NoCatalogs` saying that there is no `catalog` in the model. Note however, that this does not actually remove the field from the model but only forces it to be empty. In fact, although this is not the case in our example, the user would have to override every element of the base model where field `catalog` was called, as they would still be considered with a permanently empty `catalog`.

As already pointed out when presenting fSMV, feature interaction in composition-based approaches often requires additional derivative features to solve conflicts. The two features we just presented in fact interfere with each other: fact `Thumbnails` retrieves thumbnails from all products associated with a catalog, which is no longer valid when categories are considered, since images must be indirectly retrieved through categories. Figure 31 denotes such a derivative feature, that should be considered when both Thumbnails and Categories are selected, overriding the original `Thumbnails` fact with one considering categories.

```
refines module ecommerce

fact Thumbnails {
    all c:Catalog | c.thumbnails in (category.inside.c).images
}
```

Figure 31: Derivative feature for Categories and Thumbnails of the e-commerce catalog in FeatureAlloy.

Likewise fSMV, the cons and pros of FeatureAlloy are those typically shared by composition-based approaches. Features can be developed in a modular and step-wise fashion. However, the final model is obtained by recursively applying refinement for the models selected by a user, so if features interact the result depends on the order of feature selection. As shown, refinements are incremental and code deletion is not supported. The workaround we implemented pollutes the code and is not manageable in more complex features. Moreover, solving feature interactions usually requires developing additional features to be merged after the interacting ones.

Δ-modelling

The Unified Modelling Language (UML) is a large and popular modeling language in software development for specifying, constructing, visualizing, and documenting products. However, the standard UML does not support the modelling of variability in system families because it was proposed to model a single system. Although we can use mechanisms such as specification and redefinition to model variability, it is not sufficient for the systems with large variability. Therefore, researchers have drawn their attention to feature-oriented extensions to UML both using compositional and annotative techniques. Here we present one such composition-based approach proposed for model-driven development, which can be applied to UML models.

Δ-modelling (Schaefer, 2010) is an approach proposed for step-wise refinement of models with variability. At each level of abstraction there is a core model and a set of Δ-models. Each Δ-model specifies a set of changes to the core model (additions, modifications, and removals of elements and relations) and is associated with a logic constraint to determine for which feature configurations this fragment is to be applied. In order to obtain a specific product configuration, the changes specified by the Δ-models are applied to the core model. Feature interactions may cause conflicts, for instance if the same element is removed and modified by different Δ-models occurring in the same feature configuration. Although we are not addressing this scenario, the authors show that when considering multiple abstraction layers, refining

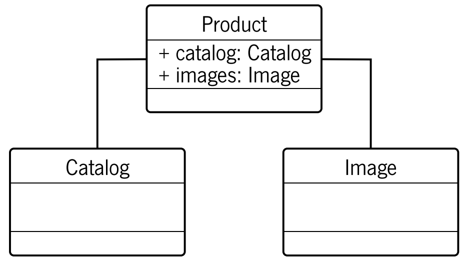


Figure 32: Core class diagram for e-commerce.

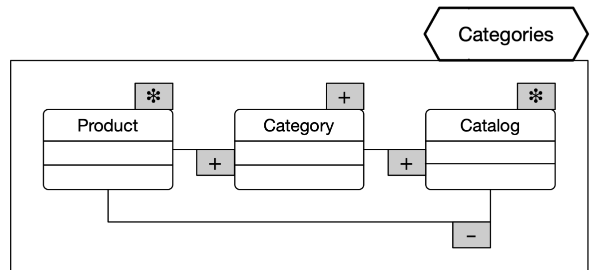


Figure 33: Class diagram Δ-Model for Categories.

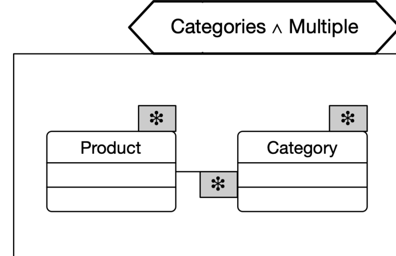


Figure 34: Class diagram Δ-Model for Multiple.

the core model and Δ-models and then configuring a product, commutes with first configuring a product and then refining it.

As an example, we present a possible specification of the class diagram of the e-commerce catalog example, along with a few features. This amounts essentially to the signature and field declarations of the Alloy model, since the approach does not consider additional constraints, e.g. in OCL. Figure 32 shows the core class diagram of the base system, declaring products, catalogs and images, as well as their associations. Two Δ-models are then presented in the graphical style proposed in (Schaefer, 2010), one for feature Categories in Fig. 33 and another for feature Multiple in Fig. 34. The logic conditions on the top-right corner of the diagram determine the application condition of the Δ-model. The Δ-model for Category adds the `Category` class (denoted by the + annotation) and modifies `Product` and `Catalog` (∗ annotation) by removing the `catalog` association (— annotation) and adding the `category` one. The Δ-model for Multiple modifies the association between products and categories, allowing many-to-many relationships.

The level of granularity of approaches for conceptual models is often finer than those for models with additional constraints, such as fSMV or FeatureAlloy. For instance, recall that we were not able to remove the field `catalog` in FeatureAlloy, while Δ-models may remove arbitrary elements. Another difference from those approaches is the relaxation of the one-to-one mapping between models and features, which in Δ-modelling is determined by feature presence conditions, which enables, for instance, the solving of feature interactions without explicitly creating a derivative feature.

3.2.3 *Annotation-based Languages*

Annotative approaches directly mark parts of the code with constraints over the features, for example, feature expressions. This is more practical than the composition-based approach for coding. For instance, `#ifdef` and `#endif` statements of C/C++ style preprocessors are traditionally used to encode variability in an *ad hoc* manner. These directives are used to control the inclusion of the code that belongs to the selected features or the exclusion of the code from the deselected ones. However, such annotation style obfuscates the code and makes it hard to understand and maintain.

Likewise composition-based approaches, many annotation-based approaches extend existing languages, in this case with feature annotations. fPromela (Cordy et al., 2013) extends Promela by introducing a feature variable over which transitions can be guarded; FMs, provided in TVL, are also integrated in subsequent analysis. Cordy et al. (2012) proposed feature timed automata – timed automata annotated with feature constraints – accompanied by an input language that extends HyTEC; FMs are also provided in TVL. Besides the composition-based extension already mentioned, Sabouri and Khosravi (2013) also proposed an annotative extension for the actor-based Rebeca language, where entities are annotated with application conditions. CL4SPL (Gnesi and Petrocchi, 2012) and FLan (ter Beek et al., 2013) are annotation-based approaches proposed in the context of process algebras and implemented in Maude.

There is considerable work on supporting variability modelling in conceptual modelling, namely focusing on UML-like models (see, for instance, (Jézéquel, 2012; Reinhartz-Berger and Sturm, 2014)). Approaches mostly propose lightweight extensions to UML using *profiles* – sets of stereotypes, tags and constraints – rather than actually extending the UML meta-model. A stereotype extends the vocabulary of a profile, defining how existing UML metaclasses can be extended. Tags are the specific properties defined for each stereotype, which are assigned values for stereotyped elements. An advantage of the use of standard extension mechanisms is the usage of the UML meta-model and its corresponding tools without any adaptation. Several of these approaches, however, do not view features as first-class artefacts, rather just annotating certain elements as variable and enforcing restrictions on their selection. For instance, Gomaa (2000) proposed a method to use predefined UML stereotypes to model variability in class diagrams. The «`Kernel`» and «`Optional`» stereotypes are used to represent commonality and variability in system families. Features are interpreted as optional elements that always appear together.

Nonetheless, there are still several approaches that explicitly consider FMs in the process. FeatuRSEB (Griss et al., 1998) includes a method to implement FD notation. Each feature node is implemented as a class with a stereotype «`feature`» and a feature name. An optional attribute is used to

indicate the kind of features, rationale, and notes. Use case diagrams are then merged into a single model with variation points denoting alternatives and «trace» abstractions used to map back to the FM. Clauß (2001) proposed to use the «variationPoint» and «variant» stereotypes in classes, components, packages, collaborations, and association elements. The «variationPoint» is used to describe the UML elements containing the variability and the «variant» specifies the alternatives to bind those variation points. The dependencies between variants are described by the «mutex» or «require» stereotypes. The FM follows the approach from (Griss et al., 1998), and variation points are connected to features using the «trace» abstraction. In (Robak et al., 2002) variable features in component and activity diagrams are specified by a new stereotype «variable» and with a tagged value named `feature` that maps back to a feature in the FM. In addition, variables can be annotated with further information to support choosing variants. Czarnecki and Antkiewicz (2005) propose fmp2rsm, connecting FMs specified in FeaturePlugin (Antkiewicz and Czarnecki, 2004) with UML models. Here, elements from class diagrams and activity diagrams can be annotated stereotypes representing presence conditions (propositional formulas over features) or meta-expressions (to compute feature-dependant values). In SMarty (Junior et al., 2010), besides «variationPoint» and «variant» stereotypes (the latter specialized into variants with additional restrictions imposed), a «variability» stereotype for comments is also introduced, which contains information regarding a feature. Use case, class and component diagrams are annotated with variability annotations, and the traceability between variation points and features registered. In COVAMOF (Sun et al., 2010) class, activity, sequence, and deployment diagrams are also annotated with «variationPoint» and «variant» stereotypes, and a Variation point Interaction Diagram (an extended UML class diagram) relates variation elements with features, here provided in an FM encoded in XVL. Devroey et al. (2012) propose SDVA, UML-like state diagrams annotated with feature constraints.

In the remainder of this section we present in more detail two annotation-based approaches that we feel are most relevant to the work developed in this thesis, namely fPromela for dynamic modelling and fmp2rsm for conceptual modelling.

fPromela

The fPromela language (Cordy et al., 2013) is a feature-oriented extension of Promela (Holzmann, 2003), a well-known formal modeling language used by the model checker SPIN. The syntax of fPromela is almost the same as Promela specification language. The key elements in fPromela are processes that describe behaviours of single units in the design. Such processes are specified with the **proctype** keyword. A

process is either declared **`active`** from the start or activated by another process. If several processes are active, their executions are interleaved, and communication performed through shared variables or explicit channels. Like Promela, fPromela supports five basic data types declared with keywords **`int`**, **`short`**, **`bit`**, **`byte`**, and **`bool`**. A collection of variables can also be declared using an array. The keyword **`mtype`** allows the introduction of symbolic values, that is, enumerated types. Control flow can be introduced, for instance, by **`do`** loops (which are only interrupted by **`break`** statements) or **`if`** statements. Their bodies contain several *options*, declared by a double colon (`::`). During execution, an option is non-deterministically chosen from those evaluating to true.

fPromela extends Promela by allowing the user explicitly declare features and then introduce guards over them to control the behaviour of processes. All features must be declared with a new special type **`features`**, which generally contains a Boolean field for each possible feature (although multi-features and numeric features are also supported). A feature variable is then declared with this type, which can be called in feature expressions. These expressions are used to guard statements using **`gd`** blocks. Feature expressions may only use feature variables or the **`else`** keyword, and this is the only place where features may occur. A product can execute a statement if and only if it satisfies the expression that guards it. FMs in TVL are provided to the analysis tool, SNIP, restricting which values can be assigned to the feature variables. Analysis itself is performed by FTS model checking (see Section 3.3) or by projecting and model checking each product individually. Assertions to be checked are written in feature LTL (see Section 3.3).

As an example, consider the specification of the vending machine example in fPromela in Fig. 35. Three features are declared in the **`features`** type, Cancel, Free and MultiSelection. A feature variable `f` is declared with this type. The valid states of the state machine are declared as an **`mtype`**. The behaviour of the state machine is then encoded in the process `vending`, which starts by initializing the variables and then entering a **`do`** loop. Each option of the loop tests the current `State` and assigns it a succeeding value. When these transitions depend on the selected features, guarded conditions are introduced in **`gd`** blocks depending on the value of feature variable `f`. Recall that if more than one of the options is true, e.g., when state is `Selected`, the choice will be non-deterministic. Since only transitions may be annotated with feature expressions, other elements, such as variable `cnt` for MultiSelection, must exist in every configuration. Assertions are not integrated in the model but rather passed as parameters of the model checking tool. For our example, we could check whether every time the machine is in the selected state it will eventually reach the ready state as follows.

```
$ ./snip -check -fm fm.tvl -filter 'Cancel' \
  -ltl '[] (State == Selected -> <> State == Ready)' -vending.pml
```

```
typedef features {
  bool Free;
  bool Cancel;
  bool MultiSelection;
};
features f;

mtype = {Ready, Paid, Selected, Served, Done}

active proctype vending() {
    int cnt = 1;
    mtype State = Ready;
    do :: State == Ready;
          gd :: f.Free;
                State = Selected;
             :: !f.Free;
                State = Paid;
          dg;
       :: State == Paid;
          State = Selected;
       :: State == Selected;
          gd :: f.Cancel && f.Free:
                State = Ready;
             :: f.Cancel & !f.Free;
                State = Done;
             :: f.MultiSelection;
                State = Selected;
                cnt = cnt+1;
             :: State = Served;
          dg;
       :: State == Served;
          gd :: f.Free;
                State = Ready;
             :: !f.Free;
                State = Done;
          dg;
          gd :: f.MultiSelection;
                cnt = 1;
          dg;
       :: State == Done;
          State = Ready;
    od;
}
```

Figure 35: Vending machine in fPromela with Free, Cancel and MultiSelection features.

In this example the trade-off between annotation- and composition-based approaches becomes clear. On the one hand, since the user directly specifies the superimposed model, it is easier to control the interactions between the features. For instance, the interaction between Free and Cancel is solved by introducing a guard `f.Cancel && f.Free`. On the other hand, there is no separation of concerns nor modular development, which may hinder the development process. Another issue is maintainability: as

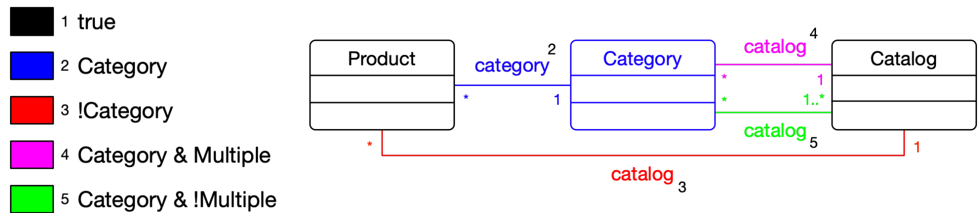


Figure 36: Annotated class diagram for the e-commerce catalog.

the number of features grow, the superimposed model may become cluttered with annotations, making it difficult to identify the effect of each feature.

fmp2rsm

One of the annotation-based approaches for model-based development that better fits our purpose is the one proposed by Czarnecki and Antkiewicz (2005), focusing on class and activity diagrams. A model family is comprised by an FM and annotated models – dubbed model templates in this approach. Annotations are either presence conditions (typically formulas over features) attached to model elements, or meta-expressions used to calculate attribute values. Some implicit presence conditions are inferred from the relationships between the elements (e.g., an association can only exist if the classifiers at its ends are also present, so their presence conditions are implicitly added to the association). The technique is realized in fmp2rsm, an Eclipse plug-in for feature-oriented modelling using UML. Annotations are realized as UML stereotypes and model templates colored according to their presence condition. Feature modelling is performed graphically and integrated in the plug-in.

As an example, consider the class diagram for the e-commerce catalog example, depicted in Fig. 36 in an abstract representation similar to that of (Czarnecki and Antkiewicz, 2005). The class diagram presents the superimposition of all possible elements, annotated with presence conditions. Elements in black are those present in every product, and in color we have elements that are associated with (different) conditions. Note that conditions may test the presence or absence of features, which may affect the readability of the diagram. For instance, the base model is not actually represented by the black elements, since association `catalog` of products, with presence condition `Categories`, is also present there.

As another example, consider the activity diagram for the vending machine example, depicted in Fig. 37. In this case, every state is present in every product, with only the transitions being affected by presence conditions. Again, this annotation-based approach allows a finer control over feature interactions, with the caveat of reduced readability as different presence conditions are created (recall that each color denotes

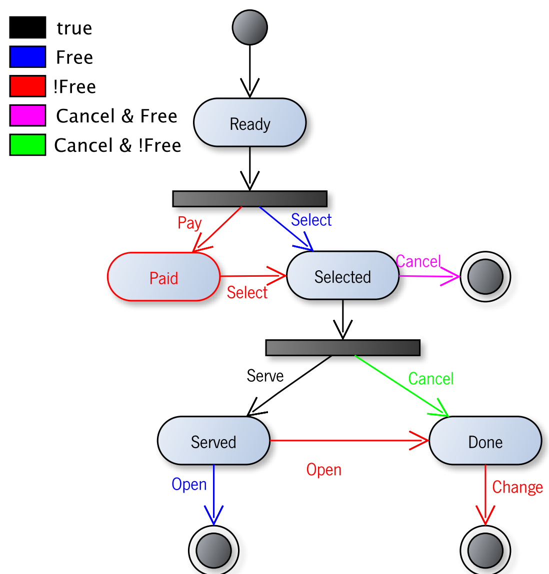


Figure 37: Annotated activity diagram for the vending machine.

a distinct presence condition and not a distinct feature). Notice that, again, approaches for conceptual modelling provide a finer level of granularity than those for richer languages.

3.3 Analysis in Feature-oriented Design

Most of the languages presented in the previous section support analysis via a naïve *enumerative* or *product-based* technique, that analyses each product separately. In this technique, a specification of a single product is first projected, and since it is specified in a standard general-purpose specification language, it can be analyzed with the tools available for the respective language. For example, in FeatureAlloy the final model obtained after composing features is written in plain Alloy and can be analyzed with the standard Alloy Analyzer. The main drawback of this approach is that it can be unfeasible for systems with many features, as the number of products grows exponentially with the number of features.

However, some research has been done on developing so-called *family-based* or *amalgamated* analysis techniques that check some property on all the product family at once, returning the violating products if the property does not hold for the all family. The key example of this approach is the model-checking technique initially developed for fSMV and fPromela, based on the so-called *feature transition systems*, which we will detail next. Other amalgamated approaches include those based on process algebras (Gnesi and Petrocchi, 2012; ter Beek et al., 2013), which expand input models into a transition system analyzable by Maude.

Feature Transition Systems FTSs, first introduced by Classen et al. (2010), are a well known low level formalism for modeling the behavior of SPLs. Essentially, FTSs extend classical labeled transition systems with feature annotations on transitions. Here we will present the definition of FTSs introduced in (Classen et al., 2011b), that generalized the initial definition by allowing transitions to be annotated with feature expressions (Boolean functions over features), rather than just single features.

Definition 3.3.1. *An FTS is a tuple* $(S, Act, trans, I, AP, L, d, \gamma)$, *where*

- S *is a set of states,*
- *Act is a set of actions,*
- *trans* $\subseteq S \times$ *Act* $\times S$ *is a set of transitions,*
- $I \subseteq S$ *is a set of initial states,*
- *AP is a set of atomic propositions,*
- $L : S \to 2^{AP}$ *is a labeling function,*
- d *is a feature model with features drawn from a set* N*,*
- γ : *trans* $\to (\{0,1\}^{|N|} \to \{0,1\})$ *is a total function that labels transitions with feature expressions.*

The concrete notation used to describe FMs is abstracted in this definition: if d is a feature diagram its semantics $[\![d]\!]_{FD} \subseteq P(N)$ captures the set of products in the SPL. Feature expressions are Boolean functions over the set of features N and allow greater expressiveness than a single feature. For example, transitions that are only included when feature a is present but not feature b can be labeled with expression $a \wedge \neg b$. In order to obtain the behavior of a particular product, one needs to remove all transitions whose feature expression is not valid in that product. This process is called *projection*. The projection of an FTS *fts* to a product p is denoted by $fts_{|p}$. All feature expressions are removed when projecting and $fts_{|p}$ is thus a normal transition system, whose semantics – the set of all valid paths – is denoted by $[\![fts_{|p}]\!]_{TS}$. The semantics of an FTS describes the behavior of a system with several products: the semantics of an FTS is the union of the behaviours of all the projections on all valid products.

Definition 3.3.2. *The semantics of an FTS fts, denoted as* $[\![fts]\!]_{FTS}$, *is defined as follows.*

$$[\![fts]\!]_{FTS} = \bigcup_{p \in [\![d]\!]_{FD}} [\![fts_{|p}]\!]_{TS}$$

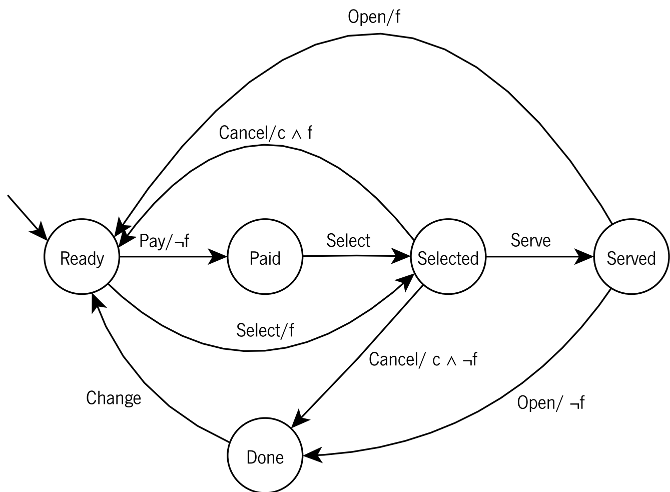


Figure 38: FTS of a vending machine.

Figure 38 presents an example of an FTS of a vending machine with two optional features: *free* drinks and *cancel* selection, abbreviated by f and c, respectively. A transition action and the respective feature expression are separated by a slash operator. Notice how feature expressions are used to control which actions are available at each state, and the result of a particular action. For example, in the initial state `Ready` if feature f is not present only `Pay` is possible, otherwise `Select` is immediately available. In state `Selected`, when feature c is present action `Cancel`, that cancels a selection, has a different outcome depending on feature f also being present or not – if products are free it causes a transition back to the initial state, otherwise if transitions to state `Done` where change still has to be returned to the client.

Specifying Properties with fLTL and fCTL The semantics of an FTS *fts* is a set of paths, being each path $\pi \in [\![fts]\!]_{FTS}$ an infinite sequence of states. Features are not present in paths, so standard temporal logics such as the *Linear Temporal Logic* (LTL) or the *Computation Tree Logic* (CTL) could be used to specify properties over FTSs. However, in (Classen et al., 2013) an extension of LTL has been proposed, denoted *feature LTL* (fLTL), that allows properties to be parameterized with a feature expression that captures the products for which they should hold. This parameter limits the range of products over which properties should be checked.

Definition 3.3.3. *An fLTL property φ is an expression $\varphi := [\chi]\phi$ where χ is a feature expression and ϕ is an LTL formula, i.e.,*

$$\phi ::= \top \mid a \in AP \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid \mathsf{X}\ \phi \mid \phi_1\ \mathsf{U}\ \phi_2$$

with the usual derived operators, for example, $\mathsf{F}\ \phi \equiv \top\ \mathsf{U}\ \phi$ *or* $\mathsf{G}\ \phi \equiv \neg\mathsf{F}\ \neg\phi$*.*

An FTS *fts* satisfies φ, denoted *fts* $\models \varphi$, iff all projections to valid products satisfy ϕ, that is $\forall p \in [\![d]\!]_{FD} \cap [\![\chi]\!] \cdot fts_{|p} \models \phi$. For example, in our example FTS of the vending machine, the fLTL formula $[\top]\ \mathsf{G}\ (\mathit{Selected} \to \mathsf{F}\ \mathit{Ready})$, specifies that, in all products, every time a product is selected the machine will eventually be ready again. To specify that, in all products without feature f, every time the product is paid it will eventually be served, we would use formula $[\neg f]\ \mathsf{G}\ (\mathit{Paid} \to \mathsf{F}\ \mathit{Served})$. Notice that this formula is not true, namely in products where feature c is present, because a selected product can be canceled and will not be served.

Similarly, *feature CTL* (fCTL) (Classen et al., 2014) extends CTL with feature expressions.

Definition 3.3.4. *An fCTL property φ is an expression $\varphi := [\chi]\phi \mid \varphi_1 \wedge \varphi_2$ where χ is a feature expression, φ_1 and φ_2 are fCTL formulae, and ϕ is a CTL property, i.e.,*

$$\phi ::= \top \mid a \in AP \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid \mathsf{EX}\ \phi \mid \phi_1\ \mathsf{EU}\ \phi_2 \mid \mathsf{EG}\ \phi$$

again with the usual derived operators.

Note that fCTL allows different top-level CTL formulas to be annotated with different feature expressions. When fCTL was initially introduced in (Classen et al., 2011b), feature expressions were allowed in all temporal operators, but that nesting of product quantification was later abandoned in (Classen et al., 2014) as it was seldom used in practice. The semantics of fCTL is defined similarly to that of fLTL by projecting on all valid products. As an example of an fCTL formula consider $[c]\ \mathsf{EG}\ \neg\mathit{Served}$ that states that in every product with feature c it is possible to never be served.

Model Checking fLTL and fCTL Model checking (Baier and Katoen, 2008) is a well-known automatic technique for verifying that a given temporal logic formula ϕ holds in a transition system *ts* that models the behaviour of a system, that is, to check that $ts \models \phi$. The model checking of SPLs differs from the classical model checking algorithms since it must consider all the behaviors of all variants of the family. In fact, given a set of N features it may have to consider up to $2^{|N|}$ variants. An SPL model checker to verify that an FTS *fts* satisfies formula φ, that is, to check if $fts \models \varphi$, should return *true* only when all the product instances satisfy the desired property. If that is not the case, a counterexample (π, ps) is returned, with a path π and a list of violating products $ps \subseteq [\![d]\!]_{FD}$ where that path is possible. In (Classen et al., 2010) an alternative model checking problem particularly relevant to SPLs is also defined, where the goal is to return a set of counter-example pairs such that every product where the property is violated is contained in at least one of the counter-examples.

As mentioned above, a simple and commonly used approach to SPL model checking is the enumerative method that verifies each variant individually. In this case, all products are projected to obtain normal transition systems and enable the application of standard model checking algorithms. This method does not take advantage of the commonality between different variants and is in principle highly inefficient due to the potential exponential number of variants. Consider, for example, the fLTL property $[\neg f]\ \mathsf{G}\ (\mathit{Paid} \rightarrow \mathsf{F}\ \mathit{Served})$. To apply the enumerative approach to model check this formula one would first project the FTS to all products that satisfy feature expression $\neg f$, and then apply a standard LTL model checking algorithm to verify if formula $\mathsf{G}\ (\mathit{Paid} \rightarrow \mathsf{F}\ \mathit{Served})$ is true in all the obtained transition systems. In this case we have two variants that satisfy the feature expression, namely the products without or with the possibility of cancelling a selection. The respective projected transition systems are presented in Figs. 39a and 39b, respectively. In this case the formula does not hold in the second transition system, being the violating product $\{c\}$ with a counter-example path that cycles indefinitely through states `Ready`, `Paid`, `Selected`, and `Done`.

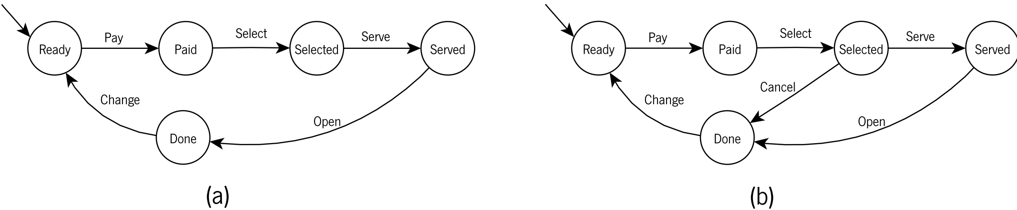


Figure 39: Projected transition systems for the vending machine.

In (Classen et al., 2010, 2012) a semi-symbolic model checking algorithm was introduced to perform amalgamated analysis of FTSs (for both the simple model checking problem and the extended one that returns counter-examples for all violating products). The algorithm relied on a generalization of the explicit-state *Depth-First Search* used in model-checking normal transition systems, but where visited states are marked with feature expressions instead of Boolean flags. This algorithm was implemented in the model checking tool SNIP to verify fPromela models. This algorithm was shown to speed up considerably the model checking of properties when compared with the enumerative approach. For example, for an FTS with 9 features and 64 products it achieved average speedups of 3.5 Classen et al. (2010).

Unfortunately, like all explicit-state model checking algorithms, this technique is prone to the state explosion problem. To mitigate this problem, an amalgamated symbolic model checking algorithm was later proposed for fCTL (Classen et al., 2011b, 2014), and used to verify fSMV specifications. Essentially, this algorithm reduces SPL model checking to classic model checking and enables the usage of the standard symbolic model checker NuSMV. The idea is to enrich the state definition with Boolean variables to encode

all the possible features, a technique denoted as *feature lifting*. The value of these feature variables is kept constant in all transitions, and the feature expressions associated with transitions in the FTS become guards defined over them. If a counter-example is obtained, the initial state of the feature variables is inspected to obtain a violating product. For our example, the verification of fCTL property $[c]$ EG $\neg$*Served* could be done by model checking the amalgamated SMV model of Fig. 40. This symbolic SPL model checking technique can achieve order-of-magnitude speedups over the enumerative approach: in an fSMV model of an elevator system with 9 independent features (that is, 2^9 variants) the verification of some properties has only marginally faster then the enumerative approach, but for other properties it achieved speedups greater than 250 and up to 1000 (Classen et al., 2011b). As expected, the higher speedups occurred in true properties, for which all the variants had to be checked. However, the enumerative approach was competitive in some false properties, most likely in cases where most products yield a counter-example.

```
MODULE main
VAR
  f : boolean;
  c : boolean;
  State : Ready, Paid, Selected, Served, Done;
ASSIGN
  init(State) := false;
  next(State) := case State = Ready & !f: Paid;
                      State = Ready & f : Selected;
                      State = Paid : Selected;
                      State = Selected & c & f : {Ready,Served};
                      State = Selected & c & !f : {Done,Served};
                      State = Selected & !c : Served;
                      State = Served & !f : Done;
                      State = Served & f : Ready;
                      State = Done : Ready;
                 esac;
CTLSPEC c -> EF !(State = Served)
```

Figure 40: SMV model for the amalgamated verification of the vending machine example.

3.4 Supporting Clone-and-own

A code clone is a piece of code fragment that occurs in multiple locations with the same or similar form. Usually, a clone appears when a developer copies a piece of code from one location to another and afterwards customizes the copy to address new requirements. This phenomenon, which is called *clone-and-own* occurs commonly in the software development process. Clones can be classified into 4 types according to their similarity, namely Type-1 to Type-4 clones (Bellon et al., 2007). The Type-1 clones are

exact copies of each other. Everything is exactly the same in the two clones. The Type-2 clones subsume all Type-1 clones, but also allow slight modification, such as variable renaming. The Type-3 clones subsume all Type-2 clones but include bigger changes, such as adding or removing statements, to implement some product-specific requirements. The Type-4 clones, also called functional clones, implement similar functionality but have few or no code similarity. The clone-and-own technique is well-supported by version-control systems, such as Git, via operations like forking, merging, and pull requests. Since clone-and-own offers a simple, intuitive, and time and cost-saving technique to develop new software variants, it is well accepted by industry as a way to develop SPLs (Dubinsky et al., 2013).

However, clone-and-own leads to *ad hoc* product portfolios with multiple, yet similar, variants, with no explicit connections, which brings significant increase in the cost and effort of maintenance. In particular, changes such as bug fixes must be carefully synchronized in all variants. Unfortunately, reuse tracking in clone-and-own is typically embodied in the personal knowledge of the developer (Dubinsky et al., 2013), which further difficults change propagation between clones.

In feature-oriented software design, the clone-and-own approach is also a simple way to create a new variant of a model in an SPL, and to explore the impact of different features while exploring the design of a software system. The basic idea is again to derive a new variant of a design by copying from an existing one and then adapting to accommodate the new requirements. Likewise to code, anecdotal evidence seems to indicate that clone-and-own is also common when developing software models (Störrle, 2013). In particular, in Alloy this is a common approach, for example, in several of the case-studies that come pre-packaged in the Analyzer there are multiple variants that share a great percentage of code. Some of these case-studies will be used in the Chapter 6 to evaluate the proposed clone migration technique.

3.4.1 *Migrating Clones into an SPL*

Maintenance of SPLs developed with clone-and-own is a big challenge. As the number of clones grows and differences between their implementations increases, it becomes harder to perform many of the tasks needed in SPL engineering, for example, keeping track of changes made to each variant and of which features are shared between specific clones, reconcile changes, or derive new variants. A good way to solve this maintainability problem is to migrate (by merging) all variants into a single SPL model, where the common parts of the variants are factored out and implemented only once. This is also known as an *extractive* approach to SPL engineering. With this single model, changes of the systems only need to be

implemented once, which provides significant benefits for management and synchronization, with minimal workload in the subsequent maintenance and evolution of the design.

Many techniques have been proposed to migrate product variants into managed SPLs, as detailed in the survey conducted by Assunção et al. (2017). Most of these techniques work at the code level, and only a few have been proposed specifically for design models (although many techniques developed for object-oriented programs could in principle be adapted for models such as UML class diagrams). In this section we review a couple of techniques that have been specifically developed or applied to design models.

Rubin et al. (2015) proposed a high-level approach based on seven conceptual operators that can be composed to implement several complex development activities related to SPL engineering, namely migrating clones into an SPL, in this work denoted as *merge-refactoring*. These operators can be applied to all possible artifacts such as requirements documents, design models, source code, and so on, and the authors illustrate the technique by applying it to transition systems, merge-refactored to a single model akin to an FTS. The seven proposed operators are:

findFeatures that identifies a set of features in a given variant;

findFeatureImplementation that locates the implementation artifacts of a given feature in a variant, thus establishing a trace between features and their artifacts;

dependsOn? that checks if one feature requires another in a given variant;

same? that determines whether the functionality described by one feature in one variant is consistent with the functionality described by another feature in a possibly different variant;

interact? that checks whether the combining functionalities described by a set of features would alter the behavior of one or more of those functionalities;

merge that combines several input variants into a single system; this operator has two parameters: *matches*, that specifies which artifacts are considered similar and should be combined in the resulting system, and *resolution* that indicates how to resolve disagreements and interactions between the input features (for example, keep both implementations as separate functionalities or override one feature implementation by the other);

reorganize that improves the structure of a system (either a single variant or a combined SPL after merge) by refactoring, without modifying its behaviour.

```
sig Product {
    catalog : some Catalog,
    images : set Image
}
sig Image {}
sig Catalog {
    thumbnails : set Image
}
fact Thumbnails {
    all c:Catalog | c.thumbnails in (catalog.c).image
}
pred Scenario {
    some Product.images
}
run Scenario for 2
```

Figure 41: Clone Alloy model of an e-commerce platform with thumbnails.

These operators can be implemented and composed in many ways to achieve different SPL engineering tasks. The implementation of the operators is left open and is highly dependent on the kind of targeted artifacts and systems. Next we will illustrate this technique using an Alloy implementation of the e-commerce example, possibly obtained by clone-and-own, with the goal of merging it into a single *ad hoc* SPL Alloy model. One of the clones will be the base model of e-commerce presented in Fig. 28. Suppose this model was cloned and adapted to implement a variant that also supports thumbnails, originating the clone presented in Fig. 41.

As proposed in (Rubin et al., 2015), to migrate these clones into a single model, we can start by identifying the features and the respective implementation in each variant with findFeatures and findFeatureImplementation. In our case we could, for example, assume that each signature corresponds to a possible feature, with the respective declaration as implementation artifact. Then, for each pair of features in a variant, dependsOn? should be used to identify require relationships. In our example, we could assume that one feature requires another if the fields declared in the former signature mention the latter. Then for each pair of features in different variants operator same? should be used to identify if two features are similar, for example, we could consider they are similar only if the respective signature declaration is literally identical. Then, possible interactions are detected with interact?. To implement this operator we could consider the existence of any additional facts, such as `Thumbnails`, to possibly cause interactions between the implementation of two variants. The witnesses collected by these two operators are then fed to the merge operator, namely the features identified by same? will be passed on to the *matches* parameter, leaving only one implementation of the respective signature in the merged model, and for those that disagree or interact the respective implementation could be kept separate, for example

```
abstract sig Feature {}
lone sig ProductImages, CatalogEcommerce, CatalogThumbnails extends
    Feature {}
fact dependsOn {
    some CatalogEcommerce implies some ProductImages
    some CatalogThumbnails implies some ProductImages
}
sig Product {
    catalog : some CatalogA+CatalogB,
    images : set Image
}
sig Image {}
sig CatalogA {}
sig CatalogB {
    thumbnails : set Image
}
fact Disagreements {
    some CatalogEcommerce iff no CatalogThumbnails
    no CatalogEcommerce implies no CatalogA
    no CatalogThumbnails implies no CatalogB
}
fact Thumbnails {
    some CatalogThumbnails implies
    all c:CatalogB | c.thumbnails in (catalog.c).image
}
pred Scenario {
    some Product.images
}
run Scenario for 2
```

Figure 42: Possible result of merging clones with the approach proposed by (Rubin et al., 2015).

declaring additional signatures to distinguish the implementations in different variants, and making their existence conditional to the presence of atoms identifying the respective features. Finally, the reorganize operator could run an algorithm to detect atomic sets of features, to simplify the encoding of the FM. A possible resulting Alloy model is presented in Fig. 42.

Rubin and Chechik (2012) proposed a step-wise merge-refactoring operation, denoted *merge-in*, that can be used for migrating cloned models to an annotative SPL model. In this work, models are trees of typed elements, each with an id and a role that defines the relationship with its parent. For example, in an UML statechart, possible types for model elements are state, transition, or a reference to a state. As for roles, for example, elements of the latter type can either be the source or target of the parent transition. Each model element can further be annotated with the features where it is present.

Central to this technique is the operation of *model merging*, which is divided into three steps: compare, match and compose. The *compare* operation is used to determine the similarity degree between pairs of model elements. The similarity degree is a number between 0 and 1, with 1 representing identical

elements and 0 meaning the two elements have no similarity. The *match* operation detects pair of elements that should be later considered similar in the merge step. The *merge* function puts the information in two models together, keeping a single copy for for matched elements.

The compare operation between two model elements should take into consideration their types and roles in the model, as well as a weighted sum of the similarity of their sub-elements. Elements with different types or roles should always have similarity 0, and different weights can be chosen by the user for the different roles. The implementation of the match operation is based on thresholds: for each type a different threshold can be defined and pairs of elements whose comparison yields a value equal or higher than that threshold are considered similar. The merge function returns a model that contains all elements of the input models, with matched elements unified and appearing in the resulting model only once.

The *merge-in* operation merges a product model with an annotative SPL model. The idea is to iterate this operation once per clone product, starting with an empty SPL model, and adding clones one by one until the final merged SPL model is obtained. As for the features of the SPL, different products are considered as distinct features. Hence, the merge-in operation will just add a new alternative feature to all existing features already in its FM. Then, the input product model is merged with the existing SPL model, and its elements are annotated with the respective feature. The authors proved that this technique is a behavior-preserving product line migration strategy and that only the original model clones can be projected from the merged SPL.

However, by varying the compare and match parameters, as well as the order in which input models are combined, the resulting SPL model can be quite different. Although all possible results are correct and semantically equivalent, not all may be equally desirable. As such, Rubin and Chechik (2013) extended this technique to allow the user to specify a desired quality metric for the outcome, which is then used as an objective function to automatically explore different alternatives (until a specific desired quality is reached or a maximum number of alternatives is explored, after which the best alternative is return). Possible quality metrics are, for example, model size or the percentage of common elements in the resulting model.

The ModelVars2SPL technique (Assunção et al., 2020) was proposed for automatically extracting an FM and an annotative SPL model, denoted *Product Line Architecture* (PLA), from existing UML class diagram variants. The input of ModelVars2SPL is a set of variants, each of which consists of a UML class diagram describing the structure of the variant and a feature set represent the configuration of features provided by the variant. The extraction proceeds in four steps:

1. Identify feature traceability, i.e. which elements implement each feature. This is done by analyzing the overlaps between the model elements and feature sets of different variants. This step outputs

the traceability links between model elements and features and a dependency graph that represents the relationship between features.

2. Apply reverse engineering, using a multi-objective search-based technique, to obtain an FM that best represents the feature sets and that respects the dependency graph obtained in the previous step.

3. Apply a search-based technique for model merging the various UML diagrams into a single model that contains all possible model elements present in the variants.

4. Graft variability annotations in the merged class diagram obtained in the previous step in order to produce a PLA. Variability annotation is done by adding a UML comment to each model element.

All these techniques focus only on structural elements of the design, for example classes or relationships, and none addresses additional constraints, where variability can occur with finer granularity. When illustrating the technique proposed by Rubin et al. (2015) with our running Alloy example, we considered fact `Thumbnails` as a whole unit: it is not clear how to apply that technique (or the others) to merge, for example, the two distinct versions of that fact that would exist in the clones implementing variants with and without feature Categories (merging in a way that highlights the commonalities between them).

4

COLORFUL ALLOY

In this chapter we propose an extension of the popular Alloy specification language and its Analyzer to support feature-oriented software design. As we have seen in the previous chapter, most techniques for feature-oriented software development fall into one of two categories: compositional approaches, that implement features as distinct modules and use some sort of module composition technique to generate a specific software variant; and annotative approaches, that implement features with explicit (or sometimes implicit) annotations in the source code, that dictate which code fragments will be present in a specific variant. The former are well suited to support coarse-grained extensions, for example adding a complete new class to implement a particular feature, but not to support fine-grained extensions, for example adding a sentence to a method or change the expression in a conditional, to affect the way a code fragment works with different features (Kästner et al., 2008). Annotative approaches are much better suited for such fine-grained variability points.

Unfortunately, as we have also seen, explicit support for feature-oriented design in formal methods, providing a uniform formalism for feature, architectural, and behavioral modeling as advocated for SPL engineering (Schaefer and Hähnle, 2011), is still scarce. Support for features in model checking has been proposed, namely fSMV (Plath and Ryan, 2001) and fPromela (Classen et al., 2012). For structural design, a compositional approach has been proposed to explicit support features in Alloy (Apel et al., 2010). Typically, modeling and specifying in Alloy is done at high levels of abstraction, and adding a feature can require only minimal and very precise changes (e.g., adding one new relation to the model or changing part of the specification of a desired property). Compositional approaches such as the one proposed by Apel et al. (2010) are not well suited for these fine-grained extensions. Our Alloy extension addresses precisely this problem, proposing an annotative approach to add explicit support for features to Alloy and its Analyzer.

A classic annotative approach for source code is the use of `#ifdef` and `#endif` C/C++ compiler preprocessor directives to delimit code fragments that implement a specific feature. Unfortunately, such annotation style obfuscates the code and makes it hard to understand and maintain, leading to the well-

known #ifdef hell (Feigenspan et al., 2013). To alleviate this problem, while retaining the advantages of annotative approaches, Kästner et al. (2008) proposed to annotate code fragments associated with different features with different background colors, which was later shown to clearly improve SPL code comprehension and be favored by developers (Feigenspan et al., 2013). Given these results, we propose to use such annotative technique to support features in Alloy. Our *Colorful Alloy* extension allows users to annotate model and specification fragments with different background colors denoting different features[1], and run analysis commands to verify either a particular variant of the design, or several variants at once, simplifying the detection of feature combinations that may fail to satisfy the desired specification. To the best of our knowledge, this is the first annotative approach for feature support in a formal method geared towards structural design.

The next section gives a detailed description of the color background approach, first proposed by Kästner et al. (2008), and which inspired the development of Colorful Alloy. Section 4.2 formally presents the syntax of the new language extension, and Section 4.3 presents an example of proactively designing an SPL with Colorful Alloy. Section 4.4 presents its typing rules, and Section 4.5 its semantics. Finally, Section 4.6 presents a technique to perform multi-variant analysis, inspired by the feature lifting technique used proposed by Classen et al. (2011b, 2014) to model check fCTL.

4.1 The Background Color Approach

The background color approach was proposed by Kästner et al. (2008), as a mechanism to improve SPL code comprehension. The idea is to annotate the code fragments belonging to different features with different background colors. In case a code fragment is associated with multiple features (for example code within nested preprocessor #ifdef statements), the background colors associated with the respective features are mixed. The code snippet presented in Fig. 43 illustrates how background colors can be used to highlight feature code. This is an example where #ifdef preprocessor directives are used to annotate the code of different features. The code associated with feature *HAVE_QUEUE* (l. 5–16) is annotated with the yellow background color, while the code associated with feature *DIAGNOSTIC* (l. 12–14) is annotated with orange. The latter code is nested inside feature HAVE_QUEUE, so orange is the result of mixing the color yellow from feature HAVE_QUEUE with the red color that would be used to annotate code belonging only to feature DIAGNOSTIC. Notice also, that the same color is used to annotate code in the #ifndef and #else branches of feature HAVE_QUEUE, because both pieces of code are relevant for that feature:

1 Unlike the technique proposed by Czarnecki and Antkiewicz (2005), where different colors represent different presence conditions.

```
1  static int __rep_queue_filedone(dbenv,rep,rfp)
2    DB_ENV *dbenv;
3    REP *rep
4    __re_fileinfo_args *rfp;{
5    #ifndef HAVE_QUEUE
6    COMPQUIET(rep,NULL);
7    return (__db_no_queue_am(dbenv));
8    #else
9    db_pgno_t first,last;
10   u_int32_t flags;
11   int empty,ret,t_ret;
12   #ifdef DIAGNOSTIC
13   DB_MSGBUF mb;
14   #endif
15   ... //over 100 lines of additional code
16   #endif
17 }
```

Figure 43: Excerpt of Berkeley DB with background colors (Feigenspan et al., 2013).

the former is included when the feature is not selected, while the latter is included when it is selected. This example shows how colors, in principle, help the developers distinguish code associated with different features at first sight, rather than requiring a search for the beginning and end annotations, which can be difficult, especially in a large code base.

A tool to support the background color approach, named *Colored Integrated Development Environment* (CIDE), was proposed by Kästner et al. (2008) to help users decompose large legacy application into features. CIDE is an Eclipse-based prototype tool, that besides supporting background colors, provides code folding of feature code (hiding the source code of selected features), and provides different views on the source code. Unlike most code annotation-based approaches it associates code fragments from different features with distinct background colors, rather than relying on `#ifdef` preprocessor directives. A screenshot of CIDE is shown in Fig. 44. As with typical annotative approaches, in this approach a system is implemented with the code of all features together in a single code base. Instead of associating the code fragments with features using markers or delimiters, it does so by associating the background color with the representation layer of the editor. In the case of a code fragment from multiple features, which is traditionally done with nested preprocessor statements as shown in Fig. 43, a mix of the corresponding background colors is chosen. It may not be possible to recognize the features of a code fragment only by background colors in CIDE, in particular when many features overlap, which is common in SPL. However, colors make it easier for developers to quickly identify the beginning and the end of a code fragment associated with a set of features, which can then be confirmed by inspection using tool-tips.

Unlike code annotated with `#ifdef` directives, CIDE does not allow to assign features to arbitrary code fragments, avoiding the problem of feature annotation errors, such as a pair of corresponding opening and

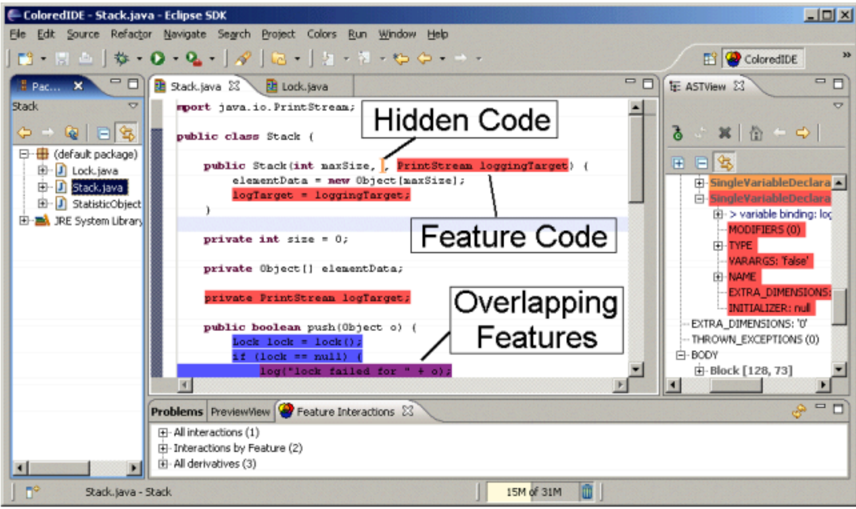


Figure 44: CIDE screenshot (Kästner et al., 2008).

ending braces annotated with different features. Tracking such kind of errors directly at source code is difficult since both brackets are visible. Instead, CIDE assigns features to structural code elements in an *Abstract Syntax Tree* (AST) representation of the source code. For example, a feature can be assigned to a class node (making that class optional) or to a statement node. Children nodes are annotated automatically if the parent node is also annotated with a particular feature. This simplifies detecting annotation errors and removing code when projecting to obtain the implementation of a particular variant. Also, by annotating structural elements, the developers do not need to deal with syntactical elements that are abstracted away in the AST, like commas separating the parameters of a method, which significantly reduces the size of the final code. For example, suppose we have a method with two parameters annotated with two distinct features using `#ifdef` directives. In order to make sure there are no syntactic errors for every projected variant, we have to annotate the separating comma with a nested annotation, to make sure it is included only when both features are selected. However, annotating only structural elements in the AST also has some limitations: in principle only optional elements of the AST can be annotated with features, which for example does not allow to specify two alternative return statements for a method. There are some exceptions to the general rules that children are automatically annotated with their parent's features, and that only optional AST nodes can be annotated. Concerning the former, it is possible to annotate a control flow statement without annotating its inner code (for example, to introduce an if statement to guard the execution of some code in a variant). As for the latter, children of binary operators can be individually annotated even if they are not optional. If only one child is annotated the operator will be removed when the feature is not selected.

Annotations in CIDE are just feature sets, and not arbitrary presence conditions involving those features. In particular, negative features are not supported. CIDE does not deal directly with consistency constraints nor uses a FM to restrict the set of possible variants, but can check the feature association of all AST elements to ensure that every possible projected variant can be correctly parsed and compiled.

Later, some of the authors of CIDE and others conducted a study to evaluate whether and how background colors improve program comprehension of code annotated with `#ifdef` preprocessor directives (Feigenspan et al., 2013). This study involved three controlled experiments with a total of 77 subjects. The first experiment aimed to understand if colors improve program comprehension in a medium sized software system with four optional features, and asked the users to perform a series of tasks, such as locating bugs that were only present in some features. The second experiment aimed to understand if users prefer to use the background color or the preprocessor directives when given the choice, using the same tasks as the first experiment. Finally, the third experiment aimed to understand if the background color approach scales to large software systems, namely systems with more than 100k lines of source code and hundreds of features. Since color mixing does not scale to this number of features, in this experiment a nested feature was annotated with just its color. The main conclusions of these experiments were that:

- Carefully chosen colors can improve program comprehension, independently of code size and programming language of the underlying SPL, as a performance increase was observed in the assigned tasks when the background color approach was used.
- Choosing colors with a high saturation can slow down the comprehension process, probably due to visual fatigue.
- In general, subjects liked and preferred the background color approach.

Based on the insights collected in this study, a new tool was developed to support the comprehension of SPL code devolved with preprocessor directives, named FeatureCommander[2]. This tool already supports FMs, and allows users to choose which colors and opacity to assign to each feature (to avoid high saturation colors). Since it is intended to be applied to large code bases, if a code fragment is annotated with multiple features, only the background color of the innermost feature is shown, being the set of all features visualized in side bars. Also, to allow the user to focus only on a particular set of features, the remaining ones can just be assigned shades of gray to not pollute the visualization. A screenshot of this tool can be seen on Fig. 45.

2 https://www.tu-chemnitz.de/informatik/ST/research/material/xenomai/

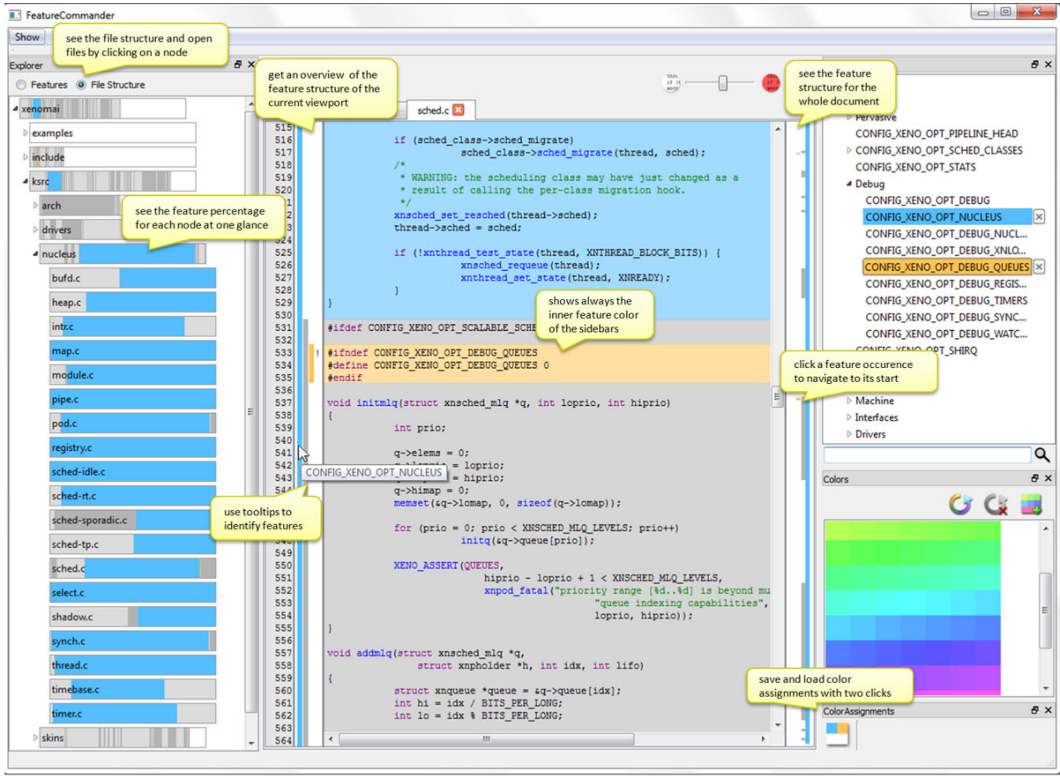


Figure 45: FeatureCommander screenshot (Kästner et al., 2008)

4.2 Colorful Alloy Syntax

One of the reasons behind the initial proposal of color annotations in CIDE was to avoid obfuscating the code with additional constructs (Kästner et al., 2008). There, colors are internally handled by an IDE developed specifically for that purpose, which hinders saving, sharing and editing models, particularly when dealing with simple, single-file, models as is typical in Alloy. Our approach aims at a middle ground, using minimal annotations that are colored when using the Analyzer, but that can still be saved and interpreted as a pure text file. Additionally, unlike in CIDE, our language allows elements to be marked with the absence of features. Thus, although not allowing full propositional formulas, elements can be assigned a present condition consisting of a conjunction of positive or negative features.

The Colorful Alloy language is thus a minimal extension to regular Alloy mainly by allowing, first, elements to be associated with the presence or absence of features; and second, analysis commands to focus on particular sets of features. Features are identified by circled symbols, $\textcircled{c}$ and $\textcircled{\textbf{c}}$, denoting the presence and absence of a feature, respectively, for $1 \leq c \leq 9$ (throughout the thesis, symbol $\textcircled{c}$ will denote either $\textcircled{c}$ or $\textcircled{\textbf{c}}$, but it is not itself an acceptable annotation). To ease understanding, the Colorful Alloy extension to the Alloy Analyzer employs background colors (for positive annotations) and colored strike-through lines

```
spec         ::= [ moduleDecl ] import* paragraph*
moduleDecl   ::= module qualName [ [ name,+ ] ]
import       ::= ⓒ open qualName [ [ qualName,+ ] ] [ as name ] ⓒ
paragraph    ::= colPara | cmdDecl
colPara      ::= ⓒ colPara ⓒ | sigDecl | factDecl | funDecl | predDecl | assertDecl
sigDecl      ::= [ abstract ] [ mult ] sig name,+ [ sigExt ] { colDecl,* } [ block ]
sigExt       ::= extends qualName | in qualName [ + qualName ]*
mult         ::= lone | some | one
decl         ::= [ disj ] name,+ : [ disj ] expr
colDecl      ::= ⓒ colDecl ⓒ | decl
factDecl     ::= fact [ name ] block
assertDecl   ::= assert [ name ] block
funDecl      ::= fun name [ [ decl,* ] ] : expr block
predDecl     ::= pred name [ [ decl,* ] ] block
expr         ::= const | qualName | @name | this | unOp expr | expr binOp expr
                | colExpr colBinOp colExpr | expr arrowOp expr | expr [ expr,* ]
                | expr [ ! | not ] compareOp expr | expr ( ⇒ | implies ) expr else expr
                | quant decl,+ blockOrBar | ( expr ) | block | { decl,+ blockOrBar }
colExpr      ::= ⓒ colExpr ⓒ | expr
const        ::= none | univ | iden
unOp         ::= ! | not | no | mult | set | ~ | * | ^
binOp        ::= ⇔ | iff | ⇒ | implies | – | ++ | <: | :> | .
colBinOp     ::= || | or | && | and | + | &
arrowOp      ::= [ mult | set ] → [ mult | set ]
compareOp    ::= in | =
letDecl      ::= name = expr
block        ::= { colExpr* }
blockOrBar   ::= block | | expr
quant        ::= all | no | mult
cmdDecl      ::= [ check | run ] [ qualName ] ( qualName | block ) [ colScope ] [ typeScopes ]
typeScopes   ::= for number [ but typeScope,+ ] | for typeScope,+
typeScope    ::= [ exactly ] number qualName
colScope     ::= with [ exactly ] [ ⓪ | ⓒ ],+
qualName     ::= [ this/ ] ( name/ )* name
```

Figure 46: Concrete syntax of the Colorful Alloy language (additions w.r.t. the Alloy syntax are colored red).

(for negative features) to highlight annotated elements. This allows models with at most 9 distinct features, which we believe to be adequate for Alloy, where models are typically small and defined at a high-level of abstraction. Also, the full coloring approach, where all feature code is background colored and code associated with multiple features has blended colors, is known to not scale up well to models with a large number of features (Feigenspan et al., 2013). This problem would be exacerbated with negative features. To support that scenario, as in FeatureCommander, we would have to allow the user to select which features to color at any time, which would require a more complex extension to the Alloy Analyzer. Also, as mentioned above, one of our goals was to, unlike CIDE, allow users to save models in pure text files, and this restriction allows us to use the respective UTF characters with circled numbers to represent feature annotations, meaning the annotated models can still be inspected and reasonably understood in a normal text editor that does not provide feature color highlighting. Figure 46 presents the syntax of Colorful Alloy, highlighting changes with regard to the regular Alloy language.

Features are associated to model elements by using feature marks as delimiters surrounding those elements. An element within a positive delimiter ⓒ will only exist in variants where c is selected, while

those within a negative delimiter 🅒 only exist if c is absent from the variant. Color annotations can be nested, denoting the conjunction of presence conditions. For example, with ②❶ϕ❶②, formula ϕ will be present in any variant with feature 2 selected but not feature 1. Likewise (Kästner et al., 2008), in general only optional elements of the Alloy AST can be annotated. In general, any node whose removal does not invalidate the AST can be marked with features, including all global declarations (i.e., signatures, fields, facts, predicates, functions and asserts) and individual formulas within blocks. Currently, only non-annotated modules can be imported, such as the libraries packaged with the standard Analyzer. The marking of local declarations (i.e., predicate and function arguments, and quantified variables) is left as future work. One exception to the AST validity rule is allowed for binary expressions with a neutral element, in which case the sub-expressions can be annotated even if the whole binary expression is not. For instance, ❷Φ❷+②Ψ② is interpreted just as Φ in variants where feature 2 is not selected.

Like in Alloy, *run* commands can be declared to animate the model under certain properties and *check* commands to verify assertions, both within a certain universe of atoms specified by a scope. In Colorful Alloy, a scope on features may also be provided to restrict the variants that should be considered by a command. Run and check commands can be instructed to focus, possibly **`exactly`**, on certain features using a **`with`** *feature scope*: if not exact, commands will consider every variant where the positive/negative features specified in this scope are present/absent; otherwise, exactly the variant with the presence features will be considered (negative features are spurious in that case). For instance, **`run {} with`** ①,❷ will consider every variant with feature 1 selected but not feature 2, while **`run {} with exactly`** ①,❷ will only consider the variant with exactly feature 1 selected. An additional feature mark ⓪ denotes the empty variant (no features selected), and can be used to analyze every possible variant (if the feature scope is not exact), the default behavior if a feature scope is not provided, or solely the base variant (if the feature scope is exact), in this case being equivalent to having all negative features in the scope.

4.3 An Example of Proactive SPL Design

To illustrate Colorful Alloy we will use it to formalize the design of multiple variants of the e-commerce example, that was used to illustrate Alloy in Chapter 2. The FD of the full e-commerce SPL was depicted in Fig. 18. Here we will consider a simpler version of this SPL with just three optional features: ① allowing products to be classified in *categories*; ② allowing *hierarchical* categories; and ③ allowing products to belong to *multiple* categories. Thumbnails exist on all variants and image formats will not be considered, nor on sale products. Not all combinations of these three features are valid, namely both hierarchical

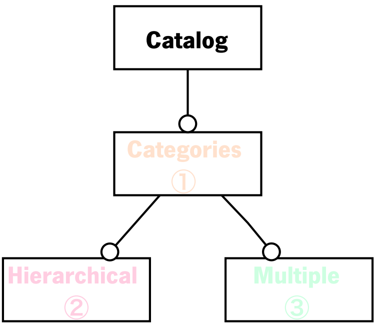


Figure 47: Feature diagram of the colorful e-commerce specification.

categories and multiple categories require the existence of categories. The FD of this reduced e-commerce SPL is depicted in Fig. 47, already depicting the feature numbers and the respective background colors.

The Colorful Alloy specification of this example is shown in Fig. 48. To avoid deviating a lot from the standard Alloy syntax, Colorful Alloy does not explicitly support FMs, but the user can still restrict valid variants using normal facts. In Fig. 48 fact `FeatureModel` (l. 1–6) encodes the FD of Fig. 47, forcing feature ① to be selected in a variant whenever either ② or ③ are. This is achieved by introducing an inconsistency in the model if either feature ② or ③ is selected together with ➊. Alloy does not have a keyword to denote *False*, but there are several expressions that can be used instead, for example **`some none`**. In line 3, this formula is annotated both with ② and ➊, meaning that whenever feature Hierarchical is selected without feature Categories, that expression would be introduced in the model, thus creating an inconsistency. A similar technique is used in line 5 for forcing feature Categories to be selected when feature Multiple is as well.

When proactively specifying a design with Colorful Alloy, we usually start by defining the base model, and incrementally add or remove elements (with the respective annotations) to support optional features. In this case, the base model declares signatures `Product` (l. 8–13), `Image` (l. 14), and `Catalog` (l. 15–17). It also declares fields `images` (l. 9), to relate products with their images, `catalog` (l. 10), to relate products with their catalog, and `thumbnails` (l. 16), to relate catalogs with the images that will illustrate them. This base model also includes a fact to force the thumbnails of a catalog to be images of one of its products (l. 19), and a command to search for a scenario where there are some products with images (l. 33–36).

After specifying this base model, we proceed by specifying the changes introduced by feature ①, allowing products to be classified in categories, instead of being directly placed in a catalog. This introduces two main changes: field `catalog` should be replaced by `category`, to relate products with one category, and a new signature `Category` should be declared. To remove `catalog` when feature ① is selected it suffices to annotate it with ➊. On the other hand, the new field `category` should be annotated with

```
fact FeatureModel {
  --② Hierarchical requires ① Categories
  ②①some none①②
  --③ Multiple requires ① Categories
  ③①some none①③
}

sig Product {
  images: set Image,
  ①catalog: one Catalog①,
  ①③category: one Category③①,
  ①③category: some Category③①
}
sig Image {}
sig Catalog {
  thumbnails: set Image
}
fact Thumbnails {
  ①all c:Catalog | c.thumbnails in (catalog.c).images①
  ①all c:Catalog | c.thumbnails in (category.(②inside②+②^inside②).c).images①
}

①②sig Category {
  inside: one Catalog
}②①
①②sig Category {
  inside: one Catalog+Category
}②①
①②fact Acyclic {
  all c:Category | c not in c.^inside
}②①

pred Scenario {
  some Product.images and ①all c:Category | lone category.c①
}
run Scenario for 10

assert AllCataloged {
  ②all p:Product | some (p.category.^inside & Catalog)②
}
check AllCataloged with ①,② for 10
```

Figure 48: E-commerce specification in Colorful Alloy.

① to force its inclusion with this feature. Similarly, the new signature `Category` declaration should be annotated with ①. Inside this signature we should also declare the new field `inside`, that relates it to the respective catalog. After this changes, the declaration of these signatures would look as follows.

```
sig Product {
  images: set Image,
  ①catalog: one Catalog①,
  ①category: one Category①
}
```

```
①sig Category {
  inside: one Catalog
}①
```

Fact `Thumbnails` also needs to be adapted with the introduction of this feature, so that products are retrieved indirectly from the categories of the catalog. To fetch the products inside a category `c`, instead of expression `catalog.c` we should use `category.inside.c`. As such, we could adapt this fact by annotating the initial constraint with ➊ and adding the new variant annotated with ①, as follows.

```
fact Thumbnails {
  ➊all c:Catalog | c.thumbnails in (catalog.c).images➊
  ①all c:Catalog | c.thumbnails in (category.inside.c).images①
}
```

Notice that we could alternatively have used finer grained annotations, to pinpoint more precisely where the changes need to be made. For operators with a clear neutral element, their branches can also be annotated by features in Colorful Alloy, so we could have specified the above fact as follows.

```
fact Thumbnails {
  all c:Catalog | c.thumbnails in (➊catalog.c➊ + ①category.inside.c①).images
}
```

As for our example scenario, we might want to add a extra restriction to variants where categories are present, for example forcing all products to have different categories, by annotating the respective expression with ① (Fig. 48, l. 34).

Having specified the changes introduced by feature ①, we can specify the adaptions needed for the other features. Colorful Alloy allows two signatures or fields with the same identifier to be declared, as long as they have disjoint feature annotations. For example, the multiplicity of field `category` should change when feature ③ is selected, to allow a product to belong to multiple categories. To do so, in Fig. 48, two alternative declarations of this field are present which, depending on whether ③ is selected, assign exactly one (l. 11) or multiple (l. 12) categories to a product. Also, depending on whether ② is selected or not, the signature `Category` should declare a different field `inside`: without hierarchical categories, each category is inside exactly one catalog (l. 24), while with hierarchical categories each category is inside one catalog or another category, determined by the union `Catalog+Category` (l. 27). Of course, we could also have opted to have a single declaration for `Category` with the two alternative field declarations inside, as we did for `Product` and field `category`, but opted for this formulation to illustrate the possibility

of declaring multiple versions of the same signature. Hierarchical categories require an additional fact `Acyclic` (l. 29–31) that forbids cyclic `inside` fields by calculating its transitive closure. Also, fact `Thumbnails` must be adapted when hierarchical categories are introduced (l. 20), so that all categories directly or indirectly contained in a catalog are considered: this can be done with a very precise annotation that replaces `inside` by `^inside` when feature ② is selected.

Notice that the run command on Fig. 48 has no feature scope imposed, meaning that the scenario can be run for any of the 5 valid variants (although with a slightly different behavior for variants with feature ① selected). To verify the correctness of the design for hierarchical categories, an assertion `AllCatalogued` can be specified (l. 38–40) to check whether every product is contained inside at least one catalog. The formula inside this assertion only makes sense when feature ② is selected. Note that due to the restrictions on the `FeatureModel` fact, feature ① will also be present, so also annotating this formula with ① will not change its presence condition. The feature scope of the respective check command (l. 41) is restricted to consider only the two relevant variants with ① and ② selected (namely, with or without feature ③, corresponding to multiple categories).

4.4 Type Checking Rules

The grammar of the language restricts which elements can be annotated, but additional type checking rules must be employed to guarantee consistent and analyzable colorful models. The Alloy type inference rules presented in Section 2.2.3 aim at discovering irrelevant expressions. These typically represent specification errors, but actually do not affect the semantics or prevent a model from being analyzed. In fact, the Analyzer allows the user to opt to consider these type errors as mere warnings, and proceed with the analysis even when they occur. There are however some type errors that cannot be treated as warnings and that are caught by a simpler type checking mechanism, not described in Section 2.2.3, namely arity errors, that can occur, for example, when a user tries to compute the union of two relational expressions of different arity. The type checking system we describe in this section aims at catching such kind of errors. In particular, for completeness, we also included arity type checking in the system.

The Colorful Alloy type checking rules aim mainly at detecting three kinds of coloring issues. First, calls to identifiers (for example, signature or predicate names) must occur in a color context that guarantees its existence. For instance, signature `Category` of our running example, which is declared only when feature ① is selected, cannot be called in a context where ① is not guaranteed to be selected. For example, `fact { some Category }` should raise an error, since such paragraph would be included

$\mathsf{decls}(c, \mathsf{p}_1, \ldots, \mathsf{p}_i) = \mathsf{decls}(c, \mathsf{p}_1) \cup \ldots \cup \mathsf{decls}(c, \mathsf{p}_i)$

$\mathsf{decls}(c,$ Ⓒ p Ⓒ $) = \mathsf{decls}(c \cup \{$Ⓒ$\}, \mathsf{p})$

$\mathsf{decls}(c, \textbf{module}\ \mathsf{n}\ [\ \mathsf{n}_1, \ldots, \mathsf{n}_k\ ]) = \mathsf{n}_1 \mapsto (\emptyset, 1) \cup \ldots \cup \mathsf{n}_k \mapsto (\emptyset, 1)$

$\mathsf{decls}(c, \textbf{open}\ \mathsf{n}\ [\ \mathsf{n}_1, \ldots, \mathsf{n}_k\ ]) = \mathsf{decls}(c, \mathsf{p}_1, \ldots, \mathsf{p}_i)$, where $\mathsf{p}_1, \ldots, \mathsf{p}_i$ are the paragraphs of n

$\mathsf{decls}(c, [\textbf{abstract}]\ [\mathsf{m}]\ \textbf{sig}\ \mathsf{n}\ [\textbf{extends}\ \mathsf{n}_1]\ \{\ \mathsf{d}_1, \ldots, \mathsf{d}_i\ \}\ [\{\ \mathsf{e}\ \}]) =$
$\quad \mathsf{n} \mapsto (c, 1) \cup \mathsf{decls}(c, \mathsf{d}_1) \cup \ldots \cup \mathsf{decls}(c, \mathsf{d}_i)$

$\mathsf{decls}(c, [\mathsf{m}]\ \textbf{sig}\ \mathsf{n}\ \textbf{in}\ \mathsf{n}_1 + \ldots + \mathsf{n}_k\ \{\ \mathsf{d}_1, \ldots, \mathsf{d}_i\ \}\ [\{\ \mathsf{e}\ \}]) =$
$\quad \mathsf{n} \mapsto (c, 1) \cup \mathsf{decls}(c, \mathsf{d}_1) \cup \ldots \cup \mathsf{decls}(c, \mathsf{d}_i)$

$\mathsf{decls}(c,$ Ⓒ d Ⓒ $) = \mathsf{decls}(c \cup \{$Ⓒ$\}, \mathsf{d})$

$\mathsf{decls}(c, \mathsf{n} : \mathsf{e}) = \mathsf{n} \mapsto (c, \mathsf{arity}(\mathsf{e}))$

$\mathsf{decls}(c, \textbf{fact}\ \{\ \mathsf{e}\ \}) = \emptyset$

$\mathsf{decls}(c, \textbf{pred}\ \mathsf{n}\ [\ \mathsf{d}_1, \ldots, \mathsf{d}_i\ ]\ \{\ \mathsf{e}\ \}) = \mathsf{n} \mapsto (c, i)$

$\mathsf{decls}(c, \textbf{fun}\ \mathsf{n}\ [\ \mathsf{d}_1, \ldots, \mathsf{d}_i\ ] : \mathsf{e}_1\ \{\ \mathsf{e}_2\ \}) = \mathsf{n} \mapsto (c, i + \mathsf{arity}(\mathsf{e}_1))$

$\mathsf{decls}(c, \textbf{run}\ \{\ \mathsf{e}\ \}\ [\textbf{with}\ [\textbf{exactly}]\ c_0]\ [\textbf{for}\ \mathsf{s}]) = \emptyset$

$\mathsf{decls}(c, \textbf{check}\ \{\ \mathsf{e}\ \}\ [\textbf{with}\ [\textbf{exactly}]\ c_0]\ [\textbf{for}\ \mathsf{s}]) = \emptyset$

Figure 49: Collecting a typing context from declarations.

in all possible variants and `Category` is not declared in all of them. This applies to calls in expressions, the class hierarchy (the parent signature must exist in every variant that the children do), and calls to predicates/asserts in run/check commands. Second, when multiple declarations of the same identifier exist, they must be guaranteed to have disjoint color contexts, so that only one such declaration exists in a particular variant. Thus, both declarations of `Category` are valid since contexts ①❷ and ①② refer to distinct variants, but it would not be the case if, for instance, one of them was just annotated with ①. Third, the nesting of negative and positive annotations of the same feature should be forbidden, since this conjunction of conditions is necessarily inconsistent (i.e., under ①❶e❶①, e will never exist). This also applies to feature scopes of commands, where the presence and absence of a feature would allow no variant.

The context of the type rules will be a mapping Γ from identifiers to the color annotation (a set of positive and negative feature marks) and arity of their declaration, and a color annotation c under which it is being evaluated. Since the same entity can be declared multiple times, as long as their color annotations are disjoint, Γ is actually a relation that associates declared identifiers with a set of pairs with color annotations and arities. We denote a singleton mapping as $\mathsf{n} \mapsto (c, k)$, for an identifier n, color annotation c, and arity k. The union of mappings can be done with $\cup$, while $+\!\!+$ denotes overriding. This context can be collected from a Colorful model using function decls defined in Fig. 49. For simplicity, this definition considers

only a kernel of paragraphs, namely commands that call predicates or assertions are not considered. Also, function arity used here is an oversimplification, since calculating the arity of an expression requires prior knowledge of the arity of other declared signatures and fields. In this definition (and in the typing rules presented later), p denotes a paragraph (or an open statement), d a declaration, n an identifier, m a multiplicity keyword, e an arbitrary expression (either a formula or a relational expression), $\boldsymbol{c}$ a color annotation (a set of feature marks), and i, j, k arities.

A context Γ collected by decls is valid iff the color annotations of all declarations with the same identifiers are disjoint and agree on arity, that is

$$\forall \mathsf{n}_0 \mapsto (\boldsymbol{c}_0, i), \mathsf{n}_1 \mapsto (\boldsymbol{c}_1, j) \cdot n_0 = n_1 \rightarrow \lceil \boldsymbol{c}_0 \rceil \cap \lceil \boldsymbol{c}_1 \rceil = \emptyset \wedge i = j$$

where $\lceil \boldsymbol{c} \rceil$ is a function that computes the set of all concrete variants that are valid according to $\boldsymbol{c}$, taking into consideration only the features that used in the specification. For example, in the e-commerce example ⌈{①,❷}⌉ = {{①,❷,③},{①,❷,❸}}, and the context collected from the model declarations would be

{Product ↦ ({},1), images ↦ ({},2), catalog ↦ ({❶},2),
category ↦ ({①,❸},2), category ↦ ({①,③},2), ...}

The typing rules for paragraphs are presented in Fig. 50. The fact that a paragraph p is well typed in context Γ with color annotation $\boldsymbol{c}$ is denoted by $\Gamma, \boldsymbol{c} \vdash \mathsf{p}$. The fact that a colorful model comprised by paragraphs $\mathsf{p}_1 \dots \mathsf{p}_i$ is well-typed is denoted by $\vdash \mathsf{p}_1 \dots \mathsf{p}_i$. Imported modules must also be well-typed according to the same rules. The typing rules for paragraphs are mainly responsible for aggregating color annotations as we traverse the model, to be later used when type checking expressions, as well as detecting the third kind of issue described above, color annotations with the same feature occurring positively and negatively. For an arbitrary feature mark ⓒ, ¬ⓒ converts between the positive and negative version. In Colorful Alloy commands are not annotated, their color context being instead defined by the feature scope. This scope is used to type check the expression inside the command. When the feature scope is exact, the respective color annotation must be expanded with the negation of all marks not present in it, which is done by function $\lfloor \boldsymbol{c} \rfloor$. For example, in the e-commerce example ⌊{①,②}⌋ = {①,②,❸}.

The typing rules for expressions is presented in Fig. 51 for a kernel of operators. The fact that an expression e of arity k is well-typed is denoted by $\Gamma, \boldsymbol{c} \vdash_k \mathsf{e}$ (arity 0 denotes formulas). Again most rules just aggregate feature marks as the expression is traversed, detecting contradictory marks and checking

$$\frac{\Gamma = \mathsf{decls}(\emptyset, \mathtt{p}_1, \ldots, \mathtt{p}_i) \quad \Gamma,\emptyset \vdash \mathtt{p}_1 \quad \ldots \quad \Gamma,\emptyset \vdash \mathtt{p}_i}{\vdash \mathtt{p}_1\ \ldots\ \mathtt{p}_i}$$

$$\frac{\Gamma, \boldsymbol{c} \cup \{\textcircled{c}\} \vdash \mathtt{p} \quad \vdash \textcircled{c}, \boldsymbol{c}}{\Gamma, \boldsymbol{c} \vdash \textcircled{c}\ \mathtt{p}\ \textcircled{c}} \qquad \frac{\vdash \boldsymbol{c} \quad \neg\textcircled{c} \notin \boldsymbol{c}}{\vdash \textcircled{c}, \boldsymbol{c}}$$

$$\frac{}{\Gamma,\emptyset \vdash \mathbf{module}\ \mathtt{n}\ [\ \mathtt{n}_1,\ldots,\mathtt{n}_i\ ]} \quad \frac{\Gamma,\boldsymbol{c} \vdash_1 \mathtt{n}_1 \quad \ldots \quad \Gamma,\boldsymbol{c} \vdash_1 \mathtt{n}_i}{\Gamma,\boldsymbol{c} \vdash \mathbf{open}\ \mathtt{n}\ [\ \mathtt{n}_1,\ldots,\mathtt{n}_i\ ]}$$

$$\frac{\Gamma,\boldsymbol{c} \vdash_{k_1} \mathtt{d}_1 \quad \ldots \quad \Gamma,\boldsymbol{c} \vdash_{k_i} \mathtt{d}_i \quad \Gamma,\boldsymbol{c} \vdash_0 \mathtt{e} \quad \Gamma,\boldsymbol{c} \vdash_1 \mathtt{n}_2 \quad k_1 \ldots k_i > 0}{\Gamma,\boldsymbol{c} \vdash [\mathbf{abstract}]\,[\mathtt{m}]\ \mathbf{sig}\ \mathtt{n}_1\ [\mathbf{extends}\ \mathtt{n}_2]\ \{\ \mathtt{d}_1,\ldots,\mathtt{d}_i\ \}\ [\{\ \mathtt{e}\ \}]}$$

$$\frac{\Gamma,\boldsymbol{c} \vdash_{k_1} \mathtt{d}_1 \quad \ldots \quad \Gamma,\boldsymbol{c} \vdash_{k_i} \mathtt{d}_i \quad \Gamma,\boldsymbol{c} \vdash_0 \mathtt{e} \quad \Gamma,\boldsymbol{c} \vdash_1 \mathtt{n}_1 \quad \ldots \quad \Gamma,\boldsymbol{c} \vdash_1 \mathtt{n}_j \quad k_1 \ldots k_i > 0}{\Gamma,\boldsymbol{c} \vdash [\mathtt{m}]\ \mathbf{sig}\ \mathtt{n}\ \mathbf{in}\ \mathtt{n}_1 + \ldots + \mathtt{n}_j\ \{\ \mathtt{d}_1,\ldots,\mathtt{d}_i\ \}\ [\{\ \mathtt{e}\ \}]}$$

$$\frac{\Gamma, \boldsymbol{c} \cup \{\textcircled{c}\} \vdash_k \mathtt{d} \quad \vdash \textcircled{c}, \boldsymbol{c}}{\Gamma, \boldsymbol{c} \vdash_k \textcircled{c}\ \mathtt{d}\ \textcircled{c}} \qquad \frac{\Gamma,\boldsymbol{c} \vdash_k \mathtt{e} \quad k > 0}{\Gamma,\boldsymbol{c} \vdash_k \mathtt{n}\ :\ \mathtt{e}}$$

$$\frac{\Gamma,\boldsymbol{c} \vdash_0 \mathtt{e}}{\Gamma,\boldsymbol{c} \vdash \mathbf{fact}\ \{\ \mathtt{e}\ \}}$$

$$\frac{\Gamma,\boldsymbol{c} \vdash_1 \mathtt{d}_1 \quad \ldots \quad \Gamma,\boldsymbol{c} \vdash_1 \mathtt{d}_i \quad \Gamma,\boldsymbol{c} \vdash_0 \mathtt{e}}{\Gamma,\boldsymbol{c} \vdash \mathbf{pred}\ \mathtt{n}\ [\ \mathtt{d}_1,\ldots,\mathtt{d}_i\ ]\ \{\ \mathtt{e}\ \}}$$

$$\frac{\Gamma,\boldsymbol{c} \vdash_1 \mathtt{d}_1 \quad \ldots \quad \Gamma,\boldsymbol{c} \vdash_1 \mathtt{d}_i \quad \Gamma,\boldsymbol{c} \vdash_k \mathtt{e}_1 \quad \Gamma,\boldsymbol{c} \vdash_k \mathtt{e}_2 \quad k > 0}{\Gamma,\boldsymbol{c} \vdash \mathbf{fun}\ \mathtt{n}\ [\ \mathtt{d}_1,\ldots,\mathtt{d}_i\ ]\ :\ \mathtt{e}_1\ \{\ \mathtt{e}_2\ \}}$$

$$\frac{\Gamma,\boldsymbol{c} \vdash_0 \mathtt{e} \quad \vdash \boldsymbol{c}}{\Gamma,\emptyset \vdash \mathbf{run}\ \{\ \mathtt{e}\ \}\ [\mathbf{with}\ c]\ [\mathbf{for}\ \mathtt{s}]} \qquad \frac{\Gamma,\lfloor \boldsymbol{c} \rfloor \vdash_0 \mathtt{e} \quad \vdash \boldsymbol{c}}{\Gamma,\emptyset \vdash \mathbf{run}\ \{\ \mathtt{e}\ \}\ [\mathbf{with\ exactly}\ c]\ [\mathbf{for}\ \mathtt{s}]}$$

$$\frac{\Gamma,\boldsymbol{c} \vdash_0 \mathtt{e} \quad \vdash \boldsymbol{c}}{\Gamma,\emptyset \vdash \mathbf{check}\ \{\ \mathtt{e}\ \}\ [\mathbf{with}\ c]\ [\mathbf{for}\ \mathtt{s}]} \qquad \frac{\Gamma,\lfloor \boldsymbol{c} \rfloor \vdash_0 \mathtt{e} \quad \vdash \boldsymbol{c}}{\Gamma,\emptyset \vdash \mathbf{check}\ \{\ \mathtt{e}\ \}\ [\mathbf{with\ exactly}\ c]\ [\mathbf{for}\ \mathtt{s}]}$$

Figure 50: Type rules for kernel paragraphs.

$$\frac{}{\Gamma, \boldsymbol{c} \vdash_1 \texttt{none}} \qquad \frac{}{\Gamma, \boldsymbol{c} \vdash_1 \texttt{univ}} \qquad \frac{}{\Gamma, \boldsymbol{c} \vdash_2 \texttt{iden}} \qquad \frac{\forall \boldsymbol{c}_0 \in \lceil \boldsymbol{c} \rceil \cdot \exists \mathsf{n} \mapsto (\boldsymbol{c}_1, k) \in \Gamma \cdot \boldsymbol{c}_1 \subseteq \boldsymbol{c}_0}{\Gamma, \boldsymbol{c} \vdash_k \mathsf{n}}$$

$$\frac{\Gamma, \boldsymbol{c} \vdash_2 \mathsf{e}}{\Gamma, \boldsymbol{c} \vdash_2 \hat{}\,\mathsf{e}} \qquad \frac{\Gamma, \boldsymbol{c} \vdash_2 \mathsf{e}}{\Gamma, \boldsymbol{c} \vdash_2 \sim\mathsf{e}} \qquad \frac{\Gamma, \boldsymbol{c} \vdash_0 \mathsf{e}}{\Gamma, \boldsymbol{c} \vdash_0 \texttt{not e}} \qquad \frac{\Gamma, \boldsymbol{c} \vdash_0 \mathsf{e}_1 \quad \Gamma, \boldsymbol{c} \vdash_0 \mathsf{e}_2}{\Gamma, \boldsymbol{c} \vdash_0 \mathsf{e}_1 \texttt{ and } \mathsf{e}_2}$$

$$\frac{\Gamma, \boldsymbol{c} \vdash_k \mathsf{e}_1 \quad \Gamma, \boldsymbol{c} \vdash_k \mathsf{e}_2 \quad k > 0}{\Gamma, \boldsymbol{c} \vdash_0 \mathsf{e}_1 \texttt{ in } \mathsf{e}_2} \qquad \frac{\Gamma, \boldsymbol{c} \vdash_k \mathsf{e}_1 \quad \Gamma, \boldsymbol{c} \vdash_k \mathsf{e}_2 \quad k > 0 \quad \square \in \{\&, +, -\}}{\Gamma, \boldsymbol{c} \vdash_k \mathsf{e}_1 \square \mathsf{e}_2}$$

$$\frac{\Gamma, \boldsymbol{c} \vdash_i \mathsf{e}_1 \quad \Gamma, \boldsymbol{c} \vdash_j \mathsf{e}_2 \quad k = i + j - 2 \quad i, j, k > 0}{\Gamma, \boldsymbol{c} \vdash_k \mathsf{e}_1 \,.\, \mathsf{e}_2}$$

$$\frac{\Gamma, \boldsymbol{c} \vdash_i \mathsf{e}_1 \quad \Gamma, \boldsymbol{c} \vdash_j \mathsf{e}_2 \quad k = i + j \quad i, j > 0}{\Gamma, \boldsymbol{c} \vdash_k \mathsf{e}_1 \rightarrow \mathsf{e}_2}$$

$$\frac{\Gamma, \boldsymbol{c} \vdash_1 \mathsf{e}_1 \quad \Gamma + \mathsf{n} \mapsto (\emptyset, 1), \boldsymbol{c} \vdash_0 \mathsf{e}_2}{\Gamma, \boldsymbol{c} \vdash_0 \texttt{all n : } \mathsf{e}_1 \texttt{ | } \mathsf{e}_2} \qquad \frac{\Gamma, \boldsymbol{c} \cup \{\textcircled{c}\} \vdash_k \mathsf{e} \quad \vdash \textcircled{c}, \boldsymbol{c}}{\Gamma, \boldsymbol{c} \vdash_k \textcircled{c}\, \mathsf{e}\, \textcircled{c}}$$

Figure 51: Type rules for kernel expressions.

the arity. The most interesting rule is the one for identifiers, in the upper right corner. A reference to an identifier n is well typed in a context Γ and color annotation $\boldsymbol{c}$ if in all possible variants $\boldsymbol{c}_0 \in \lceil \boldsymbol{c} \rceil$ that identifier is declared. For example, in the e-commerce example, expression ①**some** `Category`① is well-typed because either ①②`Category`②① or ①❷`Category`❷① is declared in all variants where ① is selected. Unfortunately, this rule is too restrictive when the FM actually restricts the possible set of variants. For example, expression ②**some** `Category`② would not be considered well-typed because in some variants where ② is selected `Category` is not declared, for example, in variant {❶,②,③}. However, this variant is not allowed by the FM of this example, since fact `FeatureModel` requires ① to be selected when ② is also selected. Thus, the rule implemented in the Colorful Alloy typing system is a bit more general, so that less spurious errors are returned. In particular, the model is first scanned to detect FM constraints with the shape $\boldsymbol{c}$ **some none** $\boldsymbol{c}$, where $\boldsymbol{c}$ denotes a feature combination that will be considered forbidden. From these, the set F containing all possible valid variants in the model is computed (five, in the case of our running example), and the typing rule for identifiers is adapted as follows.

$$\frac{\forall \boldsymbol{c}_0 \in \lceil \boldsymbol{c} \rceil \cap F \cdot \exists \mathsf{n} \mapsto (\boldsymbol{c}_1, k) \in \Gamma \cdot \boldsymbol{c}_1 \subseteq \boldsymbol{c}_0}{\Gamma, \boldsymbol{c} \vdash_k \mathsf{n}}$$

$$\langle \mathtt{p}_1 \;\cdots\; \mathtt{p}_i \rangle_{\boldsymbol{c}} \equiv \langle \mathtt{p}_1 \rangle_{\boldsymbol{c}} \;\cdots\; \langle \mathtt{p}_i \rangle_{\boldsymbol{c}}$$

$$\langle \textcircled{c}\ \mathtt{p}\ \textcircled{c} \rangle_{\boldsymbol{c}} \equiv \begin{cases} \langle \mathtt{p} \rangle_{\boldsymbol{c}} & \text{if } \textcircled{c} \in \lfloor \boldsymbol{c} \rfloor \\ \epsilon & \text{otherwise} \end{cases}$$

$$\langle \mathbf{module}\ \mathtt{n}\ [\ \mathtt{n}_1,\ldots,\mathtt{n}_i\ ] \rangle_{\boldsymbol{c}} \equiv \mathbf{module}\ \mathtt{n}\ [\ \mathtt{n}_1,\ldots,\mathtt{n}_i\ ]$$

$$\langle \mathbf{open}\ \mathtt{n}\ [\ \mathtt{n}_1,\ldots,\mathtt{n}_i\ ] \rangle_{\boldsymbol{c}} \equiv \mathbf{open}\ \mathtt{n}\ [\ \mathtt{n}_1,\ldots,\mathtt{n}_i\ ]$$

$$\langle [\mathbf{abstract}]\,[\mathtt{m}]\ \mathbf{sig}\ \mathtt{n}_1\ [\mathbf{extends}\ \mathtt{n}_2]\ \{\ \mathtt{d}_1,\ldots,\mathtt{d}_i\ \}[\{\ \mathtt{e}\ \}] \rangle_{\boldsymbol{c}} \equiv$$
$$\qquad [\mathbf{abstract}]\,[\mathtt{m}]\ \mathbf{sig}\ \mathtt{n}_1\ [\mathbf{extends}\ \mathtt{n}_2]\ \{\langle \mathtt{d}_1 \rangle_{\boldsymbol{c}},\ldots,\langle \mathtt{d}_i \rangle_{\boldsymbol{c}}\ \}[\{\langle e \rangle_{\boldsymbol{c}}\}]$$

$$\langle [\mathtt{m}]\ \mathbf{sig}\ \mathtt{n}\ \mathbf{in}\ \mathtt{n}_1 + \ldots + \mathtt{n}_j\ \{\ \mathtt{d}_1,\ldots,\mathtt{d}_i\ \}[\{\ \mathtt{e}\ \}] \rangle_{\boldsymbol{c}} \equiv$$
$$\qquad [\mathtt{m}]\ \mathbf{sig}\ \mathtt{n}\ \mathbf{in}\ \mathtt{n}_1 + \ldots + \mathtt{n}_j\ \{\langle \mathtt{d}_1 \rangle_{\boldsymbol{c}},\ldots,\langle \mathtt{d}_i \rangle_{\boldsymbol{c}}\ \}[\{\langle e \rangle_{\boldsymbol{c}}\}]$$

$$\langle \textcircled{c}\ \mathtt{d}\ \textcircled{c} \rangle_{\boldsymbol{c}} \equiv \begin{cases} \langle \mathtt{d} \rangle_{\boldsymbol{c}} & \text{if } \textcircled{c} \in \lfloor \boldsymbol{c} \rfloor \\ \epsilon & \text{otherwise} \end{cases}$$

$$\langle \mathtt{n}\ :\ \mathtt{e} \rangle_{\boldsymbol{c}} \equiv \mathtt{n}\ :\langle \mathtt{e} \rangle_{\boldsymbol{c}}$$

$$\langle \mathbf{fact}\ \{\ \mathtt{e}\ \} \rangle_{\boldsymbol{c}} \equiv \mathbf{fact}\ \{\ \langle \mathtt{e} \rangle_{\boldsymbol{c}}\}$$

$$\langle \mathbf{pred}\ \mathtt{n}\ [\ \mathtt{d}_1,\ldots,\mathtt{d}_i\ ]\ \{\ \mathtt{e}\ \} \rangle_{\boldsymbol{c}} \equiv \mathbf{pred}\ \mathtt{n}\ [\ \langle \mathtt{d}_1 \rangle_{\boldsymbol{c}},\ldots,\langle \mathtt{d}_i \rangle_{\boldsymbol{c}}\ ]\ \{\langle \mathtt{e} \rangle_{\boldsymbol{c}}\}$$

$$\langle \mathbf{fun}\ \mathtt{n}\ [\ \mathtt{d}_1,\ldots,\mathtt{d}_i\ ]\ :\ \mathtt{e}_1\ \{\ \mathtt{e}_2\ \} \rangle_{\boldsymbol{c}} \equiv \mathbf{fun}\ \mathtt{n}\ [\ \langle \mathtt{d}_1 \rangle_{\boldsymbol{c}},\ldots,\langle \mathtt{d}_i \rangle_{\boldsymbol{c}}\ ]\ :\langle \mathtt{e}_1 \rangle_{\boldsymbol{c}}\ \{\langle \mathtt{e}_2 \rangle_{\boldsymbol{c}}\}$$

$$\langle \mathbf{run}\ \{\ \mathtt{e}\ \}\ [\mathbf{for}\ \mathtt{s}] \rangle_{\boldsymbol{c}} \equiv \mathbf{run}\ \{\ \langle \mathtt{e} \rangle_{\boldsymbol{c}}\ \}[\mathbf{for}\ \mathtt{s}]$$

$$\langle \mathbf{run}\ \{\ \mathtt{e}\ \}\ \mathbf{with}\ \boldsymbol{c}_0\ [\mathbf{for}\ \mathtt{s}] \rangle_{\boldsymbol{c}} \equiv \begin{cases} \mathbf{run}\ \{\ \langle \mathtt{e} \rangle_{\boldsymbol{c}}\ \}\ [\mathbf{for}\ \mathtt{s}] & \text{if } \boldsymbol{c}_0 \subseteq \lfloor \boldsymbol{c} \rfloor \\ \epsilon & \text{otherwise} \end{cases}$$

$$\langle \mathbf{run}\ \{\ \mathtt{e}\ \}\ \mathbf{with\ exactly}\ \boldsymbol{c}_0\ [\mathbf{for}\ \mathtt{s}] \rangle_{\boldsymbol{c}} \equiv \begin{cases} \mathbf{run}\ \{\ \langle \mathtt{e} \rangle_{\boldsymbol{c}}\ \}\ [\mathbf{for}\ \mathtt{s}] & \text{if } \lfloor \boldsymbol{c}_0 \rfloor = \lfloor \boldsymbol{c} \rfloor \\ \epsilon & \text{otherwise} \end{cases}$$

$$\langle \mathbf{check}\ \{\ \mathtt{e}\ \}\ [\mathbf{for}\ \mathtt{s}] \rangle_{\boldsymbol{c}} \equiv \mathbf{check}\ \{\ \langle \mathtt{e} \rangle_{\boldsymbol{c}}\ \}[\mathbf{for}\ \mathtt{s}]$$

$$\langle \mathbf{check}\ \{\ \mathtt{e}\ \}\ \mathbf{with}\ \boldsymbol{c}_0\ [\mathbf{for}\ \mathtt{s}] \rangle_{\boldsymbol{c}} \equiv \begin{cases} \mathbf{check}\ \{\ \langle \mathtt{e} \rangle_{\boldsymbol{c}}\ \}\ [\mathbf{for}\ \mathtt{s}] & \text{if } \boldsymbol{c}_0 \subseteq \lfloor \boldsymbol{c} \rfloor \\ \epsilon & \text{otherwise} \end{cases}$$

$$\langle \mathbf{check}\ \{\ \mathtt{e}\ \}\ \mathbf{with\ exactly}\ \boldsymbol{c}_0\ [\mathbf{for}\ \mathtt{s}] \rangle_{\boldsymbol{c}} \equiv \begin{cases} \mathbf{check}\ \{\ \langle \mathtt{e} \rangle_{\boldsymbol{c}}\ \}\ [\mathbf{for}\ \mathtt{s}] & \text{if } \lfloor \boldsymbol{c}_0 \rfloor = \lfloor \boldsymbol{c} \rfloor \\ \epsilon & \text{otherwise} \end{cases}$$

Figure 52: Paragraph projection.

4.5 Semantics

The semantics of Colorful Alloy can be defined in terms of a projection operator, that extracts from a colorful model a plain Alloy model representing a concrete variant. Assuming that a colorful model is well-typed according to the rules presented in the previous section, and that all unique feature marks ⓒ that are used in it have been collected in a color annotation $\boldsymbol{c}$ during that process (i.e., the features relevant for the specified family of models), an instance M is valid in a colorful model m iff there exists a variant $\boldsymbol{c}_0 \subseteq \boldsymbol{c}$ such that M is valid in $\langle m \rangle_{\boldsymbol{c}_0}$, the projection of m for variant $\boldsymbol{c}_0$, according to the plain Alloy semantics defined in Section 2.2.2.

$$\langle \textbf{none} \rangle_c \equiv \textbf{none}$$
$$\langle \textbf{univ} \rangle_c \equiv \textbf{univ}$$
$$\langle \textbf{iden} \rangle_c \equiv \textbf{iden}$$
$$\langle \mathtt{n} \rangle_c \equiv \mathtt{n}$$
$$\langle \Box\ \mathtt{e} \rangle_c \equiv \Box\ \langle \mathtt{e} \rangle_c$$
$$\langle \mathtt{e}_1 \Box \mathtt{e}_2 \rangle_c \equiv \langle \mathtt{e}_1 \rangle_c \Box \langle \mathtt{e}_2 \rangle_c \qquad \text{if } \Box \notin \{+, \&, \textbf{or}, \textbf{and}\}$$
$$\langle c_1 \mathtt{e}_1 c_1 \Box c_2 \mathtt{e}_2 c_2 \rangle_c \equiv \begin{cases} \langle \mathtt{e}_1 \rangle_c \Box \langle \mathtt{e}_2 \rangle_c & \text{if } c_1 \subseteq \lfloor c \rfloor \text{ and } c_2 \subseteq \lfloor c \rfloor \\ \langle \mathtt{e}_1 \rangle_c & \text{if } c_1 \subseteq \lfloor c \rfloor \text{ and } c_2 \not\subseteq \lfloor c \rfloor \\ \langle \mathtt{e}_2 \rangle_c & \text{if } c_1 \not\subseteq \lfloor c \rfloor \text{ and } c_2 \subseteq \lfloor c \rfloor \\ \text{neutral}(\Box, \text{arity}(\mathtt{e}_1)) & \text{otherwise} \end{cases} \quad \text{if } \Box \in \{+, \&, \textbf{or}, \textbf{and}\}$$
$$\langle \textbf{all}\ \mathtt{n} : \mathtt{e}_1 \mid \mathtt{e}_2 \rangle_c \equiv \textbf{all}\ \mathtt{n} : \langle \mathtt{e}_1 \rangle_c \mid \langle \mathtt{e}_2 \rangle_c$$

Figure 53: Expression projection.

The projection of a model to a concrete variant c is defined in Fig. 52 and is rather straight-forward: basically it projects away paragraphs and declarations not relevant in that variant, namely those enclosed in an annotation ⓒ that is not selected in c. The projection of expressions is also straight-forward and is defined in Fig. 53. Recall that only direct sub-expressions of binary operators with a neutral element can be annotated in Colorful Alloy. In this case, if both sub-expressions are to be projected out, the parent expression will be replaced by the respective neutral element, defined as follows.

$$\text{neutral}(+, a) = \underbrace{\textbf{none} \rightarrow \ldots \rightarrow \textbf{none}}_{a}$$
$$\text{neutral}(\&, a) = \underbrace{\textbf{univ} \rightarrow \ldots \rightarrow \textbf{univ}}_{a}$$
$$\text{neutral}(\textbf{or}, a) = \textbf{some none}$$
$$\text{neutral}(\textbf{and}, a) = \textbf{no none}$$

An example of this process can be seen in the projected Alloy model of the e-commerce colorful specification for the variant where only ① and ② are selected (the variant with hierarchical categories), that is presented in Fig. 54. In the colorful specification the `FeatureModel` fact consists of a conjunction of two annotated formulas, that encode the restrictions of the respective FM. None of these formulas should be included in the projected model, because their annotations are not part of the variant. As such, the (implicit) conjunction inside `FeatureModel` is replaced by **no none**, the respective neutral element, and will thus have no effect on the analysis. If instead we asked for the projection to the variant where only

```
fact FeatureModel {
  no none
}

sig Product{
  images: one catalog,
  category: one Category
}
sig Image {}
sig Catalog {
  thumbnails: set Image
}
fact Thumbnails {
  all c : catalog | c.thumbnails in (category.^inside.c.image)
}
sig Category{
  inside: one Catalog + Category
}
fact Acyclic {
  all c: category | c not in c.^inside
}

pred Scenario {
  some Product.images and all c: Category | lone category.c
}
run Scenario for 10

assert AllCataloged {
  all p:Product | some (p.category.^inside & Catalog)
}
check AllCataloged for 10
```

Figure 54: E-commerce example projection to variant ①,②.

② is selected, this fact would contain formula **`some none`**, which would imply this variant would have no valid instances, as expected.

4.6 Analysis

Analysis of Colorful Alloy models is achieved through translation into regular Alloy. We defined two alternative ways to do this: *i*) through the generation and analysis, for every feature combination, of a projected version of the model; *ii*) through the generation of a single 'amalgamated' Alloy model through feature lifting, which encompasses all the alternative behaviors of the model family. Concerning the former *iterative* analysis, since Colorful Alloy does not natively support FMs, all the $2^{\#c_0}$ projected models must be generated and analyzed (although the process can be stopped once one of those models is found to be satisfiable). However, the codification of FMs proposed in Section 4.2, actually renders invalid variants

trivially unsatisfiable and instantaneously discharged: the projection of the model for such variants will end up with a fact enforcing **`some none`**, which is detected during the translation into SAT before the solving process is even launched.

Figures 55 and 56 present the translation of the colorful model into the amalgamated version for paragraphs and expressions, respectively. It is assumed that the colorful model is well-typed at this stage, and that all unique colorful marks c_0 that occur in it have been collected during that process (i.e., the features relevant for this family of models). For a model $\mathsf{p}_1\ \dots\ \mathsf{p}_i$, the translation $\langle\langle \mathsf{p}_1\ \dots\ \mathsf{p}_i \rangle\rangle_{c_0}$ starts by introducing an abstract signature `Feature`, that is extended exactly by singleton signatures that represent each of the relevant features in c_0. Signature `Variant`, a sub-set of `Feature`, represents particular feature combinations under consideration.[3] Variability points are introduced by tests over the valuation of `Variant`, here abstracted by a macro isPresent(c) that for a given color context c generates a formula that tests whether `Variant` holds for c, defined as:

$$\text{isPresent}(c) = \begin{cases} \underbrace{\texttt{Fp1 in Variant and} \dots \texttt{and Fpi in Variant}}_{\text{(p1)},\dots,\text{(pi)} \in c} \\ \texttt{and} \\ \underbrace{\texttt{Fn1 not in Variant and} \dots \texttt{and Fnj not in Variant}}_{\text{(n1)},\dots,\text{(ni)} \in c} \end{cases}$$

When c is empty this degenerates to true, so the element will be present in every variant, as expected.

To control the existence of structural elements (signatures and fields), additional facts force them to be empty whenever the presence conditions do not hold. Even though these elements are always declared, the colorful type checking rules guarantee that they are not referenced in invalid variants. For this mechanism to hold, multiplicities of declarations are relaxed in the general case, and additional facts only enforce them in variants where the elements are present. In the kernel language, only sub-expressions of binary operators may be associated with features (blocks of formulas have been converted into binary conjunctions). If a branch is present in a variant, its original expression is returned, otherwise the neutral element of the operation is. Lastly, commands are also expanded so that behavior depends on their feature scope, so that only relevant variants are considered in the analyses.

Besides variability points, there is an additional issue that must be addressed during this translation, the potential existence of non-unique identifiers which cannot exist in plain Alloy after expansion. In (global) declarations, each identifier `n` is renamed with a label that uniquely identifies the color context c of that

3 To avoid collisions with the identifiers of the colorful model, the implemented translation actually uses obfuscated identifiers for these signatures.

declaration, abstracted by procedure $\mathsf{id}(\mathsf{n}, \boldsymbol{c})$. Since the color context of non-unique identifiers is always disjoint, such labels will be unique. Then, whenever an identifier is called, it is expanded into the union of all available renamed identifiers that are available in that context. For that purpose, we rely on the context Γ to find all alternative declarations of the same identifier `m`. Being disjoint, in each variant at most one of those references will be non-empty. This approach has, however, some limitations, namely when the call to an element cannot be replaced by a union of identifiers. In particular, this occurs in signature extensions, import statements, and assertions and predicates invoked in non-block commands. In these situations, to use amalagamated analysis a stronger type rule must be imposed over identifiers, so that exactly one exists in the given context, namely:

$$\frac{\exists^1 \mathsf{n} \mapsto (\boldsymbol{c}_1, k) \in \Gamma \cdot \forall \boldsymbol{c}_0 \in \lceil \boldsymbol{c} \rceil \cap F \cdot \boldsymbol{c}_1 \subseteq \boldsymbol{c}_0}{\Gamma, \boldsymbol{c} \vdash_k \mathsf{n}}$$

Figures 57 and 58 presents the automatically created amalgamated Alloy model for the e-commerce colorful model, with some minor simplifications for readability (such as merging facts enforcing multiplicities, not translating presence conditions for empty annotations). Note also that, although not present in the kernel, blocks are simply treated as conjunctions. Notice how duplicated identifiers in the colorful model are renamed (here, function $\mathsf{id}(\mathsf{n}, \boldsymbol{c})$ is implemented as `n_f1...fi` for ⓕ1, …, ⓕi $\in \boldsymbol{c}$), and then calls to that identifier are translated into the union of all renamed declarations. Since they are necessarily disjoint due to the colorful type rules, only one of them will exist in each variant. Notice also how the assert (not part of the kernel) is converted to a predicate so that the a precondition corresponding to the feature scope can be introduced in the check command.

To show that the amalgamated version preserves the semantics of the colorful model, we must show that for each concrete variant, the amalgamated model will behave as the corresponding projected model. Note that an instance M for an amalgamated model belongs to a particular variant $\boldsymbol{c}_0$ (the valuation of `Variant`), and all signatures/fields that do not belong to that variant are necessarily empty. Let $M|_{\boldsymbol{c}_0}$ represent instance M with those signatures/fields and feature signatures removed, and with $\mathsf{id}(\mathsf{n}, \boldsymbol{c})$ identifiers renamed to their original name (possible since only one unique a identifier may exist per variant, and thus per instance). Then, for a colorful model m, for each variant $\boldsymbol{c}_0 \subseteq \boldsymbol{c}$, an instance M must be valid in $\langle\!\langle m \rangle\!\rangle_{\boldsymbol{c}}$ **and fact** { isPresent($\lfloor \boldsymbol{c}_0 \rfloor$) } iff $M|_{\boldsymbol{c}_0}$ is valid in $\langle m \rangle_{\boldsymbol{c}_0}$, both according to the plain Alloy semantics. It is easy to see that is the case. For signatures and fields, if they are absent they are assigned multiplicity **no**, if present they are kept the same except for their multiplicity, which is moved to a fact (a semantics-preserving refactoring, see Chapter 5). For expressions, if a branch is absent it is

$$\langle\!\langle \mathtt{p}_1 \;\dots\; \mathtt{p}_i \rangle\!\rangle_{\{\textcircled{k},\dots,\textcircled{l}\}} \equiv \begin{array}{l} \textbf{abstract sig}\ \mathtt{Feature}\ \{\} \\ \textbf{one sig}\ \mathtt{Fk},\ \dots,\ \mathtt{Fl}\ \textbf{extends}\ \mathtt{Feature}\ \{\} \\ \textbf{sig}\ \mathtt{Variant}\ \textbf{in}\ \mathtt{Feature}\ \{\} \\ \langle\!\langle \mathtt{p}_1 \rangle\!\rangle_\emptyset \;\dots\; \langle\!\langle \mathtt{p}_i \rangle\!\rangle_\emptyset \end{array}$$

$$\langle\!\langle \textcircled{c}\ \mathtt{p}\ \textcircled{c} \rangle\!\rangle_c \equiv \langle\!\langle \mathtt{p} \rangle\!\rangle_{c \cup \{\textcircled{c}\}}$$

$$\langle\!\langle \textbf{module}\ \mathtt{n}\ [\ \mathtt{n}_1,\dots,\mathtt{n}_i\ ] \rangle\!\rangle_c \equiv \textbf{module}\ \mathtt{n}\ [\langle\!\langle \mathtt{n}_1 \rangle\!\rangle_c,\dots,\langle\!\langle \mathtt{n}_i \rangle\!\rangle_c]$$

$$\langle\!\langle \textbf{open}\ \mathtt{n}\ [\ \mathtt{n}_1,\dots,\mathtt{n}_i\ ] \rangle\!\rangle_c \equiv \textbf{open}\ \mathtt{n}\ [\ \mathtt{n}_1,\dots,\mathtt{n}_i\ ]$$

$$\langle\!\langle [\textbf{abstract}]\,[\mathtt{m}]\ \textbf{sig}\ \mathtt{n}\,[\textbf{extends}\ \mathtt{n}_1]\ \{\ \mathtt{d}_1,\dots,\mathtt{d}_i\ \}\ \{\ \mathtt{e}\ \} \rangle\!\rangle_c \equiv$$

$$\begin{array}{l} [\textbf{abstract}]\ \textbf{sig}\ \mathrm{id}(\mathtt{n},c)\ [\textbf{extends}\ \langle\!\langle \mathtt{n}_1 \rangle\!\rangle_c]\ \{\ \langle\!\langle \mathtt{d}_1 \rangle\!\rangle_c,\dots,\langle\!\langle \mathtt{d}_i \rangle\!\rangle_c\ \}\ \{\ \langle\!\langle \mathtt{e} \rangle\!\rangle_c\ \} \\ \textbf{fact}\ \{\ \textbf{not}\ \mathrm{isPresent}(c)\ \textbf{implies}\ \textbf{no}\ \mathrm{id}(\mathtt{n},c)\ \} \\ [\textbf{fact}\ \{\ \mathrm{isPresent}(c)\ \textbf{implies}\ \mathtt{m}\ \mathrm{id}(\mathtt{n},c)\ \}] \\ \textbf{fact}\ \{\ \mathrm{trans}(\mathtt{d}_1,c)\ \dots\ \mathrm{trans}(\mathtt{d}_i)\ \} \end{array}$$

where

$$\mathrm{trans}(\textcircled{c}\ \mathtt{d}\ \textcircled{c}, c_1) = \mathrm{trans}(\mathtt{d}, c_1 \cup \{\textcircled{c}\})$$

$$\mathrm{trans}(\mathtt{v}\ :\ \mathtt{e}, c_1) = \mathrm{isPresent}(c_1)\ \textbf{implies}\ \mathrm{id}(\mathtt{v},c_1)\ \textbf{in}\ \langle\!\langle \mathtt{n} \rightarrow \mathtt{e} \rangle\!\rangle_c\ \textbf{else}\ \textbf{no}\ \mathrm{id}(\mathtt{v},c_1)$$

$$\langle\!\langle [\mathtt{m}]\ \textbf{sig}\ \mathtt{n}\ \textbf{in}\ \mathtt{n}_1{+}\dots{+}\mathtt{n}_j\ \{\ \mathtt{d}_1,\dots,\mathtt{d}_i\ \}\ \{\ \mathtt{e}\ \} \rangle\!\rangle_c \equiv$$

$$\begin{array}{l} \textbf{sig}\ \mathrm{id}(\mathtt{n},c)\ \textbf{in}\,\langle\!\langle \mathtt{n}_1 \rangle\!\rangle_c{+}\dots{+}\langle\!\langle \mathtt{n}_j \rangle\!\rangle_c\ \{\ \langle\!\langle \mathtt{d}_1 \rangle\!\rangle_c,\dots,\langle\!\langle \mathtt{d}_i \rangle\!\rangle_c\ \}\ \{\ \langle\!\langle \mathtt{e} \rangle\!\rangle_c\ \} \\ \textbf{fact}\ \{\ \textbf{not}\ \mathrm{isPresent}(c)\ \textbf{implies}\ \textbf{no}\ \mathrm{id}(\mathtt{n},c)\ \} \\ [\textbf{fact}\ \{\ \mathrm{isPresent}(c)\ \textbf{implies}\ \mathtt{m}\ \mathrm{id}(\mathtt{n},c)\ \}] \\ \textbf{fact}\ \{\ \mathrm{trans}(\mathtt{d}_1,c)\ \dots\ \mathrm{trans}(\mathtt{d}_i)\ \} \end{array}$$

where

$$\mathrm{trans}(\textcircled{c}\ \mathtt{d}\ \textcircled{c}, c_1) = \mathrm{trans}(\mathtt{d}, c_1 \cup \{\textcircled{c}\})$$

$$\mathrm{trans}(\mathtt{v}\ :\ \mathtt{e}, c_1) = \mathrm{isPresent}(c_1)\ \textbf{implies}\ \mathrm{id}(\mathtt{v},c_1)\ \textbf{in}\ \langle\!\langle \mathtt{n} \rightarrow \mathtt{e} \rangle\!\rangle_c\ \textbf{else}\ \textbf{no}\ \mathrm{id}(\mathtt{v},c_1)$$

$$\langle\!\langle \textcircled{c}\ \mathtt{d}\ \textcircled{c} \rangle\!\rangle_c \equiv \langle\!\langle \mathtt{d} \rangle\!\rangle_{c \cup \{\textcircled{c}\}}$$

$$\langle\!\langle \mathtt{v}\ :\ \mathtt{m}\ \mathtt{e} \rangle\!\rangle_c \equiv \mathrm{id}(\mathtt{v},c)\ :\ \textbf{set}\ \langle\!\langle \mathtt{e} \rangle\!\rangle_c$$

$$\langle\!\langle \mathtt{v}\ :\ \mathtt{e}_1\ \mathtt{m}_1 \rightarrow \mathtt{m}_2\ \mathtt{e}_2 \rangle\!\rangle_c \equiv \mathrm{id}(\mathtt{v},c)\ :\ \langle\!\langle \mathtt{e}_1 \rangle\!\rangle_c\ \textbf{set} \rightarrow \textbf{set}\ \langle\!\langle \mathtt{e}_2 \rangle\!\rangle_c$$

$$\langle\!\langle \textbf{fact}\ \{\ \mathtt{e}\ \} \rangle\!\rangle_c \equiv \textbf{fact}\ \{\ \langle\!\langle \mathtt{e} \rangle\!\rangle_c\ \}$$

$$\langle\!\langle \textbf{pred}\ \mathtt{n}\ [\ \mathtt{d}_1,\dots,\mathtt{d}_i\ ]\ \{\ \mathtt{e}\ \} \rangle\!\rangle_c \equiv \textbf{pred}\ \mathrm{id}(\mathtt{n},c)\ [\ \langle\!\langle \mathtt{d}_1 \rangle\!\rangle_c,\dots,\langle\!\langle \mathtt{d}_i \rangle\!\rangle_c\ ]\ \{\ \langle\!\langle \mathtt{e} \rangle\!\rangle_c\ \}$$

$$\langle\!\langle \textbf{fun}\ \mathtt{n}\ [\ \mathtt{d}_1,\dots,\mathtt{d}_i\ ]\ :\ \mathtt{e}_1\ \{\ \mathtt{e}_2\ \} \rangle\!\rangle_c \equiv \textbf{fun}\ \mathrm{id}(\mathtt{n},c)\ [\ \langle\!\langle \mathtt{d}_1 \rangle\!\rangle_c,\dots,\langle\!\langle \mathtt{d}_i \rangle\!\rangle_c\ ]\ :\ \langle\!\langle \mathtt{e}_1 \rangle\!\rangle_c\ \{\ \langle\!\langle \mathtt{e}_2 \rangle\!\rangle_c\ \}$$

$$\langle\!\langle \textbf{run}\ \{\ \mathtt{e}\ \}\ \textbf{with}\ c\ \textbf{for}\ \mathtt{s} \rangle\!\rangle_\emptyset \equiv \textbf{run}\ \{\ \mathrm{isPresent}(c)\ \textbf{and}\ \langle\!\langle \mathtt{e} \rangle\!\rangle_c\ \}\ \textbf{for}\ \mathtt{s}$$

$$\langle\!\langle \textbf{run}\ \{\ \mathtt{e}\ \}\ \textbf{with exactly}\ c\ \textbf{for}\ \mathtt{s} \rangle\!\rangle_\emptyset \equiv \textbf{run}\ \{\ \mathrm{isPresent}(\lfloor c \rfloor)\ \textbf{and}\ \langle\!\langle \mathtt{e} \rangle\!\rangle_c\ \}\ \textbf{for}\ \mathtt{s}$$

$$\langle\!\langle \textbf{check}\ \{\ \mathtt{e}\ \}\ \textbf{with}\ c\ \textbf{for}\ \mathtt{s} \rangle\!\rangle_\emptyset \equiv \textbf{check}\ \{\ \mathrm{isPresent}(c)\ \textbf{implies}\ \langle\!\langle \mathtt{e} \rangle\!\rangle_c\ \}\ \textbf{for}\ \mathtt{s}$$

$$\langle\!\langle \textbf{check}\ \{\ \mathtt{e}\ \}\ \textbf{with exactly}\ c\ \textbf{for}\ \mathtt{s} \rangle\!\rangle_\emptyset \equiv \textbf{check}\ \{\ \mathrm{isPresent}(\lfloor c \rfloor)\ \textbf{implies}\ \langle\!\langle \mathtt{e} \rangle\!\rangle_c\ \}\ \textbf{for}\ \mathtt{s}$$

Figure 55: Paragraph translation into the amalgamated model with variability.

$$
\begin{aligned}
&\langle\!\langle \textbf{none} \rangle\!\rangle_c \equiv \textbf{none}\\
&\langle\!\langle \textbf{univ} \rangle\!\rangle_c \equiv \textbf{univ}\\
&\langle\!\langle \textbf{iden} \rangle\!\rangle_c \equiv \textbf{iden}\\
&\langle\!\langle \mathsf{n} \rangle\!\rangle_c \equiv \underbrace{\mathsf{id}(\mathsf{n}, c_1) + \ldots + \mathsf{id}(\mathsf{n}, c_i)}_{c_j \in \Gamma(\mathsf{n}) \cdot c_j \subseteq c}\\
&\langle\!\langle \Box \mathsf{e} \rangle\!\rangle_c \equiv \Box \langle\!\langle \mathsf{e} \rangle\!\rangle_c\\
&\langle\!\langle \mathsf{e}_1 \Box \mathsf{e}_2 \rangle\!\rangle_c \equiv \begin{cases} \langle\!\langle \mathsf{e}_1 \rangle\!\rangle_c \Box \langle\!\langle \mathsf{e}_2 \rangle\!\rangle_c & \text{if } \Box \notin \{+, \&, \textbf{or}, \textbf{and}\} \\ \mathsf{trans}(\mathsf{e}_1) \Box \mathsf{trans}(\mathsf{e}_2) & \text{otherwise} \end{cases}\\
&\quad \text{where}\\
&\qquad \mathsf{trans}(\mathsf{e}) \equiv \mathsf{isPresent}(\lfloor c \rfloor) \textbf{ implies } \langle\!\langle \mathsf{e} \rangle\!\rangle_c \textbf{ else } \mathsf{neutral}(\Box, \mathsf{arity}(\mathsf{e}))\\
&\langle\!\langle \textbf{all } \mathsf{v} : \mathsf{e}_1 \mid \mathsf{e}_2 \rangle\!\rangle_c \equiv \textbf{all } \mathsf{v} : \langle\!\langle \mathsf{e}_1 \rangle\!\rangle_c \mid \langle\!\langle \mathsf{e}_2 \rangle\!\rangle_c\\
&\langle\!\langle \textcircled{c}\ \mathsf{e}\ \textcircled{c} \rangle\!\rangle_c \equiv \langle\!\langle \mathsf{e} \rangle\!\rangle_{c \cup \{\textcircled{c}\}}
\end{aligned}
$$

Figure 56: Expression translation into the amalgamated model with variability.

converted to the neutral element (the same as removing it), otherwise it is left unchanged. Lastly, run and check commands are always unsatisfiable when absent in a variant, the former never producing an instance (conjuncted with false) and the latter never producing a counter-example (pre-conditioned with false).

```
abstract sig Feature {}
one sig F1,F2,F3 extends Feature{}
sig Variant in Feature {}

fact FeatureModel {
  (F1 not in Variant and F2 in Variant) implies some none else no none
  (F1 not in Variant and F3 in Variant) implies some none else no none
}

sig Product {
  images: set Image,
  catalog: set Catalog,
  category_0: set Category_0+Category_1,
  category_1: set Category_0+Category_1
}
fact {
  (F1 not in Variant) implies
     catalog in Product → one Catalog else no catalog
  (F1 in Variant and F3 not in Variant) implies
    category_0 in Product → one (Category_0+Category_1) else no category_0
  (F1 in Variant and F3 in Variant) implies
    category_1 in Product → some (Category_0+Category_1) else no category_1
}
sig Image {}
sig Catalog {
  thumbnails: set Image
}
fact Thumbnails {
  (F1 not in Variant) implies
    (all c:Catalog | c.thumbnails in (catalog.c).images) else no none
  (F1 in Variant) implies
    (all c:catalog | c.thumbnails in ((category_0+category_1).((F2 not in Variant
    implies inside else none→none)+(F2 in Variant implies ^inside else none→none))
    .c.images)) else no none
}
sig Category_0 {
  inside: one Catalog
}
sig Category_1 {
  inside: one Catalog+Category_0+Category_1
}
fact {
  not (F1 in Variant and F2 not in Variant) implies
    no Category_0
  not (F1 in Variant and F2 in Variant) implies
    no Category_1
}
fact Acyclic {
  (F1 in Variant and F2 in Variant) implies
    (all c:Category_1+Category_1 | c not in c.^inside) else no none
}
```

Figure 57: Amalgamated translation of the e-commerce model from Fig. 48 (except commands).

```
pred Scenario {
  some Product.images
  (F1 in Variant) implies
     (all c:Category_0+Category_1 | lone (category_0+category_1).c)
    else no none
}
run { Scenario } for 10

pred AllCataloged {
  (F2 in Variant) implies
     all p:Product | some (p.(category_0+category_1).^inside & Catalog)
}
check { (F1 in Variant and F2 in Variant) implies
  AllCataloged } for 10
```

Figure 58: Amalgamated translation of the e-commerce model from Fig. 48 (commands).

5

MERGING CLONED ALLOY MODELS WITH COLORFUL REFACTORINGS

In this chapter we propose a refactoring approach for migrating a collection of plain Alloy models, possibly developed with the clone-and-own approach, into a single Colorful Alloy model. As mentioned in Chapter 3, modern software systems are often highly-configurable, effectively encoding a family of software products, and feature-oriented software development is the most successful approach proposed to support the development of such systems. In such a family-based development, maintenance is an important challenge, and refactoring is one of the key activities used to improve the internal quality during software evolution. A refactoring is a kind of transformation that changes the structure of the source code while preserving its external behavior.

Variability annotations introduce an additional complexity layer in the development process, making refactoring crucial to keep software maintainable and understandable. However, classical refactoring is not well-suited for feature-oriented development, since it does not take into account the behavior of a SPL as a whole. SPL refactoring must consider all variants, in particular, the set of possible variants and the behavior of each variant must be preserved (Schulze et al., 2012). Furthermore, typical code refactoring laws do not suffice for formal software design, since most refactoring laws are typically too coarse-grained, focusing on constructs such as entire functions or classes. Given the high level of abstraction on which design is performed, to improve the quality of a specification one might need also fine-grained refactoring laws, for example, affecting just part of an assertion.

As discussed in Section 3.4, one of the standard ways to implement multiple variants in a SPL is through the *clone-and-own* approach. However, as the cost to maintain the clones and synchronize changes in replicas increases, developers may benefit from migrating (by merging) such variants into a single feature-oriented artifact. In fact, even when feature-oriented development has been fully adopted, it is useful in practice to switch between the two perspectives (Rubin et al., 2015). Fully-automated approaches for clone merging (e.g., the technique proposed by Rubin and Chechik (2012), discussed in Section 3.4) assume a quantifiable measure of quality to guide the process, but such measure is not easy to define

when the goal is to merge code, and even less so when the goal is to merge formal abstract specifications. Alternative techniques where the user still somehow controls the merging process have been proposed. In particular, Fenske et al. (2017) have proposed a migration strategy where variant preserving refactorings were defined to support the user in performing step-wise, semi-automated transformations, a technique that inspired our own proposal for migrating plain Alloy clones into a single Colorful Alloy model.

In this chapter we start by reviewing the proposal of (Fenske et al., 2017) in Section 5.1. Then, in Section 5.2 we propose a catalog of variability-aware refactoring laws for Colorful Alloy, covering all model constructs – from structural declarations to axioms and assertions – and granularity levels – from whole paragraphs to formulas and expressions. Then, in Section 5.3 we show how these refactorings can be used to migrate a set of legacy Alloy clones into a colorful SPL using an approach similar to the one of (Fenske et al., 2017). Fine-grained refactoring is particularly relevant in this context: design in Alloy is done at high levels of abstraction and variants often introduce precise changes – refactoring only at the paragraph level, likewise the refactorings proposed by Gheyi (2007) (and presented in Section 2.3), would lead to unnecessary code replication and a difficulty to identify variability points. Finally, in Section 5.4 we also introduce an automatic merging strategy that composes together several refactorings, and that can be used to simplify the task of the user when migrating clones to a managed SPL design.

5.1 Migrating Code Clones into an SPL with Refactoring

Fenske et al. (2017) propose a technique for migrating cloned product variants into an SPL that relies on variant-preserving refactorings. This technique was applied to Java code clones, with the resulting SPL encoded using *Feature-Oriented Programming* (FOP) (Prehofer, 1997), a composition-based technique where all code artifacts that implement each feature are modularized in different units, called *feature modules*. The composition technique used to derive a specific variant implementation allows feature modules being composed later to refine definitions introduced previously, namely by adding fields or methods, by adapting the behaviour of methods by calling the original implementation, or by completely overriding the previous definition.

The process of the variant-preserving migration is shown in Fig. 59. Suppose we have n product variants $p_1, p_2, \cdots, p_n$, possibly sharing a large amount of code, to be migrated into a single FOP SPL. In the first step, the process creates a trivial initial SPL whose FM contains n features $p_1, p_2, \cdots, p_n$ grouped in a single alternative relation, each of which representing a product. Furthermore, the source code of each variant becomes a separate feature module associated with the respective feature. This FOP

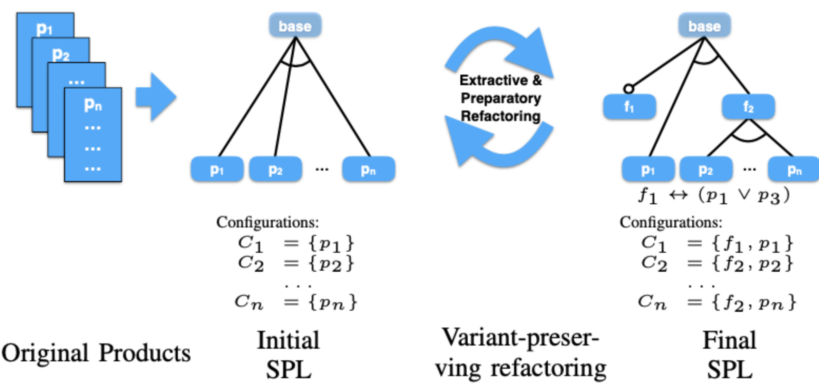


Figure 59: Clone migration process of (Fenske et al., 2017)

implementation together with the FM entails n different configurations $C_1, C_2, \dots, C_n$ each with exactly one of the features, and we can easily recover the original variant p_i through configuration C_i. This first step does not yield any improvement in reuse, but rather provides the basis for the subsequent iterative variant-preserving refactoring process.

Then a step-wise refinement process is applied and repeated as often as needed in order to improve the quality of the SPL implementation. This process uses a combination of *extractive* and *preparatory* variant-preserving refactorings: the goal of the former is to move identical code fragments (including different elements, such as fields or methods) introduced by a set of features into a common one and higher up in the feature hierarchy, while the latter focus on aligning implementations with Type-2 clones (for example, identical methods with different names) so that extractive refactorings can be applied. *Variant-preserving refactorings* (Schulze et al., 2012) extend the notion of behavior-preservation to SPLs, ensuring that all potential products within an SPL remain compilable and keep their previous behavior, and that all configurations that were valid before the refactoring must remain valid afterwards. This behavior-preservation notion allows the introduction of new variants as long as they do not affect the behavior of the existing ones.

The key extractive variant-preserving refactoring used in this work is Pull Up to Common Feature, an extension to FOP of the Pull Up refactoring first introduced by Fowler (2018) for object-oriented code. The goal of the Pull Up refactoring is to move equal definitions of class members (for example, methods or fields) to a common super-class, thus reducing code clones. The goal of the Pull Up to Common Feature is similar, but the idea is to move equal definitions of class members up in the feature hierarchy (instead of up in the class hierarchy) from a set F_s of *source features* $\{f_{s_1}, \dots, f_{s_n}\}$ to a single *target feature* f_t. There are several versions of this refactoring, but we will detail here the version applied to methods, the Pull Up Method to Common Feature. Notice that this refactoring should be applied only to Type-1 clones, after the

common code has been consolidated in the initial SPL. Given a method m and a class c the goal is to create c in f_t (if it does not exist already) and define m in that class, deleting all definitions of m in the source features $f_s \in F_s$. To be variant-preserving, this refactoring has several preconditions:

1. Each source feature $f_s \in F_s$ contains the class c with method m.
2. All fields and methods referred by m must be defined in the target feature f_t or in one of the features implied by it.
3. f_t must be a *concrete feature*, that is, one containing code (instead of an *abstract feature* that is just used in the FM to organize other features, for example the parent feature of an alternative relation between concrete features).
4. If c already exists in f_t it must not declare m.
5. Any valid configuration that contains an $f_s \in F_s$ must contain f_t and vice-versa (the latter prevents m from shadowing other definitions in features composed before f_t in undesired configurations containing f_t but not one of the source features).
6. No valid configuration containing an $f_s \in F_s$ may contain a $f_d \in F - F_s$ that contains a class c with m and that is composed after f_t and before f_s (to prevent m from being overwritten or refined before f_s being composed).

The most important issue with this refactoring is identifying the appropriate target feature f_t, namely one that obeys the above preconditions. If all source features have a common parent, then that parent feature might be used as the target feature (if it satisfies all the preconditions), otherwise a new target feature can be created. In the latter case, first an appropriate parent feature f_p is selected (required to satisfy only some of the pre-conditions but not all of them, for example it might not be concrete or not imply one of the source features) and then f_t is created as a concrete optional child of f_p. To fulfill precondition 5 a cross-tree constraint is created stating that f_t is selected if and only if one of the source features is selected. For example, in Fig. 59 feature f_1 could be the target feature created by the Pull Up Method to Common Feature to refactor a cloned method in features p_1 and p_3, hence the new cross tree constraint $f_1 \leftrightarrow (p_1 \vee p_3)$. The corresponding set of configurations also changes according to the new added features during the refactoring. For instance, configuration C_1 is now changed to include both feature f_1 and feature p_1.

As mentioned above, the Pull Up to Common Feature refactoring only applies to identical codes fragments from source code (Type-1 clones). For dealing with Type-2 clones, for example methods with the same

implementation but distinct names, the preparatory Rename refactoring should used first, to enable the later application of Pull Up to Common Feature. One important thing in this refactoring step is to update all the references to the new name. Otherwise, compilation errors would shown up in variants that contain the feature of the element that was renamed.

This clone migration process was implemented in Eclipse on top of other tools, namely FeatureIDE (Thüm et al., 2014), the Eclipse refactoring framework, and copy and paste detectors. The latter were used to implement Type-1 and Type-2 clone detection, to suggest potential applications of the Pull Up to Common Feature and Rename refactorings. The authors also implemented an automated migration strategy that just applies repeatedly the former refactoring to all detected Type-1 clones. They evaluated the migration process by applying it to five variants of Android games developed with Java with a high-percentage of cloned code (in total around 69% of the LOCs were identified as inter-feature code clones), by measuring the reduction in the total LOCs. Since the technique only considers Type-1 and Type-2 clones (meaning that methods with even very small differences in the implementation could not be refactored), the automatic migration strategy only achieved a reduction of 4.2% of the total LOCs, with an additional 15.8% reduction after applying preparatory Rename refactorings. In the final FM, 15 additional target features were created to group common code.

5.2 Refactoring Rules for Colorful Alloy

This section proposes a catalog of variant-preserving refactorings for Colorful Alloy, complementing the non-variability-aware ones previously proposed for standard Alloy (Gheyi, 2007), which were presented in Section 2.3. Although the main use of this catalogue will be in our clone migration technique, which is inspired by the technique described in the previous section, they can also be used to promote the quality and maintenance of Colorful Alloy models, while preserving the set of variants and their individual behavior. Although the general idea of our technique for clone migration is similar to the work of (Fenske et al., 2017), in the sense that we first migrate all clones into an initial trivially correct SPL which is then improved by variant-preserving refactorings, there are some key differences that justify our particular choice of refactorings, apart from the obvious ones that our technique applies to formal design instead of code, and targets an annotative approach, instead of a compositional one:

- We will not consider alignment issues at all, assuming that names of entities that model the same concept in different variants have been previously made equal. The name alignment problem is mostly orthogonal to the migration problem: some automation can be provided to help with that,

for example using clone detection tools, but mostly it is up to the user to decide when two entities are to be considered the same. For example, in the process proposed by Fenske et al. (2017), the automatic migration strategy did not include the Rename refactoring, which was left to the user to be applied.

- However, we intend to address not only Type-1 code clones, but also Type-3 ones, namely to enable the merging of two entities (predicates, facts, etc) whose specification might differ. To that end we propose several refactorings that work on the formula and expression level, a feature that distinguishes our approach from other works, allowing finer variability annotations, as is common in feature-oriented design, particularly in the clone-and-own context where changes may be small. This also distinguishes our approach from that of (Gheyi, 2007), where the proposed Alloy refactorings are coarse-grained.

- Unlike the technique of (Fenske et al., 2017), our refactorings do not create new features nor change the FM (as was the case with the Pull Up to Common Feature refactoring). We believe domain analysis, where the variability is modelled (with FMs), should be an orthogonal (and prior) task to clone migration, and that the developer should be ultimately responsible for deciding which features exist and how are they related. As such, we will assume that prior to migration the final FM has been defined and each clone has already been labeled with the configuration (set of features) it implements.

The refactoring laws for Colorful Alloy are presented in the form of an equation between two templates (with square brackets marking optional elements), following the style from (Gheyi, 2007), under the context of a particular FM F extracted from the model under analysis (as described in the previous chapter). Unlike in the refactoring laws of (Gheyi, 2007), our templates can be matched to part of a model (and not the all model), meaning that the refactoring is just to be applied locally to that part. As long as the preconditions are met and the left or right templates matched, the refactoring can be applied in either direction. Also, our notion of equivalence is different: we do not consider only a subset of relevant signatures and fields, but all that are declared in the model.

Throughout the section, $\textcircled{c}$ and $\textcircled{c}$ will denote a positive or negative annotation for feature c, while $\textcircled{c}$ will denote either a positive or negative annotation for c; moreover, $\textcircled{c}$ will denote a (possibly empty) sequence of positive or negative annotations[1]. Models are assumed to be type checked when the rules

1 Essentially $\textcircled{c}$ is just a different notation for $\mathbf{c}$, as used in the previous chapter, the only difference being that the annotations in the former have a particular order while the latter is an unordered set. We believe that this alternative notation improves the readability of the refactoring laws presented in this section.

are applied, so without loss of generality, in an expression ⓒeⓒ we assume that the features c in the closing annotations appear in the reverse order as those in the opening annotations, that there are no contradictory annotations, and that only supported elements of the AST are annotated. We use `ann` to refer to any model element amenable of being annotated (possibly itself already annotated), `exp` and `frm` for expressions and formulas, respectively, `n` for identifiers, `ds` for (possibly-annotated) relation declarations, and `scp` for scopes. We assume that using the extracted FM F (a set of configurations) we can answer simple questions about the FM, namely whether a particular set of features ⓐ entails a particular feature set ⓑ, denoted by $F \models$ ⓐ $\rightarrow$ ⓑ, which is defined as follows.

$$F \models \textcircled{a} \rightarrow \textcircled{b} \quad \text{iff} \quad \forall \textcircled{c} \in F \cdot \textcircled{a} \subseteq \textcircled{c} \rightarrow \textcircled{b} \subseteq \textcircled{c}$$

Annotation Laws The first set of rules we address manage the feature annotations which, as will be shown, are often useful to align annotations on elements to enable more advanced refactorings. They act essentially as the preparatory refactorings in (Fenske et al., 2017).

Law 11 (Annotation reordering)**.**

ⓐⓑannⓑⓐ	$=_F$	ⓑⓐannⓐⓑ

This basic rule arises from the commutativity property of conjunctions, and allows users to reorganize feature annotations. Therefore, users can rearrange the order of marked features on elements as needed. For example, the field `category` declaration inside signature `Product` (Fig. 48, l. 11), marked with features ①❸, could alternatively be declared as follows.

```
❸①category : one Category①❸
```

Law 12 (Redundant annotations)**.**

ⓐⓑannⓑⓐ	$=_F$	ⓐannⓐ

provided

($\leftrightarrow$) $F \models$ ⓐ $\rightarrow$ ⓑ and `ann` $\neq$ **`some none`**.

This rule relies on the FM to identify redundant annotations that can be removed or introduced. In order to not affect the implicitly specified FM (from which F is extracted) its application is forbidden for **`some none`** formulas. For instance, if the FM imposes ② → ① (as in our example), then whenever a ② annotation is present ① is spurious, and whenever ❶ is present ❷ is spurious. In an extreme case it

can also be used to remove duplicated annotations, since trivially ⓒ → ⓒ. Similar rules are defined to manage the annotations in the feature scope of commands. In the e-commerce example, using this refactoring, the declaration of signature Category for variant ①② (l. 26–28) could be simplified as follows.

```
②sig Category {
  inside: one Catalog+Category
}②
```

Signature Laws The next set of refactoring rules manage signatures with shared presence conditions, including their merging. In particular, most refactorings from (Gheyi, 2007), and others, to remove syntactic sugar constructs from signatures, must be adapted to the colorful context. Here we present a few as examples.

Law 13 (Remove signature multiplicity qualifier)**.**

ⓐ [abstract] mlt sig n [ext] {…} ⓐ	$=_F$	ⓐ [abstract] sig n [ext] {…} ⓐ ⓐ fact { mlt n } ⓐ

where

mlt ∈ {lone, one, some}.

This law removes multiplicity of signatures with an additional fact constraint. This rule is similar to that of normal Alloy, with the only difference being that the same feature expression as the original signature needs to be marked on the additional fact which indicates that the fact only applies when the original signature is selected. This rule is very useful as a preparatory refactoring in merging cloned variants, since signatures with the same identifier but different multiplicity often occur in systems developed with clone-and-own approaches. Using this law we can then remove these multiplicity constraints to enable further merging.

Law 14 (Remove abstract qualifier)**.**

ⓐabstract sig n [ext] {…}ⓐ ⓐⓑsig n_0 extends n {…}ⓑⓐ … ⓐⓒsig n_l extends n {…}ⓒⓐ	$=_F$	ⓐsig n [ext] {…}ⓐ ⓐⓑsig n_0 extends n {…}ⓑⓐ … ⓐⓒsig n_l extends n {…}ⓒⓐ ⓐfact { n=ⓑn_0ⓑ+…+ⓒn_lⓒ }ⓐ

provided

(↔) $l \geq 0$.

This law removes the **abstract** qualifier from signature declarations. The last line on the right-hand side of the law adds an additional **fact** that constrains the relationship between a parent signature and all its sub signatures. Law 13 and Law 14 remove syntactic sugar annotations from signatures, and are often used as a preparatory step to support the following merge signature refactorings. Note however, that while Law 13 is a direct adaptation of the rules for normal Alloy by considering an annotation context ⓐ, that is not the case for Law 14, where the possible distinct annotations of the children signatures must be taken into consideration.

Law 15 (Merge signature)**.**

```
ⓐⓑsig n [extends n0] {
  ds0,…, dsk }ⓑⓐ
ⓐ❶sig n [extends n0] {
  ds'0,…, ds'l }❶ⓐ
```

$=_F$

```
ⓐsig n [extends n0] {
  ⓑds0ⓑ,…,ⓑdskⓑ,
  ❶ds'0❶,…,❶ds'l❶ }ⓐ
```

Signatures cannot be freely merged independently of their annotations, since in Colorful Alloy they are not sufficiently expressive to represent the disjunction of presence conditions. Signatures with the same identifier can be merged if they partition a certain annotation context ⓐ on ⓑ, in which case the latter can be dropped (but pushed down to the respective field declarations). Due to the opposite ⓑ annotations the two signatures never coexist in a variant, and the merged signature will exist in exactly the same variants, those determined by ⓐ. Notice that these laws act on signatures without qualifiers. If qualifiers are compatible between the two signatures, they can be reintroduced after merging by applying the syntactic sugar laws in the opposite direction. Similar laws are defined for merging inclusion signatures (declared with **in** instead of **extends**).

Returning to the e-commerce example of Fig. 48, it could be argued that the declaration of two distinct `Category` signatures under ① depending on whether ② is also selected or not, is not ideal (l. 23–28). Since neither signature has other qualifiers, Law 15 can be applied directly from left to right to merge the two signature declarations, resulting in the following single declaration.

```
①sig Category {
  ❷inside: one Catalog❷,
  ②inside: one Catalog+Category②
}①
```

Notice that fields are left unmerged by this refactoring. These will be the target of a separate set of refactorings to be presented below.

Field Laws Much like signatures, fields must be stripped down of syntactic sugar constructs in order to be merged, which also amounts to adapting fields refactoring from (Gheyi, 2007), and others, to the colorful context.

Law 16 (Remove binary field multiplicity qualifier)**.**

```
ⓐ sig n {
  ⓑ n0: mlt exp ⓑ,
  … ,
  ds
} ⓐ
```

$=_F$

```
ⓐ sig n {
  ⓑ n0: set exp ⓑ,
  … ,
  ds
} ⓐ
ⓐⓑ fact {
  all x:n | mlt x.n0
} ⓑⓐ
```

where

mlt $\in$ {**lone**, **one**, **some**} and x is a fresh variable.

This law moves the multiplicity constraint of a binary field into a properly annotated fact. Similar laws are defined for higher-arity declarations. By omission **one** is the default multiplicity, which must also be refactored out (rule omitted).

Law 17 (Merge binary field)**.**

```
ⓐⓑ n: set exp1 ⓑⓐ,
ⓐ❶ n: set exp2 ❶ⓐ
```

$=_F$

```
ⓐ n: set ⓑ exp1 ⓑ+❶ exp2 ❶ⓐ
```

This law allows fields with the same identifier to be merged, even if they have different binding expressions, if, like in signatures, they partition an annotation context ⓐ. Again, similar rules are defined for other field arities.

Back to the e-commerce example, the duplicated field `inside` introduced by the merging of signature `Category` could be merged into a single field declaration, using Law 16 for removing multiplicities and then using Law 17 for merging, resulting in

```
①sig Category {
  inside: set ❷Catalog❷+②Catalog+Category②
}①
①❷fact{ all x:Category | one x.inside }❷①
①②fact{ all x:Category | one x.inside }②①
```

The laws for formulas and expressions, presented next, will further simplify this snippet and allow the reintroduction of the multiplicities back into the declarations. Nonetheless, in general the merging of fields may result in additional facts.

Paragraph Laws Other paragraphs of Colorful Alloy can be more easily merged since there are fewer syntactic sugar constructs for them.

Open statements can also be merged only when a feature partitions their annotation context.

Law 18 (Merge import)**.**

ⓐ ⓑ **open** n[n_1,…,n_k] [**as** n_0] ⓑ ⓐ ⓐ ❶b **open** n[n_1,…,n_k] [**as** n_0] ❶b ⓐ	$=_F$	ⓐ **open** n[n_1,…,n_k] [**as** n_0] ⓐ

Law 19 (Merge fact)**.**

ⓐ **fact** [n] { frm_1 } ⓐ ⓑ **fact** [n] { frm_2 } ⓑ	$=_F$	**fact** [n] { ⓐ frm_1 ⓐ **and** ⓑ frm_2 ⓑ }

Facts, unlike the other paragraphs, are not called from other elements but always included in the model, so they can be soundly merged for whatever feature annotations (including having no annotations at all). For example, the two additional facts introduced after removing the multiplicities of the two variants of field `inside` declaration can be merged as follows.

```
fact {
  ①②all x:Category | one x.inside②① and
  ①②all x:Category | one x.inside②①
}
```

For the same reason, annotations around facts can also be pushed inside.

Law 20 (Fact annotation)**.**

ⓐ **fact** [n] { frm } ⓐ	$=_F$	**fact** [n] { ⓐ frm ⓐ }

Similarly to signatures and fields, other referable paragraphs can be merged when a feature partitions the annotation context, as long as that annotation is pushed inside to the body expressions. For instance, two variants of a predicate with the same number of arguments can be merged if they partition the annotation ⓐ over ⓑ. The respective body expressions are annotated with ⓑ and ❶b and concatenated with the **and** operator, and the type of each argument is equal to the union of the two original ones.

Law 21 (Merge predicate)**.**

```
ⓐⓑ pred n [
   n_0:exp_0,
   … ,
   n_k:exp_k
]
{ frm_1 } ⓑⓐ
ⓐ🅑 pred n [
   n_0:exp'_0,
   … ,
   n_k:exp'_k
]
{ frm_2 } 🅑ⓐ
```

$=_F$

```
ⓐ pred n [
   n_0:ⓑ exp_0 ⓑ+🅑 exp'_0 🅑,
   … ,
   n_k:ⓑ exp_k ⓑ+🅑 exp'_k 🅑
]
{ ⓑ frm_1 ⓑ and 🅑 frm_2 🅑 } ⓐ
```

Function refactoring is similar: the types of arguments, the return type, and the body are all merged with the set union operator, after being properly annotated.

Law 22 (Merge function)**.**

```
ⓐⓑ fun n [
   n_0:exp_0,
   … ,
   n_k:exp_k
] : exp_{k+1}
{ exp_{k+2} } ⓑⓐ
ⓐ🅑 fun n [
   n_0:exp'_0,
   … ,
   n_k:exp'_k
] : exp'_{k+1}
{ exp'_{k+2} } 🅑ⓐ
```

$=_F$

```
ⓐ fun n [
   n_0:ⓑ exp_0 ⓑ+🅑 exp'_0 🅑,
   … ,
   n_k:ⓑ exp_k ⓑ+🅑 exp'_k 🅑
] : ⓑ exp_{k+1} ⓑ+🅑 exp'_{k+1} 🅑
{ ⓑ exp_{k+2} ⓑ+🅑 exp'_{k+2} 🅑 } ⓐ
```

Assertions can also be referred inside commands, so their merging rule is similar to that of predicates.

Law 23 (Merge assertion)**.**

```
(a)(b) assert n { frm_1 } (b)(a)
(a)(b) assert n { frm_2 } (b)(a)
```
$=_F$
```
(a) assert n {
  (b) frm_1 (b) and (b) frm_2 (b)
} (a)
```

Since these elements do not affect the model unless referred in other paragraphs, we can define refactoring laws to introduce new declarations. Often these are useful as preparatory steps to allow the subsequent merging of declarations. Here we exemplify with a rule for assertions.

Law 24 (Insert assertion)**.**

```
ϵ
```
$=_F$
```
(a)assert n [...] { frm }(a)
```

provided

(→) for any other assertion n annotated with (b), $F \nvDash$ (a) $\wedge$ (b);

(←) for any check command referring to n with feature scope (b), $F \nvDash$ (b) $\rightarrow$ (a).

Command Laws Commands are bounded by the feature scope rather than annotated. If two commands act on a partition of the variants, they can be merged into a command addressing their union. As an example, we show the laws for non-block commands.

Law 25 (Merge predicate run command)**.**

```
run n [for scp] with (a),(b)
run n [for scp] with (a),(b)
```
$=_F$
```
run n [for scp] with (a)
```

Law 26 (Merge assertion check command)**.**

```
check n [for scp] with (a),(b)
check n [for scp] with (a),(b)
```
$=_F$
```
check n [for scp] with (a)
```

Formula and Expression Laws Lastly, we provide refactoring laws for formulas and expressions. This distinguishes our approach from other works, allowing finer variability annotations, as is common in feature-oriented design, particularly in the clone-and-own context where changes may be small.

The first law allows the removal of an annotated neutral element on the right-hand side of a binary operator. Since the target operators are commutative, it is also possible to use it to remove an annotated neutral element in the left-hand side.

Law 27 (Remove neutral element)**.**

ann op ⓐneutral(op, arity(ann))ⓐ	$=_F$	ann

where

op $\in$ {+, &, **and**, **or**}.

When the annotations of the left- and the right-hand sides of a binary operator form a partition it is possible to replace the operator by its dual, since in each variant only one of the sides is considered.

Law 28 (Exchange operator)**.**

ⓐ ann_1 ⓐ op_1 🅐 ann_2 🅐	$=_F$	ⓐ ann_1 ⓐ op_2 🅐 ann_2 🅐

where

$\mathsf{op}_1 \in$ {+, &, **and**, **or**} and op_2 is the dual operator of op_1.

The following law arises from the distributive property of operators and can be applied to both annotated formulas and expressions.

Law 29 (Merge common expression)**.**

ⓐ ann_1 op_2 ann_2ⓐ op_1 🅐 ann_1 op_2 ann_3 🅐	$=_F$	ann_1 op_2 (ⓐ ann_2 ⓐop_1🅐 ann_3 🅐)

where

$\mathsf{op}_1 \in$ {+, &, **and**, **or**} and op_2 is the dual operator of op_1.

By combining it with the previous refactoring we can obtain several useful variants of this law. For example, ⓐ ann_1ⓐ op 🅐 ann_1 op ann_2🅐 can be refactored to ann_1 op 🅐ann_2🅐, by first introducing the neutral element of op in the left-hand side, then applying Law 29, and finally removing the annotated neutral element with Law 27. An extreme case is when we have ⓐ annⓐ op 🅐 ann🅐, which can be refactored into ann. Since the operators are commutative we can use this law to merge a common expression in the right-hand side of op_2.

For the same binary operators it is also possible to merge two expressions annotated with the same features, as long as there is a third expression that is not merged.

Law 30 (Merge different expressions)**.**

ⓐann_1ⓐ op ⓐann_2ⓐ op ann_3	$=_F$	ⓐann_1 op ann_2ⓐ op ann_3

where

op ∈ {+, &, **and**, **or**}.

The reason why this third expression is required is due to the semantics of the language. Expression ⓐann_1 op ann_2ⓐ will be replaced by the the neutral element of the enclosing operator if ⓐ is not selected. If that operator is different from op we will end up with a different expression then the one obtained in ⓐann_1ⓐ op ⓐann_2ⓐ when ⓐ is not selected, which is the neutral element of op. There is a special case when the third expression is not required, which is when we have a top-level conjunction of two expressions (for example in a fact). In that case we can merge because, when ⓐ is not selected, the all top-level expression it is just removed, which is equivalent to replacing it by the neutral element of conjunction.

The following laws allow the combination of inclusion tests over the same expression, for whatever annotations, and arise from the properties of intersection and union.

Law 31 (Merge left-side inclusion)**.**

ⓐ exp_1 **in** exp_2 ⓐ **and** ⓑ exp_1 **in** exp_3 ⓑ	$=_F$	exp_1 **in** ⓐ exp_2 ⓐ&ⓑ exp_3 ⓑ

Law 32 (Merge right-side inclusion)**.**

ⓐ exp_2 **in** exp_1 ⓐ **and** ⓑ exp_3 **in** exp_1 ⓑ	$=_F$	ⓐ exp_2 ⓐ+ⓑ exp_3 ⓑ **in** exp_1

Since an equality test can be refactored into the conjunction of two inclusion tests it also possible to use this law to merge some equality tests. In particular if the annotations form a partition it is possible to combine it with Law 28 to obtain the following law.

Law 33 (Merge equality)**.**

ⓐ exp_1 = exp_2 ⓐ **and** 🅐 exp_1 = exp_3 🅐	$=_F$	exp_1 = ⓐ exp_2 ⓐ op 🅐 exp_3 🅐

where

op ∈ {+, &}.

It is also possible to merge two multiplicity tests with the following law, if their annotations are a partition.

Law 34 (Merge multiplicity test)**.**

ⓐ mlt exp_1 ⓐ op_1 ● mlt exp_2 ●	$=_F$	mlt (ⓐ exp_1 ⓐ op_2 ● exp_2 ●)

where

mlt $\in$ {**no**, **lone**, **one**, **some**}, $\text{op}_1 \in$ {**and**, **or**}, $\text{op}_2 \in$ {+, &}.

Likewise for quantifiers.

Law 35 (Merge quantification)**.**

ⓐ qnt n:exp_1 | frm_1 ⓐ **and** ● qnt n:exp_2 | frm_2 ●	$=_F$	qnt n:ⓐ exp_1 ⓐ+● exp_2 ● | ⓐ frm_1 ⓐ **and** ● frm_2 ●

where

qnt $\in$ {**all**, **some**, **lone**, **one**, **no**}.

Finally, we present two laws for merging expressions involving the essential Alloy join operator. Since join does not distribute over intersection, merging the intersection of two join expressions (when one of the operands is the same) is only possible when the respective annotations form a partition.

Law 36 (Left distribute join over intersection)**.**

ⓐ exp_1.exp_2 ⓐ&● exp_1.exp_3 ●	$=_F$	exp_1.(ⓐ exp_2 ⓐ&● exp_3 ●)

Since the join operation distributes over union, merging the union of two join expressions can be done independently of the annotation context.

Law 37 (Left distribute join over union)**.**

ⓐ exp_1.exp_2 ⓐ + ⓑ exp_1.exp_3 ⓑ	$=_F$	exp_1.(ⓐ exp_2 ⓐ+ⓑ exp_3 ⓑ)

Similar laws are defined for right distributivity.

Back to the fact resulting from the merging of the field `inside`, after applying Laws 19, 35, 34, 29, it can be refactored into

```
fact{ ①all x:Category | one x.inside① }
```

This allows the application of Laws 20 and 16 to push the multiplicity back into the field and remove the fact.

```
➀sig Category {
  inside: one ➋Catalog➋+➁Catalog+Category➁
}➀
```

Finally, with an application of Law 29, we can have the following declaration, meaning that a category is always inside exactly one element, which is either a `Catalog` or another `Category` if hierarchical categories are supported.

```
➀sig Category {
  inside: one Catalog+➁Category➁
}➀
```

As another example, consider fact `Thumbnails` from our running example (Fig. 48,l. 18–21). With Laws 35, 31 and 36 it can be refactored into

```
fact Thumbnails { all c:Catalog |
  c.thumbnails in (➊catalog.c➊&➀category.(➋inside➋+➁^inside➁).c➀).images
}
```

The resulting fact is more compact, but whether it improves model comprehension is in the eyes of the designer.

5.3 Migrating Clones into a Colorful Alloy Model

As we have seen, approaches to SPL engineering can either be *proactive* – where an *a priori* domain analysis establishes the variability points that guide the development of the product family, *reactive* – where an existing product family is extended as new products and functionalities are developed, or *extractive* – where the family is extracted from existing software products with commonalities (Krueger, 2001). Colorful Alloy was initially conceived with the proactive approach in mind, with annotations being used precisely to extend a base model with the variability points addressing each desired feature. The model in Fig. 48 could be the result of a such a proactive approach to the design of the e-commerce platform.

With plain Alloy, to develop this design we would most likely resort to the clone-and-own approach. First, a base model, such as the one in Fig. 60 would be developed. This model would then be cloned and adapted to specify a new variant adding support for categories, as depicted in Fig. 61. This model would in turn be further cloned and adapted twice to support hierarchical or multiple categories. A final clone would then be developed to combine these two features. These last three clones are not depicted, but

```
sig Product {
  images: set Image,
  catalog: one Catalog
}
sig Image {}
sig Catalog {
  thumbnails: set Image
}
fact Thumbnails {
  all c:Catalog | c.thumbnails in (catalog.c).images
}
pred Scenario {
  some Product.images
}
run Scenario for 10
```

Figure 60: E-commerce base model (variant ❶❷❸).

```
sig Product {
  images: set Image,
  category: one Category
}
sig Image {}
sig Catalog {
  thumbnails: set Image
}
fact Thumbnails {
  all c:Catalog | c.thumbnails in (category.inside.c).images
}
sig Category {
  inside: one Catalog
}
pred Scenario {
  some Product.images and all c:Category | lone category.c
}
run Scenario for 10
```

Figure 61: Clone introducing categories (variant ①❷❸).

they would very likely correspond to something like the projections of the colorful model in Fig. 48 over the respective feature combinations. This section first presents an extractive approach that could be used to migrate all such plain Alloy clone variants into a single Colorful Alloy model. Later we will also show how this technique can be adapted for a reactive scenario, where each new clone variant is migrated into a Colorful Alloy model already combining previous clones.

Our technique follows the idea proposed by Fenske et al. (2017) for migrating Java code clones into an SPL: first combine all the clones in a trivially correct, but verbose, initial SPL, and then improve it with a step-wise process using a catalog of variant-preserving refactorings. However, as we mentioned in the beginning of the previous section there are some key differences in our approach: we don't deal with alignment issues (although we also require some preparatory refactorings, changing the name of entities is

```
1  fact FeatureModel {
2    ② ❶ some none ❶ ② and
3    ③ ❶ some none ❶ ③
4  }
5
6  ❶❷❸ sig Product {
7    images: set Image,
8    catalog: one Catalog
9  }❸❷❶
10 ...
11 run Scenario with ❶,❷,❸ for 10
12
13 ①❷❸ sig Product {
14   images: set Image,
15   category: one Category
16 }❸❷①
17 ...
18 run Scenario with ①,❷,❸ for 10
19
20 ①②❸ sig Product {
21   images: set Image,
22   category: one Category
23 }❸②①
24 ...
25 run Scenario with ①,②,❸ for 10
26 check AllCataloged with ①,②,❸ for 10
27
28 ①❷③ sig Product {
29   images: set Image,
30   category: some Category
31 }③❷①
32 ...
33 run Scenario with ①,❷,③ for 10
34
35 ①②③ sig Product {
36   images: set Image,
37   category: some Category
38 }③②①
39 ...
40 run Scenario with ①,②,③ for 10
41 check AllCataloged with ①,②,③ for 10
```

Figure 62: Part of the initial migrated e-commerce colorful model.

not one of them); we also address Type-3 clones (namely, our refactoring catalogue includes fine-grained laws for expressions or formulas); our technique will not create features along the process. Regarding the latter, we assume that prior to clone migration the user identified the implemented features and the respective FM during domain engineering, and identified which variant (feature combination) corresponds to each clone.

As such, to obtain the initial (trivially correct) Colorful Alloy model it suffices to migrate every clone to a single model, annotating all paragraphs and commands of each clone with the feature expression that

exactly describes the respective variant. For example, for the e-commerce example, the base model of Fig. 60 would be annotated with the feature expression ❶❷❸, since this clone does not specify any of the three features, the clone of Fig. 61 would be annotated with the feature expression ①❷❸, since it specifies the variant implementing only simple categories, and so on. If some feature combinations are invalid according to the FM (that is, there are only clones for some of the combinations), a fact that prevents the forbidden combinations should also be added, similar to the `FeatureModel` fact of Fig. 48. For the e-commerce example, part of the initial colorful model with all five variants is depicted in Fig. 62. Notice that, since all of the elements of the different clones are included and annotated with disjoint feature expressions, this Colorful Alloy model trivially and faithfully captures all the variants, although being quite verbose.

After obtaining this initial model, the refactorings presented in the previous section can be repeatedly used in a step-wise fashion to merge common elements, reducing the verbosity (and improving the readability) of the model. For the structural elements the key refactorings are merging signatures (Law 15) and fields (Law 17), but, as already explained, some additional preparatory refactorings might be needed to enable those, for example reordering (or removing redundant) feature annotations or removing multiplicity qualifiers.

For example, in the initial model of Fig. 62 we can start by merging signature `Product` (and the respective fields) from clones ❶❷❸ and ①❷❸ and obtain

```
❷❸sig Product {
  images: set Image,
  ❶catalog: one Catalog❶,
  ①category: one Category①
}❸❷
```

and then merge this with the definition from clone ①②❸ (by first removing the redundant feature annotation ① to enable the application of Law 15 – notice that from the FM we can infer that ② implies ①) in order to obtain

```
❸sig Product {
  images: set Image,
  ❶❷catalog: one Catalog❷❶,
  ①category: one Category①
}❸
```

The same result would be obtained if we first merged the declarations of `Product` from clones ①❷❸ and ①②❸, and then the one from clone ❶❷❸ (in this case, to apply Law 15 we would first need

to remove the redundant annotation ❷, since from the FM we can also infer that ❶ implies ❷). By repeatedly merging the variants of `Product` we can eventually get to the ideal (in the sense of having the least duplicate declarations) definition for this signature.

```
sig Product {
  images: set Image,
  ❶catalog: one Catalog❶,
  ①category: set Category①
}
①fact {
  all p:Product | ❸one p.category❸ and ③some p.category③
}①
```

If we repeat this process with all other model elements, we eventually get a (slightly optimized, in terms of number of declarations) version of the Colorful Alloy model in Fig. 48.

A similar technique can be used to migrate a new clone into an existing colorful model, thus enabling a reactive approach to SPL engineering. Let us suppose we already have the ideal colorful model for e-commerce, but we decide to introduce a new variant to support multiple catalogs when categories are disabled (a new feature ④). The definition of `Product` for this clone would be

```
sig Product {
  images: set Image,
  catalog: some Catalog
}
```

To migrate this clone to the existing colorful SPL we would annotate the elements of the new variant with the feature expression that characterizes it, ❶❷❸④, annotate all elements of the existing SPL with ❹ (since neither of its variants support this new feature), refine the FM to forbid invalid variants (adding **`some none`** annotated with ①④ to forbid the new feature in the presence of categories), and then restart the refactoring process to improve the obtained model.

5.4 Automatic Merging Strategy

In order to simplify the application of the step-wise refactoring technique described in the previous section, we also propose an automatic merging strategy that implements a sequence of refactoring laws in one composed step. This strategy supports the developers in automating the tedious and error-prone merge tasks and considerably reduces the number of steps (and overall time) to perform clone migration.

To merge the declarations of a particular signature, the strategy repeatedly tries to find pairs of declarations of that signature that can be merged using Law 15, that is, where the respective annotations form a partition of the variants (ignoring feature annotation order, by implicit application of Law 11). The two declarations are first aligned using preparatory refactorings: if different, the multiplicity and **`abstract`** qualifiers from each declaration are moved into facts with Laws 13 and 14. When no more pairs of declarations can be merged by direct application of Law 15, the strategy tries to find a pair of declarations that could be merged if (at most) one redundant feature is removed from one of the annotations. We limit the search to one redundant feature for efficiency reasons. If such a pair of declarations is found, the redundant feature is removed using Law 12 and the process resumes. Whenever a pair of signature declarations is merged, a similar strategy is used to merge the field declarations inside. Similarly to signatures, if necessary, the multiplicity annotations of fields are first removed with Law 16, and when no pair of field declarations can be merged directly with Law 17, the strategy tries to find a pair where removing one redundant feature would enable merging. Similar laws are used for fields with different arities. To merge the respective bounding expressions the strategy for merging expressions detailed below can be applied. In most cases it suffices to apply Law 29 to merge common expressions.

To illustrate this merging strategy, consider its application to signature `Product` in our example. The strategy will first merge declarations whose annotations partition the variants, for example the two declarations from clones ❶❷❸ and ①❷❸, and the two declarations from clones ①②❸ and ①②③. This choice would lead to the following result, where no more pairs of declarations can be directly merged with Law 15.

```
②③sig Product {
  images: set Image ,
  ①catalog: some Catalog①,
  ①category: one Category①
}③②
①②sig Product {
  images: set Image,
  category: set Category
}②①
①②③sig Product {
  images: set Image,
  category: some Category
}③②①
①②③fact { all p: Product | some p.category }③②①
①②③fact { all p: Product | one p.category }③②①
```

Note the two facts that were created to align the declarations of field `category`. At this point, the strategy tries to find a pair of declarations that could be merged if one redundant feature is removed. For example, if redundant feature ① is removed from the third declaration then it could be merged with the first one. As such, this redundant feature is removed, the automatic signature merging process resumed and those two declarations merged. Afterwards, we would end up with two declarations for `Product` that could not be directly merged using Law 15, namely with annotations ❷ and ①②. Again, removing redundant feature ① from the latter would enable the merging. After finishing the automatic signature and field merging phase we would end up with the following single declaration for `Product`.

```
sig Product {
  images: set Image,
  ❶❷❸catalog: some Catalog❸❷❶,
  ①category: set Category①
}
①②③fact { all p: Product | some p.category }③②①
①②❸fact { all p: Product | one p.category }❸②①
①❷③fact { all p: Product | some p.category }③❷①
①❷❸fact { all p: Product | one p.category }❸❷①
```

Notice that field `catalog` is still annotated with two redundant features (❷ and ❸) that the developer may later opt to remove. The automatic strategy only removes redundant features if they enable the merging of two declarations.

Import statements, facts, predicates, functions, assertions, and non-block commands with formulas can then be merged with Laws 18, 19, 21, 22, 23, 25, and 26, respectively. Block commands are merged with similar laws. Import statements, predicates, functions, assertions, and commands are merged using a similar strategy to signatures. Pairs of paragraphs that can be directly merged with the respective laws are first repeatedly processed, and once no more such pairs remain, the strategy tries to find a pair where removing a redundant feature enables merging. Since facts can be merged irrespective of the annotations they have, all facts with the same identifier will be merged in one step. Although in the above example the facts created to align field declarations are not named, in the actual implementation they have an internal identifier to ensure that the generated facts from each signature are merged separately. The annotated formulas and expressions obtained after this iterative process are then merged by repeatedly applying laws for formulas and expressions from left to right (with the exception of Law 28 that does not reduce the size of the expression). In Laws 33 and 34, where there is a choice of operator to introduce in the result, the strategy is currently opting for `+`. The automated strategy is also implicitly using commutative laws, for

example, also merging common expressions in the right-hand side with Law 29, and also applying the law variants described above, namely the ones that result from combining Law 29 with Law 27.

Using this strategy, the five clones of our example could be merged in single step, obtaining the model in Fig. 63[2]. This model has some small differences when compared to the one in Fig. 48:

- Field `catalog` still has some redundant features in the respective annotation.
- There is a single declaration for field `category`, but an additional fact with the respective multiplicity constraints in different variants.
- There is a single declaration for signature `Category` and the respective `inside` field.
- There is a single expression inside fact `Thumbnails`, and a & operator is obtained instead of + in the sub-expression that chooses `^inside` or `inside` depending on the presence of feature ②.
- The annotations on fact `Acyclic` were pushed inside into the corresponding formula.
- There is one redundant feature ① in the annotation of assertion `AllCataloged`, and this annotation marks the all assertion instead of just the inner formula.

Although the resulting model is smaller, one may argue that some of the merged declarations can actually reduce the comprehension, namely the single declaration for field `category`. If the user so wishes it would be possible, after the automatic strategy, to apply some manual refactoring steps and obtain a model identical to Fig. 48. For example, to obtain the same `AllCataloged` assert, we could start by removing the redundant annotation with Law 12 and introduce a trivial assertion with the same name annotated with the opposite feature using Law 24.

```
②assert AllCataloged {
  all p:Product | some (p.category.^inside & Catalog)
}②
②assert AllCataloged { no none }②
```

These assertions can now be merged with Law 23, resulting in the following declaration.

```
assert AllCataloged {
  ②all p:Product | some (p.category.^inside & Catalog)② and ②no none②
}
```

2 Currently our implementation pretty-prints the resulting models with spurious parenthesis, but here we opted to remove the unnecessary ones to ease the understanding of the result. In the near future we intend to solve this issue, using a more sophisticated pretty-printer.

```
1  fact FeatureModel {
2    --(2) Hierarchical requires (1) Categories
3    (2)(1)some none(1)(2)
4    --(3) Multiple requires (1) Categories
5    (3)(1)some none(1)(3)
6  }
7
8  sig Product {
9    images: set Image,
10   (1)(2)(3)catalog: some Catalog(3)(2)(1),
11   (1)category: set Category(1)
12 }
13 fact {
14   (1)all p: Product | (3)one p.category(3) and (3)some p.category(3)(1)
15 }
16
17 sig Image {}
18 sig Catalog {
19   thumbnails: set Image
20 }
21
22 fact Thumbnails {
23   all c:Catalog | c.thumbnails in
24     ((1)catalog(1) & (1)category.((2)inside(2) & (2)^inside(2))(1).c).images
25 }
26
27 (1)sig Category {
28   inside: one Catalog+(2)Category(2)
29 }(1)
30
31 fact Acyclic {
32   (1)(2)all c:Category | c not in c.^inside(2)(1)
33 }
34
35 pred Scenario {
36   some Product.images and (1)all c:Category | lone category.c(1)
37 }
38 run Scenario for 10
39
40 (1)(2)assert AllCataloged {
41   all p:Product | some (p.category.^inside & Catalog)
42 }(2)(1)
43 check AllCataloged with (1),(2) for 10
```

Figure 63: E-commerce specification obtained with the automatic merging strategy.

Finally the formula can be simplified by removing the annotated neutral element using Law 27, resulting in the exact same declaration of Fig. 48.

6

IMPLEMENTATION AND EVALUATION

This chapter describes the implementation of Colorful Alloy and its Analyzer, as well as its evaluation. The latter has 3 main goals: to assess the expressiveness of the language, the performance and scalability of the colorful analysis procedures, and the feasibility of the automated clone migration strategy. The colorful representation of a model and the refactoring rules are implemented as an extension to the Alloy Analyzer, as detailed in Section 6.1. Section 6.2 describes a set case studies modelled in Colorful Alloy following a proactive approach to SPL engineering. Section 6.3 presents a set of Alloy clone families collected from the literature, and the result of applying the migration strategy following a extractive approach to SPL engineering. Lastly, Section 6.4 presents the performance results for the colorful models from Section 6.2 and Section 6.3, comparing the performance of the amalgamated analysis with that of a iterative analysis based on projection.

6.1 The Colorful Alloy Analyzer

The Colorful Alloy language, the analysis procedures, and the catalog of refactorings were implemented in the Alloy Analyzer[1]. Figure 64 presents an overview of the Colorful Alloy Analyzer, with the editor with colorful backgrounds and annotations, and the visualizer showing a running scenario and a panel indicating which variant it belongs to.

The colorful backgrounds and feature annotations are implemented in the Alloy editor, shown in the left-hand side of the background window in Fig. 64. A colorful model consists simply of plain text, annotated Alloy files, and is decoupled from the colorful Analyzer. Nonetheless, the colorful editor provides keyboard shortcuts to support the introduction of features in the model. Pressing *Ctrl* and a number key (1 to 9) inserts the respective positive annotation of the feature indicated by the number, while pressing *Ctrl* plus

1 All Colorful Alloy source code and example models are publicly available at `https://github.com/chongliujlu/ColorfulAlloy/`.

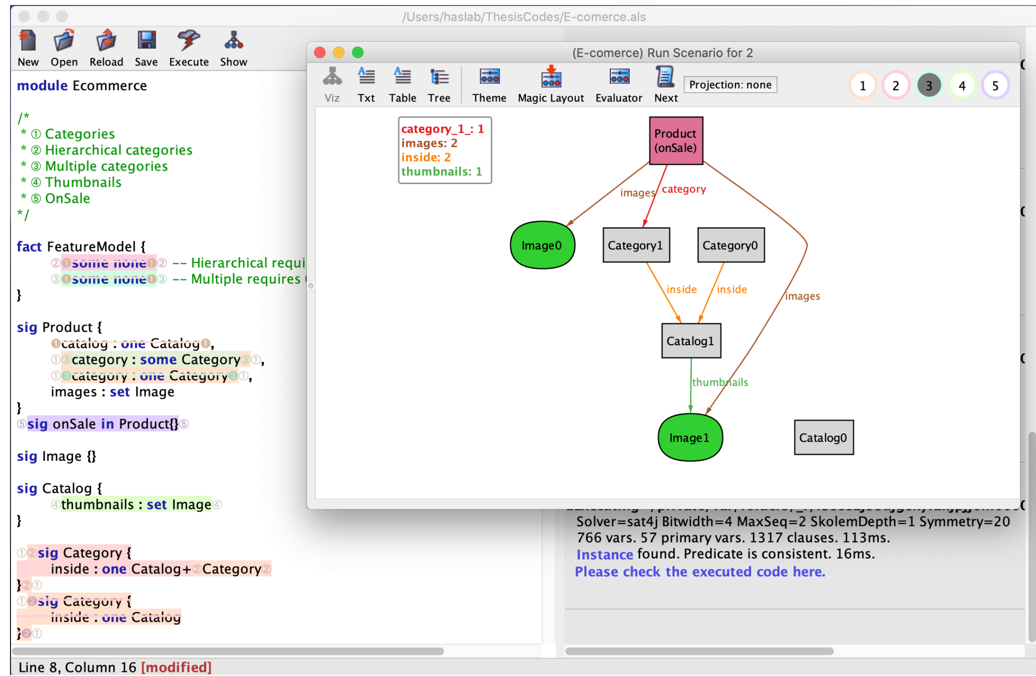


Figure 64: The e-commerce in the Colorful Analyzer, showing the editor and the visualizer.

Shift plus a number key inserts the corresponding negative feature annotation. If an excerpt of code is selected in the editor, the annotation is applied in pair to wrap the selected text, otherwise an individual annotation is added in the cursor position. Each annotation is presented in the editor with its assigned color. Following the results from (Feigenspan et al., 2013), we selected low-saturation colors for each feature, to avoid visual fatigue. For annotated code, the corresponding background color (for positive features) or a colored strike-through line (for negative features) is added automatically in the editor. Therefore, the user only needs to focus on introducing feature annotations and not worry about managing backgrounds. We implement color mixing both for backgrounds and strike-through lines for elements annotated with multiple features. While this was discarded in (Feigenspan et al., 2013) due to the number of features encountered in C code, our approach supports a fixed small number of distinct features that we believe adequate for the typical complexity of Alloy models, as we shall show in the remaining of this chapter. Analysis commands are not wrapped in annotations, but provided a feature scope through comma-separated annotations, and are not highlighted with background colors. The special annotation ⓪ can also be inserted by pressing *Ctrl* + 0). Using this annotation, in the e-commerce example used in this chapter where only five features are used, the command **run** `Scenario` **with exactly** ❶,❷,❸,❹,❺ **for** `2` can be replaced with **run** `Scenario` **with exactly** ⓪ **for** `2`.

Once the model is defined, the user can instruct the Colorful Analyzer to generate a scenario with a certain property through a **run** command or to find a counter-example to a particular property that is expected to hold through a **check** command. Annotation parsing errors (e.g., annotations not applied

to a complete AST node) and typing errors (e.g., calling elements in invalid colorful contexts), as defined in Chapter 4, are detected and reported when such commands are executed, and presented to the user in the editor's logging panel as in the regular Alloy Analyzer (right-hand side of the background window in Fig. 64). As with normal Alloy, these commands are provided with a scope indicating the maximum size of model's signature to be considered, but in the colorful Analyzer they can additionally be provided with a feature scope to control the variants that should be explored. These are a set of (positive and negative) features, and analysis will either consider all variants for which that presence condition holds, or the smallest variant if marked with `exactly`. In general, the analysis implements the amalgamated translation from Section 4.6, unless the command is analyzing exactly one variant, in which case the projected variant is analysed (see Section 6.4 for performance evaluation). To help users understand the semantics of the colorful analysis, a link in the logging panel at the end of the execution information allows the inspection of the corresponding amalgamated model (much like in plain Alloy the resulting Kodkod code can also be inspected). In the example, command `run Scenario with ① for 2` allows the user to explore scenarios for all 16 variants where feature ① is selected.

Instances or counter-examples generated during the analysis are presented in the Analyzer's visualizer, as shown in the foreground window in Fig. 64. Since the amalgamated analysis converts features into regular Alloy elements, they would appear as regular instance nodes in the visualizer, which could make it unclear to the user to identify to which variant the presented instance applied. For a feature-oriented perspective, the features are extracted from the generated instance and presented in the top right corner of the visualizer. Each ⓒ ($1 \leq c \leq 9$) represents the presence of a feature in the variant under analysis, and 🅒 its absence (only features relevant for the model under analysis are presented, in this case five). Therefore, the example shown in Fig. 64 shows a scenario for variant ①②④⑤. Iteration can then be used to inspect alternative scenarios, which will may either present a different scenario for the same variant or an alternative variant. Finer, feature-aware scenario exploration operations are left as future work.

The proposed catalog of refactoring rules is also implemented on top of the colorful Analyzer. Most individual refactorings are implemented in a contextual menu, activated by right-clicking an element in the editor. These include the rules to manage annotations and all those that do not act at the expression level, applied from the left-hand side to the right-hand side of the rule, which reduces the number of declarations in the model (rules over expressions are not implemented at this level because their context would be difficult to identify by simply right-clicking an element). The Analyzer automatically detects which refactorings can be applied in the selected context by inspecting the parsed AST. It also scans the model facts to extract FM constraints from statements with the shape ⓐ`some none`ⓐ, so that the application of laws with

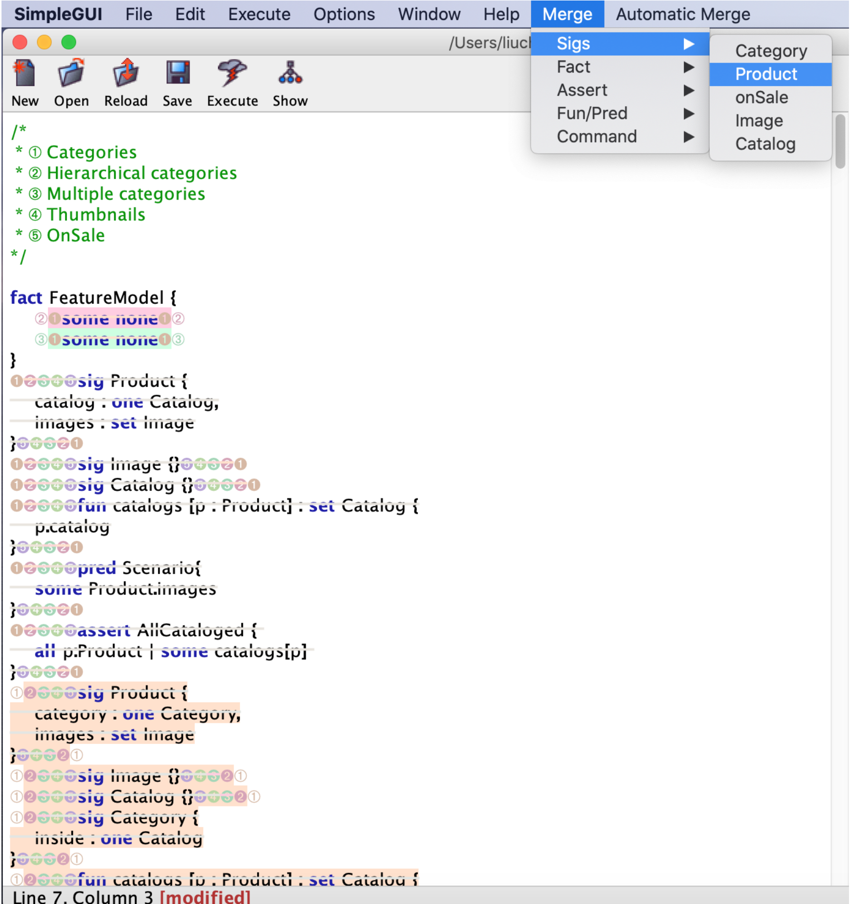


Figure 65: Automatic merge strategies.

preconditions on feature dependencies (namely Law 12) can be automated. For efficiency reasons, the prototype implements an incomplete decision procedure to check these preconditions, considering only simple implications directly derived from the FM. This does not affect the soundness of the procedure but may fail to automatically detect some possible rule applications. The automatic merging strategy from Section 5.4 has also been implemented and is accessible through the main menu. Besides the application of the automatic strategy to all elements, the user may also choose to only automatically merge certain elements, such as signatures or facts.

Figure 65 shows the menu with the automatic merge refactorings for our running example. If the option to automatically merge every element is selected, a model similar to that from Fig. 63 would be achieved, as depicted in Fig. 66. As already discussed, in this version certain redundant annotations are still present, such as ❷ and ❸ over the `catalog` field due to the ❶ annotation. These can removed using the contextual menu through right-clicking in `catalog` as shown in Fig. 66. The other merging strategies could be used for a more step-wise migration. For instance, if the option to merge signature `Product` was selected we would end up with three declarations for `Product`, since merging them any further requires the removal of a redundant feature. This could be performed manually through right-clicking in

SimpleGUI File Edit Execute Options Window Help Merge Automatic Merge
/Users/liuchong/Desktop/Ecommerce.als
New Open Reload Save Execute Show

```
/*
 * ① Categories
 * ② Hierarchical categories
 * ③ Multiple categories
 * ④ Thumbnails
 * ⑤ OnSale
 */

fact FeatureModel {
        ②➊some none➊②
        ③➊some none➊③
}

sig Product {
        ①category:  set Category①,
        images:  set Image,
        ➊➋➌catalog:  one Catalo
}
⑤sig onSale in Product { }⑤
①sig Category {
        inside:  one (Catalog + ②Category②)
}①
sig Image { }
sig Catalog {
        ④thumbnails:  set Image④
}

fact Thumbnails {
        ④all  c : Catalog |  c .thumbnails in (➊catalog. c ➊ & ①category.((②inside② & ② ^(inside)②).
}
fact Acyclic {
        ①②all  c : Category |  c  !in  c . ^(inside)②①
}
fact OnSaleThumbnails {
        ④⑤all  p : onSale, c : (catalogs[ p ]) |  some ( p .images &  c .thumbnails)⑤④
}
```

Remove ❷
Remove ❸

Line 17, Column 32 [modified]

Figure 66: Contextual refactoring menu Remove Feature.

`Product`, and then selecting again the automatic merge signature for `Product` would result in a single product. A similar process would then need to be done to merge the three `category` fields.

6.2 Proactive Case Studies

In this section, we describe six SPL case studies that were implemented in Colorful Alloy following a proactive approach. The goal is to evaluate the expressiveness of the language (and later in the chapter, the performance of the analysis procedures). Some of these have been selected from the literature, while other arose in the context of other research activities where there was need to reason about design variants.

6.2.1 *E-commerce*

The *e-commerce* case study used here is a comprehensive description of the e-commerce example adapted from (Czarnecki and Pietroszek, 2006) and used in previous chapters. The Colorful Alloy version used for

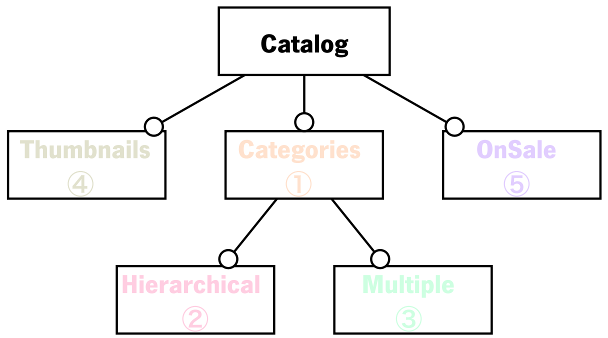


Figure 67: Feature diagram of the e-commerce specification.

evaluation extends the example described in Chapter 4 (Fig. 48) to the full FM presented in Fig. 18, except for the Image feature. This feature was omitted because it simply introduced additional signatures in the hierarchy of signature `Image`, which are not interesting when describing the expressiveness of Colorful Alloy since no additional constraints are introduced in the model. The resulting FD is displayed in Fig. 67 (where the mandatory feature Image is omitted since it would have no children in this version). The code of the full model, extending the simpler version described in Chapter 4, is available in Fig. 92 of Appendix A.

As described in previous chapters, the catalog structure of an e-commerce platform can be enhanced by classifying products into ① *Categories* that can optionally support the ② *Hierarchical* and ③ *Multiple* categories. The thumbnail image of the product is individually described as an optional feature ④ *Thumbnails*. This feature introduces an additional field on `Catalog`, and a fact to control the presence of thumbnails in a catalog, whose behaviour depends on whether categories are hierarchical or not. Another optional feature is to have products ⑤ *OnSale*, represented by an additional signature that extends `Product` by inclusion, representing the sub-set of products on sale. A new fact is introduced when both Thumbnails and OnSale are present that forces products on sale to display thumbnails in the respective catalog. This fact relies on an auxiliary function to retrieve all catalogs, whose body is also annotated.

6.2.2 *GrandpaFamily*

The *GrandpaFamily* model is based on two toy models by Jackson (2012) distributed with the Alloy Analyzer and that share certain elements: one modeling genealogical relationships (`genealogy`) and other solving the "I'm My Own Grandpa" puzzle (`grandpa`), that originated from a novelty song with the same title from 1940s. The latter is actually presented in stages to address different concepts, which are distributed as three distinct Alloy files. Our base variant considers basic biological facts, which can be extended by ① introducing Adam and Eve, who are considered as the first man and woman according to the

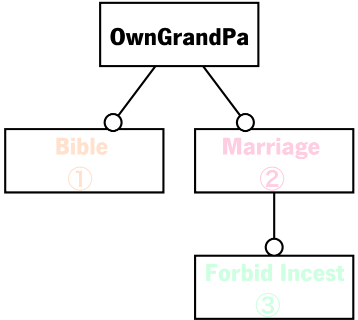


Figure 68: Feature diagram of the GrandpaFamily specification.

Bible creation myth. Then ② introduced social norms regarding marriage, optionally also ③ forbidding incestuous marriages. The FD is shown in Fig. 68 with a hierarchy relation on feature ② and ③, and the corresponding Colorful Alloy code of the design variants is given in Fig. 69. The base model (parts without annotation) simply introduces a person, either a man or a woman, who cannot be his own ancestor and can at most have one female and one male parent. A command checks whether a person can be his own grandpa, as specified in the *OwnGrandpa* assertion. Feature ① adds Adam and Eve as singleton signatures extending `Person`, with a new fact stating that only Adam and Eve have no parents. A new command checks whether all persons descend from Adam and Eve in variants with feature ①. On the other hand, feature ② introduces a spouse relation in `Person` and one additional fact that avoids reflexive and non-symmetric marriages and a social norm (valid at the time of the song) that a man can only marry one woman and vice versa. Feature ③ prohibits incest, that is, a person can not marry a sibling or a parent, through an additional fact.

6.2.3 *Alloy4Fun*

Alloy4Fun (Macedo et al., 2019) is a web-platform[2] developed by our team for learning Alloy and sharing models. Besides the online creation, analysis and sharing of models, Alloy4Fun has two additional goals: to provide a kind of auto-grading feature by marking certain parts of the model as secret, and to collect information regarding usage patterns and typical pitfalls when modeling in Alloy. To explore alternative design variants, a Colorful Alloy model was developed in its initial stages of development. The full code of this model is presented in Fig. 70 and the corresponding FD is shown in Fig. 71.

The base variant simply stores models when shared by the user, and is thus comprised by stored models (signature `StoredModel`) and assigns to a model at most one public link. By accessing this

2 `alloy4fun.inesctec.pt`

```
fact FeatureModel {
  -- ③ requires ②
  ③②some none②③
}

abstract sig Person {
  ②spouse : lone Person②,
  parents  : set Person
}

sig Man, Woman extends Person {}
①one sig Eve extends Woman {}①
①one sig Adam extends Man {}①

fact Biology {
  -- nobody is his or her own ancestor
  no p: Person | p in p.^parents
  -- every person has at most one female and one male parent
  all p : Person | lone p.parents & Woman and lone p.parents & Man
}

①fact Bible {
  -- every person except Adam and Eve has a mother and father
  all p: Person - (Adam + Eve) | one mother: Woman, father: Man | p.parents = mother + father
  -- Adam and Eve have no parents
  no (Adam + Eve).parents
  -- Adam's spouse is Eve
  ②Adam.spouse = Eve②
}①

②fact SocialNorms {
  -- nobody is his or her own spouse
   no p: Person | p.spouse = p
  -- spouse is symmetric
   spouse = ~spouse
  -- a man's spouse is a woman and vice versa
   Man.spouse in Woman && Woman.spouse in Man
}②

②③fact NoIncest {
  -- can't marry a sibling
  no p: Person | some p.spouse.parents & p.parents
  -- can't marry a parent
  no p: Person | some p.spouse & p.^parents
}③②

assert OwnGrandPa {
  no p : Person | p in p.(parents+②parents.spouse②).(parents+②parents.spouse②)
}
check OwnGrandPa with ② for 10

①assert AllDescendFromAdamAndEve {
  all p : Person - (Adam + Eve) | p in ^parents.Adam and p in ^parents.Eve
}①
check AllDescend with ① for 10
```

Figure 69: GrandpaFamily specification in Colorful Alloy.

public link, the user can access the code of the full model. Additional constraints in fact `Links` force a link to be assigned to exactly one stored model and every stored model to have a public link assigned to it in the base version. To this base model, four features can be added. Feature ① *Derivation* enables the collection of derivation trees to register the evolution of models. With this feature, each model developed after accessing a shared link stores the identifier of the model it was derived from. Additional constraints related to Derivation are specified in the fact `Derivations`, avoiding cyclic dependencies and specifying

```
1   fact FeatureModel {
2     -- ④ requires ③
3     ④③some none③④
4   }
5
6   sig Link {}
7   sig StoredModel {
8     public          : lone Link,
9     ①derivationOf : lone StoredModel①,
10    ②secret        : lone Link②,
11    ③command       : lone Command③
12  }
13  ②sig Secret in StoredModel {}②
14
15  ③sig Command {}③
16  ③④sig Instance {
17      instanceOf : one Command,
18      model      : set StoredModel,
19      link       : one Link
20  }④③
21
22  fact Links {
23    -- Links are not shared between artifacts
24    all l : Link | one (public+②secret②+③④link④③).l
25    -- all models have public links, unless commands are stored
26    ③all m : StoredModel | one m.public③
27    -- Only models with secrets can have a secret link
28    ②secret.Link in Secret②
29    -- Models with secret links also have a public link
30    ②all m : Secret | some m.secret implies some m.public②
31    -- A model with secrets with a public link either has a secret link or one of the ancestors
32    ②all m : Secret | some m.public implies some m.(①^derivationOf①+iden).secret②
33  }
34  pred BadSpec {
35    -- Private and public links, if existing, must be different
36    ②all m : StoredModel | m.public != m.secret② }
37  pred GoodSpec {
38    -- Private and public links, if existing, must be different
39    ②all m : StoredModel | no m.public & m.secret② }
40  fact Derivations {
41    -- The derivations form a tree
42    ①no m : StoredModel | m in m.^derivationOf①
43    -- Models without a link can only have at most one derivation
44    ①all m : StoredModel | no m.public implies lone derivationOf.m①
45    -- Model with secret derived from secret model
46    ①②all m : Secret | ^derivationOf.m in Secret②①
47    -- A model with secrets just with a public link cannot derive into one with a secret link
48    ①②all m : Secret | (some m.public and no m.secret) implies no (*derivationOf.m).secret②①
49  }
50  fact Commands {
51    -- Commands are unique to one model and there are no commands without models
52    ③all c : Command | one command.c③
53    -- With commands a model is either stored as result permalinking xor running a command
54    ③all m : StoredModel | no m.public iff some m.command③
55  }
56  run {some command} with ②,③ for 3
57
58  fact Instances {
59    -- Auxiliary relation for visualization
60    ③④model = instanceOf.~command④③
61    -- Commands have at most one instance
62    ③④all c : Command | lone instanceOf.c④③
63  }
64
65  assert PublicSecretDisjoint {
66    -- The set of public and secret links is disjoint
67    ②GoodSpec implies no Model.public & Model.secret② }
68  check PublicSecretDisjoint with ② for 5
```

Figure 70: Alloy4fun specification in Colorful Alloy.

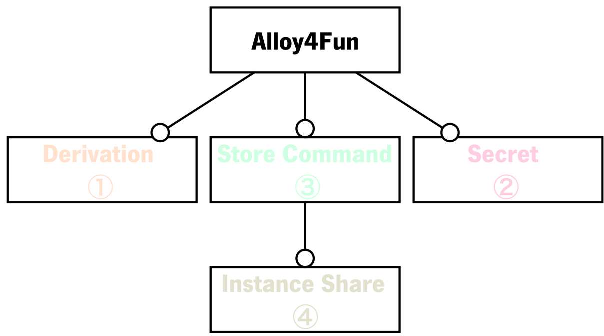
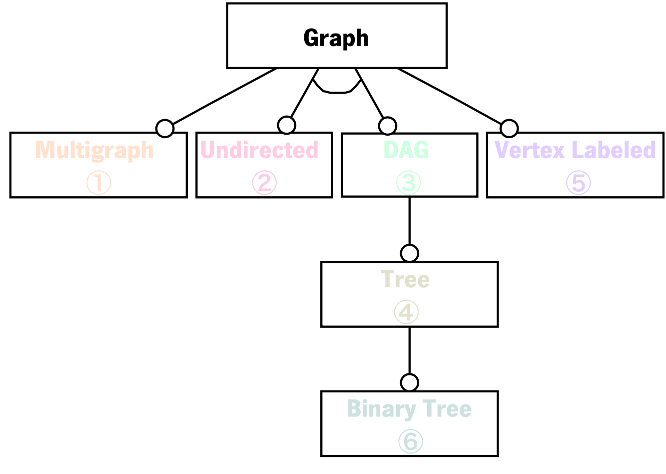


Figure 71: Feature diagram of the Alloy4fun specification.

at most one derivation for models without a link. Feature ② *Secret* allows the introduction of secret paragraphs and commands in the Alloy4Fun models. Models with secrets can now be shared with two kinds of links: a public link from the base version that, when accessed, only shows public paragraphs; and a secret one that reveals the full model including secrets paragraphs. Facts restricting the shape of the derivation tree must also be adapted when secrets are present. If a model has secrets, then all of its ancestors in the derivation also had secrets, and a model with secrets just with a public link cannot be derived into one with a secret link. Feature ③ *Store Command* allows finer data collection by storing the model when commands are executed rather than just when shared by the user. In this case, the command that originated such models is also stored and the constraint on the existence of public links is removed since stored models created through command execution are not shared. Lastly, ④ *Instance Share* allows storing and sharing the solutions of satisfiable execution commands. The constraint on links (l. 28) is relaxed, so that links may now point to stored instances rather than just stored models.

When the Store Commands feature was introduced, some run commands were specified to animate the storing of models that did not originate from sharing. One such command is shown in l. 56, and attempts to generate a scenario with some command stored in variants with secrets, as specified by the feature scope. To our surprise, such command was unsatisfiable, due to a bug in the specification of links that forbade models without links (here, represented by predicate `BadSpec` for illustrative purposes). This bug only manifested itself in variants with Secret and Store Commands, and could be missed without the feature-aware commands supported by Colorful Alloy. We often encounter such "legacy" constructs in Alloy models to illustrate interesting issues, and they could also be easily encoded as an additional feature, so that the incorrect code would be properly identified and highlighted (see, for instance, the Hotel example below). In the evaluation that will be presented in Section 6.4, this test (that under those variants, `BadSpec` guarantees that no command can exist) has been converted into a check command in order to exercise the colorful analysis procedures. This bug is fixed in predicate `GoodSpec`, that should be



Binary Tree ⑥ → Vertex Labeled ⑤

Figure 72: Feature diagram of the Graph specification.

considered an additional axiom of the model. Another assertion is specified to check whether public and private links are always disjoint.

6.2.4 *Graph*

The *Graph* example is adapted from a compositional version proposed in (Apel et al., 2013) using FeatureAlloy, which explores the specification of different classes of graphs. Its full specification is presented in Fig. 93 of Appendix A. The base model simply defines nodes and edges, which can be extended by the features presented in the FD from Fig. 72. On a first level, these features force the graph to be: ① a *Multigraph*, allowing multiple edges by relaxing a restriction in the base model; ② *Undirected*, forcing all the edges are bidirectional through an additional fact; ③ a *DAG* (Directed Acyclic Graph) by imposing an additional restriction on the adjacency relation; or ④ *Vertex Labeled* through a new field on nodes. DAG graphs can be further classified as a ⑤ *Tree* a tree, by introducing a root node and forcing all nodes but the root to have a parent. Trees can additionally be a ⑥ *Binary Tree* by restricting the outgoing edges to two. The FM additionally forbids graphs to be undirected and directed acyclic at the same time. Additionally, the FM declares that feature ⑥ also requires ⑤, since the labels `Left` and `Right` are used to mark the transitions to the two descendants of each node.

An assertion is specified for variants where the graphs form a tree, testing whether all nodes descend from the root node. Another assertion is defined that is expected to hold only for DAGs, namely that non-empty graphs have at least one source and one sink node.

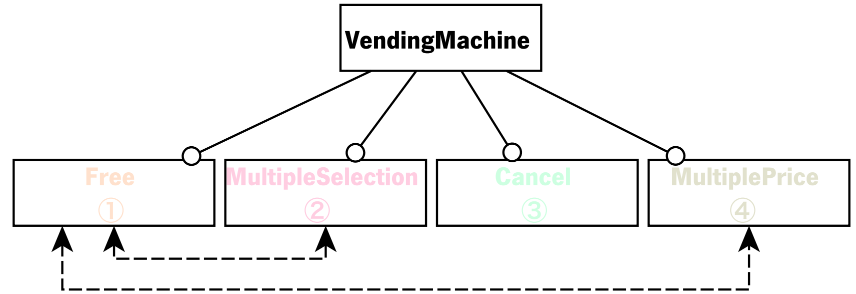


Figure 73: Feature diagram of the Vending Machine example.

6.2.5 *Vending Machine*

The *Vending machine* is an implementation of the feature transition systems described in Fig. 38 and is inspired by various vending machine examples commonly used in SPL literature (e.g., (Fantechi and Gnesi, 2008)). Unlike the examples presented so far, it models a dynamic system. A snippet of the model with a single operation `Select` is shown in Fig. 74; the encoding for other operations follows a similar style, and is presented for the interested reader in Fig. 94 of Appendix A.

The base variant encodes the process of selecting and serving products through the respective states and transitions. The machine has to count inserted money, one coin at a time, and return change after serving products. The evolution of the system is represented by a totally ordered signature `Time`, which has a `State` of the machine assigned at each instant. At each instant, the products in `stock` and the money `balance` of the machine is also registered. Lastly, interactions with the user is also registered though the current product `selection`, the `total` cost of the current selection, and the amount of `coins` that has been inserted by a customer into the machine but not yet been used to purchase something. The code of the product selection operation is shown in Fig. 74. The premise in the base version is that it can only occur in the state `Paid`, and that the selected product must be in stock and not already selected. The state is transitioned to `Selected` and the `selection` and `total` fields updated.

The base model is extensible by introducing four independent features shown in Fig. 73, by supporting: ① the serving of *Free* drinks; ② the *Multiple Selection* of different products; ③ the possibility to *Cancel* a selection; and ④ *Multiple Prices* for distinct products, adding a new field to the product signature. What's more, features ② and ④ are only allowed when free drinks are not allowed. When Free drinks are provided, states and fields controlling costs and money are removed from the model. Operations must also be adapted by annotating the predicates. In the case of `Select`, when ① Free drinks are provided, the operation is no longer triggered in the `Paid` state, but instead directly from `Ready`; moreover, restrictions related to money are removed. When ② Multiple Selection is allowed, this transition can also be triggered

```
1  module vending
2  open util/ordering[Time]
3  open util/natural as nat
4
5  abstract sig State {}
6  one sig Ready, Selected,Served extends State {}
7  1 one sig Paid extends State{} 1
8  1 one sig Done extends State{} 1
9
10 sig Product {
11  4 price : one  Natural 4
12 }
13
14 sig Time {
15  state     : one State,
16  stock     : set Product,
17  selection : set Product,
18  1 total     : one Natural 1 ,
19  1 coins     : one Natural 1 ,
20  1 balance : one Natural 1
21 }
22
23 pred Select [pre,pos : Time, p : Product] {
24   pre.state in 1 Ready 1 + 1 Paid 1 + 2 Selected 2
25   p in pre.stock - pre.selection
26   4 1 pos.total= inc[pre.total] 1 4
27   4 1 pos.total= add[pre.total, p.price] 1 4
28   pos.state = Selected
29   pos.stock = pre.stock
30   pos.selection = pre.selection + p
31   1 pos.coins = pre.coins 1
32   1 pos.balance = pre.balance 1
33 }
34 ...
35 1 assert Balance {
36   all t : Time | t.state = Served implies gte[t.balance, Zero]
37 } 1
38 check Balance with 1 for  5 but 20 Time
39
40 assert Selection {
41   all t : Time, p : Product | p not in t.stock implies all u : t.nexts |
42     p not in u.selection
43 }
44 check Selection for 5 but 20 Time
45
46 pred NoStock {
47   some t : Time | no t.stock
48 }
49 run NoStock for 5 but 10 Time
```

Figure 74: Snippet of Vending Machine specification in Colorful Alloy.

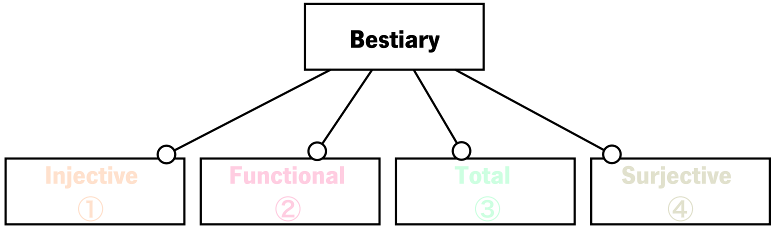


Figure 75: Feature diagram of the Bestiary specification.

when already in the `Selected` state, leading to the selection of additional products. Finally, when feature ④ Multiple Price is selected, the total price increases with the current price of the selected product (otherwise each selected product is assumed to have the price of 1).

A command tests whether a machine's balance is always greater than zero when a product is served, for variants with no free drinks. A second check command checks that once a product is out of stock, it can no longer be selected in any point in the future. A run command is also defined to explore a scenario where there exists a time when the stock of the machine is empty.

6.2.6 *Bestiary*

The *Bestiary* is a a family of very simple models that were used in classes by our team to explore different classes of binary relations. Its full code is presented in Fig. 95 of Appendix A. The base model encodes an arbitrary binary relation, and each feature forces this relation to be ① *Injective*, ② *Functional*, ③ *Total*, or ④ *Surjective* as shown in Fig. 75. Commands test alternative definitions of injectivity and functionality for all variants that are injective and functional, respectively, as well as whether relations are associative.

6.2.7 *Comparison with Compositional Approaches*

As mentioned in previous chapters, the compositional approaches such as FeatureAlloy are typically implemented through superimposition, which compose a feature-specific variant by integrating the base model with several feature-specific code units. The most evident consequence of this approach is the often coarse-grained nature of the superimposition process. For instance, in FeatureAlloy, variant modules can extend signatures with new fields, but facts, predicates or functions are simply overridden. We've seen in the presented examples that in Alloy models variation points may simply change a single formula inside a block of formulas, or even a field being called inside a complex formula. For instance, in e-commerce, the existence of hierarchical categories requires the replacement of field `inside` by its transitive closure

`^inside` at various places; in typical compositional approaches this would require the complete rewriting of the facts.

As we know, in SPL systems, semantic dependencies often occur, where one feature requires the presence of another feature, and this relationship crucially imposes the choice of the feature's code snippet. If one feature requires the presence of another feature, but the other feature is not selected, the final product will have the incorrect behavior. In our Colorful Alloy, we can easily avoid such errors by having an FM that displays the representation of feature dependencies.

Another issue that has to be considered when engineering an SPL is feature interaction, that is, features that affect the same portion of the system and thus, when used in combination, have a combined behaviour. In compositional approaches, these are usually handled by creating an additional derivative, or lifter, feature that encodes the combined behavior of the interacting features, and which must be merged on top of them. In FeatureAlloy, for instance, new derivative features would have to be developed adjusting the combined behavior. Then, when both interacting features are selected, the new derivative feature would have to be placed it the list of selected features so that the interference is eliminated. In our colorful approach, if features ⓐ and ⓑ interact, their combined behaviour can be introduced by nesting the annotations ⓐⓑ. If one wishes to disable a behaviour of ⓐ when ⓑ is present, such expression is simply annotated with ⓐ🅑. As a result, feature interaction is no longer a problem in our colorful approach. Another consequence of feature interaction is that the order in which the feature-specific code is integrated affects the code of the resulting variant, while in an annotative approach the order is irrelevant.

Annotative approaches are not without issues themselves due to the lack of modularity, namely reduced maintainability and comprehension as the number of features, and their interaction, increases. We argue however, that at the Alloy level, this process is still manageable. For our examples, we never required more than 6 features, and interactions never involved more than 2 features.

6.3 Evaluating the Clone Migration Strategy

The evaluation of the clone migration aimed to answer the following research questions: 1) Since in principle smaller specifications are easier to understand, how effective is the clone migration technique at reducing the total size of the models? 2) Is the automatic merging strategy as effective as the manual application of the refactoring rules? 3) Is our catalog of refactorings sufficient to reach an ideal colorful model specified by an expert? To answer these questions, we considered various sets of cloned Alloy models that fall in two categories: the example SPLs proactively developed by us and already detailed in the previously section,

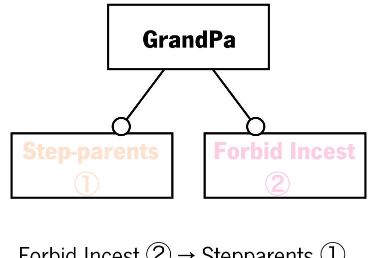


Figure 76: Feature diagram of the Grandpa specification.

and four examples developed by Jackson (2012) and packaged with the standard Alloy Analyzer distribution as sample models, for which several plain Alloy variants exist (very likely developed with clone-and-own). For the former examples, we generated the plain Alloy clones by projecting the colorful model over all the valid feature combinations. For the latter, we chose the grandpa, ring election, address book, and hotel as our examples, which will be described next.

6.3.1 *Extractive Case Studies*

The *Grandpa* example SPL considered from (Jackson, 2012) is a simpler version of the presented GrandpaFamily, with only three variants, excerpts of which are shown in Figs. 77 to 79. These models attempt to solve the "I'm my own grandpa puzzle", and each variant is developed incrementally, introducing new features to attempt to solve the puzzle. The first describes the basic structure of the model and two general constraints. That is, a person, either man or woman, has at most one father and one mother and cannot be his or her own ancestor. A man can have a wife and, similarly, a woman can have a husband, and the relationship is symmetrical. Function `grandpas` returns a person's biological grandpa. The second variant extends the notion of grandpa by including step-grandfathers through marriage. The third version adds an additional fact to avoid incest. To make the fact clear and readable, the facts are split into separate paragraphs and given suggestive names. We identified feature ① as the support for *Step-parents* and feature ② to *Forbid Incest*, as shown in Fig. 76. The presented clones were annotated as ❶❷, ①❷,and ①②, respectively. Since there is no variant forbidding incest but with step-parents, that combination is forbidden in the FM with an additional constraint.

The resulting merged code is shown in Fig. 96 of Appendix A. Notice how auxiliary predicates and functions were merged from different variants, most interesting the function defining the notion of grandpa. In our colorful refactoring, we only consider cases where the facts have the same name. Therefore, fact

```
abstract sig Person {
  father: lone Man,
  mother: lone Woman
}
sig Man extends Person {
  wife: lone Woman
}
sig Woman extends Person {
  husband: lone Man
}

fact {
  no p: Person | p in p.^(mother+father)
  wife = ~husband
}
...

fun grandpas [p: Person] : set Person {
  p.(mother+father).father
}
..
```

Figure 77: A snippet of variant ❶❷ of the GrandPa specification.

```
abstract sig Person {
  father: lone Man,
  mother: lone Woman
}
sig Man extends Person {
  wife: lone Woman
}
sig Woman extends Person {
  husband: lone Man
}

fact {
  no p: Person | p in p.^(mother+father)
  wife = ~husband
}
...
fun grandpas [p: Person] : set Person {
  let parent = mother + father + father.wife + mother.husband |
    p.parent.parent & Man
}
...
```

Figure 78: A snippet of variant ①❷ of the GrandPa specification.

```
abstract sig Person {
  father: lone Man,
  mother: lone Woman
}

sig Man extends Person {
  wife: lone Woman
}
sig Woman extends Person {
  husband: lone Man
}

fact Biology {
  no p: Person | p in p.^(mother+father)
}
fact Terminology {
  wife = ~husband
}
fact SocialConvention {
  no (wife+husband) & ^(mother+father)
}
...

fun grandpas [p: Person] : set Person {
  let parent = mother + father + father.wife + mother.husband |
    p.parent.parent & Man
}
...

pred SocialConvention1 {
  no (wife + husband) & ^(mother + father)
}
pred SocialConvention2 {
  let parent = mother + father {
    no m: Man | some m.wife and m.wife in m.*parent.mother
    no w: Woman | some w.husband and w.husband in w.*parent.father
  }
}
assert Same {
  SocialConvention1 iff SocialConvention2
}
check Same
```

Figure 79: A snippet of variant ①② of the GrandPa specification.

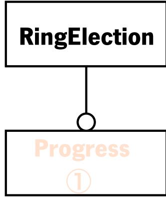


Figure 80: Feature diagram of the Ring Election specification.

`Biology` and `Terminology` in the third variant cannot be merged even though they have the same expression as the unnamed fact in variant 1 and variant 2.

The *Ring Election* example was selected from the two sample models packaged with the Alloy Analyzer describing a well-known distributed algorithm for leader election. The code excerpts of the models are shown in Figs. 81 and 82. The processes (sig `Process`) that participate in the election form a ring and have uniquely ordered identifiers. Each of them has a successor (`succ`) process which is its neighbor in the ring, a pool (`toSend`) of identifiers to be sent around the ring, and a set of `Time` elements indicating when a process is considered the leader (`elected`). The fact `ring` constraints that all processes are reachable from any process by its `succ`, forcing the processes to form a ring. There are two operations, `skip` and `step`, that can occur during trace execution. The `skip` simulates a step that does nothing and the `step` simulates an operation that passes an identifier (`id`) form the pool of a process (`from`) to the successor process. The expression `p.succ.prevs` represents a set of identifiers that precede `p.succ`. Therefore, the identifiers are passed by a process only if larger than its own. Expression `p.toSend.t—p.toSend.(t.prev)` represents the set of identifiers received at time `t`. The elected processes (`DefineElected`) are those that just received their own identifiers. Commands `AtMostOneElected` and `AtLeastOneElected` evaluate whether there is at most or at least one elected process. The second assertion is invalid in the first variant since every process can choose to skip in every step. This problem was fixed in the second variant with a predicate `progress` which forces the existence of a non `skip` step while there are processes with a nonempty pool. The FD is shown in Fig. 80 and the merged version of this two examples is in Fig. 97 of Appendix A.

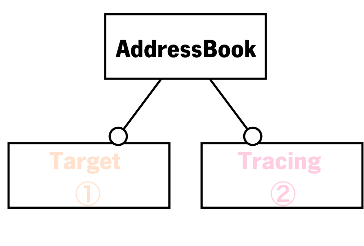
For the *AddressBook* example there are 17 variants with few differences packaged with the Analyzer. These variants are divided into three groups. The first group describes a simple address book model with names and addresses; the second group allows addresses to be organized into groups and adds an abstract notion of target to introduce a hierarchy on address entries; and the third group introduces an execution trace, allowing multiple steps to be checked from an initial state. The goal of these variants is to introduce different Alloy concepts, including a distinct dynamic idiom in the last group. For the first two groups, we selected the final version of the model, *addressBook1h.als* and *addressBook2e.als* respectively, because

```
open util/ordering[Time] as TO
open util/ordering[Process] as PO

sig Time {}
sig Process {
  succ: Process,
  toSend: Process → Time,
  elected: set Time
}

fact ring {
  all p: Process | Process in p.^succ
}

pred step [t, t': Time, p: Process] {
  let from = p.toSend, to = p.succ.toSend |
    some id: from.t {
    from.t' = from.t – id
    to.t' = to.t + (id – p.succ.prevs)
  }
}

pred skip [t, t': Time, p: Process] {
  p.toSend.t = p.toSend.t'
}

fact defineElected {
  no elected.first
  all t: Time–first | elected.t = {p: Process | p in p.toSend.t – p.
    toSend.(t.prev)}
}

assert AtMostOneElected { lone elected.Time }
check AtMostOneElected for 3 Process, 7 Time

assert AtLeastOneElected { some t: Time | some elected.t }
check AtLeastOneElected for 3 Process, 7 Time
...
```

Figure 81: A snippet of Ring Election specification, variant ❶.

```
open util/ordering[Time] as TO
open util/ordering[Process] as PO
sig Time {}
sig Process {
  succ: Process,
  toSend: Process → Time,
  elected: set Time
}

pred progress  {
  all t: Time – TO/last |
  let t' = TO/next [t] |
  some Process.toSend.t ⇒ some p: Process | not skip [t, t', p]
}
assert AtLeastOneElected { progress ⇒ some elected.Time }
check AtLeastOneElected for 3 Process, 7 Time
...
```

Figure 82: A snippet of Ring Election specification, variant ①.



Target ② → Tracing ①

Figure 83: Feature diagram of the AdressBook specification.

the other versions represent the iterative modelling of the address book, introducing new paragraphs on top of the previous versions, which would not be interesting for the merging process we are evaluating. For the third group, we just pick a simple one, *addressBook3b.als*, since the other versions have few differences and would result in a similar merging process. Excerpts of the selected variants are shown in Figs. 84 to 86 and the corresponding FD in Fig. 83. We use the *addressBook1h.als* as the base model, and two additional features, ① that adds the notion of *Target* and ② that adds *Tracing*. Again, since the variant with tracing but without target is not defined, it is excluded from the FM.

The merged version of this example is in Fig. 98 of Appendix A. Notice that the predicates encoding the operations in the base model can not be merged with those of the other variants since they have different arguments, even though the internal expressions are similar. Notice also how seamlessly the introduction of traces was, essentially imposing an order on books and facts restricting their evolution.

A snippet of the *Hotel* model is shown in Figs. 88 to 91. The first variant of the hotel example describes the process of the checking in and checking out of a hotel room locking system. This variant had a

```
sig Name, Addr { }
sig Book {
  addr: Name → lone Addr
}
pred add [b, b': Book, n: Name, a: Addr] {
  b'.addr = b.addr + n→a
}
...
```

Figure 84: A snippet of *AddressBook1h.als* specification, variant ❶❷.

```
abstract sig Target {}
sig Addr extends Target {}
abstract sig Name extends Target {}

sig Alias, Group extends Name {}

sig Book {
  names: set Name,
  addr: names → some Target
}

pred add [b, b': Book, n: Name, t: Target] {
    b'.addr = b.addr + n→t
}
...
```

Figure 85: A snippet of *AddressBook2e.als* specification, variant ①❷.

```
open util/ordering [Book] as BookOrder

abstract sig Target {}
sig Addr extends Target {}
abstract sig Name extends Target {}
sig Alias extends Name {}
sig Group extends Name {}

sig Book {
  names: set Name,
  addr: names → some Target
}
pred add [b, b': Book, n: Name, t: Target] {
    b'.addr = b.addr + n→t
}
...
fact traces {
    init [first]
    all b: Book−last | let b' = b.next |
        some n: Name, t: Target | add [b, b', n, t] or del[b, b', n, t]
}
...
```

Figure 86: A snippet of *AddressBook3b.als* specification, variant ①②.

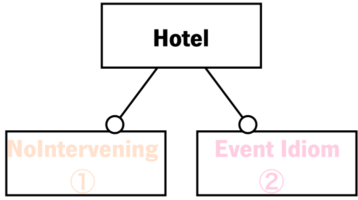


Figure 87: Feature diagram of the Hotel specification.

```
sig Key {}
sig Time {}
sig Room {
  keys: set Key,
  currentKey: keys one → Time
}
sig Guest {
    keys: Key → Time
}
...
```

Figure 88: A snippet of the Hotel variant ❶❷.

bug that generated a counter-example, which is fixed in the second version by adding an additional fact `NoIntervening` to avoid the key being entered after checking out. The third variant introduces an event-based idiom, changing the operations from predicates to being identified by the occurrence of atoms from a signature of events. The fourth version adds the `NoIntervening` constraint under the event idiom. For the migration of this model, we annotated the variants with two features: feature ① *NoIntervening* denotes whether or not the fact to fix the bug is enforced, and ② *Event Idiom* which encodes the event idiom for the evolution of the model. Therefore the presented variants are annotated with ❶❷,①❷,❶②, and ①②, respectively, and the corresponding FD is shown in Fig. 87. The merged version of these four examples is in Fig. 99 of Appendix A.

6.3.2 *Clone Migration Results*

Table 3 summarizes the results of the refactoring evaluation for the defined examples, where NP denotes the number of clones in the example, and LI and CI the total size of all plain Alloy clones measured in number of lines and characters, respectively.

To answer question 1) we applied our clone migration techniques to all of the examples, until we reached a point where no more merge refactorings could be applied, and compared the size of the resulting Colorful Alloy model with the combined size of all plain Alloy clones.

```
sig Key {}
sig Time {}
sig Room {
  keys: set Key,
  currentKey: keys one → Time
}
sig Guest {
  keys: Key → Time
}

fact NoIntervening {
  all t: Time-last | let t' = t.next, t'' = t'.next |
  all g: Guest, r: Room, k: Key |
  checkin [t, t', g, r, k] ⇒ (entry [t', t'', g, r, k] or no t'')
}
...
```

Figure 89: A snippet of the Hotel variant ①❷.

```
sig Key {}
sig Time {}
sig Room {
  keys: set Key,
  currentKey: keys one → Time
}
sig Guest {
  keys: Key → Time
}

abstract sig Event {
  pre, post: Time,
  guest: Guest
}
abstract sig RoomKeyEvent extends Event {
  room: Room,
  key: Key
}
sig Entry extends RoomKeyEvent { } {
  key in guest.keys.pre
  let ck = room.currentKey |
  (key = ck.pre and ck.post = ck.pre) or
  (key = nextKey[ck.pre, room.keys] and ck.post = key)
  currentKey.post = currentKey.pre ++ room→key
}
...
```

Figure 90: A snippet of the Hotel variant ❶②.

```
sig Key {}
sig Time {}
sig Room {
  keys: set Key,
  currentKey: keys one → Time
}
sig Guest {
  keys: Key → Time
}
abstract sig Event {
  pre, post: Time,
  guest: Guest
}
abstract sig RoomKeyEvent extends Event {
  room: Room,
  key: Key
}
sig Entry extends RoomKeyEvent { } {
  key in guest.keys.pre
  let ck = room.currentKey |
  (key = ck.pre and ck.post = ck.pre) or
  (key = nextKey[ck.pre, room.keys] and ck.post = key)
  currentKey.post = currentKey.pre ++ room→key
}

fact NoIntervening {
  all c: Checkin |
  c.post = last
  or some e: Entry {
    e.pre = c.post
    e.room = c.room
    e.guest = c.guest
  }
}
...
```

Figure 91: A snippet of the Hotel variant ①②.

Table 3: Evaluation results.

SPL	NP	Original LI	Original CI	Manual RS	Manual DL	Manual LF	Manual CF	Manual RL	Manual RC	Automatic LF	Automatic CF	Automatic RL	Automatic RC
E-commerce(3feats)	5	112	1851	101	15	34	574	70%	69%	35	586	69%	68%
E-commerce(5feats)	20	491	10197	306	17	42	757	91%	93%	42	796	91%	92%
Vending	10	942	20675	504	13	111	2304	88%	89%	113	2304	88%	88%
Bestiary	16	239	4714	207	7	22	222	91%	95%	22	231	91%	95%
OwngrandPa	6	147	3642	77	9	40	842	73%	77%	40	874	73%	76%
Alloy4fun	12	341	7353	162	14	57	1200	83%	84%	60	1300	82%	82%
Graph	18	358	7091	277	9	37	652	90%	91%	54	1160	85%	84%
RingElection	2	91	1941	25	8	52	1077	43%	43%	52	1083	43%	44%
Grandpa	3	102	1798	36	11	56	961	45%	47%	54	984	47%	45%
AddressBook	3	140	3078	26	9	75	1813	46%	41%	75	1855	46%	40%
Hotel	4	328	6653	109	9	95	2394	71%	64%	95	2458	71%	63%
Average	15	299	6272	166	11	57	1163	72%	72%	58	1239	71%	71%

The results are presented in the columns of Table 3 under Manual, where RS is the number of individual refactoring steps, DL the number of distinct refactoring laws that were used in the process, LF and CF the resulting number of lines and characters after migration, respectively, and RL and RC the reduction in relation to the original number of lines and characters, respectively. In average we achieved a reduction of around 72% on both lines and characters, which is quite substantial: the formal design of the full SPL in the final Colorful Alloy model occupies in average a quarter of the size of all the plain Alloy clones combined, which in principle considerably simplifies its understanding. The lowest reduction was for the ring elected example (around 43%), since there are only two clones to be merged. The average number of refactoring steps was 166. This number has a strong correlation with the number of clones, since the proposed merging refactorings operate on two clones at a time – if a common element exists in n clones, we will need at least $n - 1$ rule applications to merge it.

To answer question 2) we applied the automatic strategy to all examples and again compared the size of the resulting Colorful Alloy model with the combined size of the Alloy clones. The results are presented in the columns of Table 3 under Automatic, where LF and CF are the resulting number of lines and characters after automatic migration, respectively, and RL and RC the reduction in relation to the original number of lines and characters, respectively. In average, the reduction in lines and characters was only slightly smaller than the manual approaches at 71% . This is due to some issues already presented, such as the persistence of redundant annotations and the choice of + over & in some rules which may prevent further automatic refactorings.

For question 3) we relied on the seven examples where the clones were derived from previously developed Colorful Alloy models. For all of them, our catalog of refactorings was sufficient to migrate the clones and obtain the original colorful model from which they were derived. As seen in Table 3, these examples

Table 4: Evaluation of the amalgamated and iterative approaches for the proactive examples.

Model	NF	Command	Analysis	NP	NV	Scope	TA(s)	TI(s)	SU	SP(s)
E-commerce	5	AllCataloged	UNSAT	32	20	10	4.0	64.8	16.2	21.1
						11	16.9	343.0	20.3	85.2
						12	62.2	1256.3	20.2	222.2
GrandpaFamily	3	OwngrandPa	SAT	4	4	9	0.1	0.2	3.1	0.1
						10	0.1	0.3	3.7	0.1
						11	0.1	0.4	3.7	0.1
						12	0.1	0.4	4.1	0.1
		AllDescend	UNSAT	4	3	9	0.3	1.7	5.7	0.9
						10	1.0	10.9	10.9	4.0
						11	7.3	24.1	3.3	13.4
						12	26.1	132.6	5.1	57.1
Alloy4Fun	4	PublicSecretDisjoint	UNSAT	8	6	20	1.1	6.3	5.7	1.9
						25	2.8	19.6	7.0	7.6
						30	5.9	37.9	6.4	11.0
Graph	6	Connected	UNSAT	32	6	8	3.2	19.2	6.0	3.4
						9	11.9	80.9	6.8	16.7
		SourcesAndSinks	UNSAT	32	10	8	7.0	62.1	7.0	12.9
						9	187.5	1010.2	5.4	166.0
Vending Machine	4	Balance	UNSAT	8	8	20	0.4	2.2	5.7	0.4
						25	0.7	4.0	6.1	0.7
						30	1.1	6.4	5.9	1.4
		Selection	UNSAT	16	10	20	0.4	3.8	10.0	0.8
						25	0.6	7.3	11.8	2.1
						30	1.0	18.5	18.4	6.5
		NoStock	SAT	16	10	20	0.3	7.2	20.5	3.1
						25	0.6	19.3	30.6	6.3
						30	1.0	48.3	47.8	13.2
Bestiary	4	Injective	UNSAT	8	8	25	6.9	12.8	1.9	3.0
						30	9.8	49.6	5.1	16.0
		Functional	UNSAT	8	8	25	2.4	11.1	4.6	2.4
						30	10.2	33.6	3.3	8.4
		Associative	UNSAT	8	8	6	2.8	9.4	3.4	2.5
						7	52.5	211.9	4.0	62.2
						8	230.2	891.9	3.9	309.1

required a wider range of refactoring laws than the ones whose variants were developed with clone-and-own in plain Alloy, because the original Colorful Alloy models were purposely complex and diverse in terms of variability annotations, since they were originally developed to illustrate the potential of the Colorful Alloy language.

6.4 Evaluating Colorful Analysis

Our evaluation of the colorful analysis procedures aimed to answer two questions regarding the feasibility of the approach, prior to developing more advances analysis procedures: 1) is the analysis through the amalgamated technique feasible? And if so, 2) does it outperform a preprocessing approach that iteratively

analyzes all projected variants? To answer these questions, we applied our technique to all the 10 previously presented model families with different characteristics, including some rich on structural and others on behavioral properties, and mostly encoding variants of system design. Tables 4 and 5 depict execution times for the proactive and extractive examples, respectively, with a varying scope for each command. The table presents how many features each model has (NF). Then, for each pair command/scope of a model, it presents how many variants are considered by the feature scope (NP), how many of those variants are valid according to the FM (NV), what is the result of the analysis in each executed command (SAT/UNSAT)[3], the analysis time under the amalgamated model (TA), under the iterative analysis of all projected variants (TI), and the speedup of the former in relation to the latter (SU). The slowest time for a single projected variant (SP) is also presented. All commands were run 5 times on a MacBook with a 2.4 GHz Intel Core i5 and 8GB memory using the MiniSAT solver, with the reported time being the average of those runs.

Table 5: Evaluation of the amalgamated and iterative approaches for the extractive examples.

Model	NF	Command	Analysis	NP	NV	Scope	TA(s)	TI(s)	SU	SP(s)
Grandpa	2	NoSelfFather	UNSAT	4	3	30	0.7	1.5	2.2	0.6
						35	2.0	3.9	2.0	1.6
						40	1.3	2.7	2.0	1.1
		NoSelfGrandpa	SAT	4	3	35	0.8	2.1	7.3	0.9
						30	1.6	3.8	2.4	1.8
						40	2.4	5.1	2.1	2.1
RingElection	1	AtmostOneElection	UNSAT	2	2	6	0.3	0.6	2.1	0.3
						7	1.1	2.0	1.9	1.0
						8	9.2	18.8	2.0	9.4
		AtLeastOneElection	SAT	2	2	6	0.3	0.6	2.1	0.3
						7	1.1	2.0	1.9	1.0
						8	9.2	18.8	2.0	9.4
Addressbook	2	delUndoesAdd	UNSAT	4	3	15	-	4.8	-	3.5
						18	-	11.1	-	8.6
						20	-	19.5	-	14.9
		addIdempotent	UNSAT	4	3	15	-	4.6	-	3.5
						18	-	12.2	-	8.7
						20	-	19.5	-	14.3
		addLocal	SAT	4	3	15	-	9.7	-	8.2
						18	-	33.3	-	37.9
						20	-	32.8	-	36.9
Hotel	2	NoBadEntry	SAT	4	4	10	5.1	13.9	2.7	8.0
						11	8.1	30.9	3.8	16.9
						12	14.3	48.0	3.4	28.2
						13	21.1	70.9	3.4	37.0

Results show that the amalgamated approach is indeed feasible, since it proves to be always faster than the iterative analysis. However, the amalgamated analysis of the *AddressBook* example was not possible

3 When an analysis is UNSAT (unsatisfiable) it means that every candidate solution was explored (for example, when a checked assertion is true). When it is SAT (satisfiable) it stopped because a valid instance (in a run command) or a counterexample (in a check command) was found.

due to the ordering statement on signature Book (that only exists on variant ①②). The amalgamated analysis technique does not yet support the conditional opening of modules.

A surprising result was how often the amalgamated approach is actually faster than the analysis of a single, projected variant. For *GrandpaFamily* we identified a cause related to imposing signature multiplicities through the declaration rather than through a fact: e.g., in variant ①②, having `Adam` declared without any multiplicity and enforcing **`one`** in a fact actually speeds up the analysis in comparison to directly declaring the signature with multiplicity **`one`**. Another identified issue is related to the declaration of fields that are always forced to be empty: e.g., in variant ①, having `spouse` declared but forced to be empty actually speeds up the analysis when compared to removing it. We suspect that similar situations happen in the other examples where the amalgamated analysis was also faster. To understand why these refactorings of the specification affect the underlying Kodkod analysis, and research whether they can be explored to also improve performance of the iterative technique (or Alloy in general), is left as future work.

7

CONCLUSION

In this thesis, we started by proposing an annotative approach for formal feature-oriented design that minimally extends the Alloy language and its Analyzer with colorful annotations and variability-aware analysis commands. We have shown how this language extension, named Colorful Alloy, can be used to proactively design several case studies that required not only coarse-grained annotations but also fine-grained ones, for example, annotating part of a formula or a relational expression. Previous proposals of languages for formal SPL design were either compositional, not adequate for fine-grained feature support, or focused on low-level behavioural design and analysis, making Colorful Alloy the first annotative approach targeted at lightweight high-level structural software design. For the analysis of this language extension two alternative techniques have been explored: an iterative approach that analyzes a specific variant at a time; and an amalgamated approach that analyzes multiple variants at once. A preliminary study has been conducted which shows that the amalgamated analysis is better than iterating over all variants. Although not presented in this thesis, the Colorful Alloy approach was also applied to the Electrum extension and used to pro-actively design an automotive related case-study (Cunha et al., 2020).

We also proposed a catalog of variant-preserving refactoring laws for Colorful Alloy. This catalog covers most aspects of the language, from structural elements, such as signature and field declarations, to formulas in facts and assertions, including analysis commands. Using these refactorings, we proposed a step-wise technique for migrating sets of plain Alloy clones, specifying different variants of a system, into a single Colorful Alloy SPL. We manually evaluated the effectiveness of this migration technique with several sets of plain Alloy clones and achieved a substantial reduction in the size of the equivalent Colorful Alloy model, with likely gains in terms of maintainability, understandability, and efficiency of analysis. We also implemented an automatic merging strategy that composes a sequence refactorings steps, and that can be used to perform clone migration in a single step. This automatic strategy was evaluated against the best result obtained manually applying the refactoring laws and achieved almost the same reduction in size for all our examples.

As future work, we are going to extend support for additional operators and elements for Colorful Alloy. At present, except for a few exceptions, we can only annotate optional elements of the AST. Many elements that are not optional, such as quantifiers or unary operators, cannot be marked, which may require the introduction of an entirely new constraint in a particular variant, even if there is little difference to an already existing one. Therefore, we intend to support the markup of these elements to make the use of our Colorful Alloy more convenient. We also want to explore whether specific syntactic constructs would help specify the FM. Currently, FMs must be specified with normal Colorful Alloy syntax, for instance through annotated **some none** expression. In the future we intend to add a small DSL to specify the FM, and thus simplify the process of extracting the respective constraints.

Regarding the analysis processes, we plan to continue exploring the issues that caused the amalgamated analysis to be faster than the projected one, even for a single variant, and whether they can be exploited to improve the efficiency of the latter. We also intend to improve the scenario exploration capabilities of the Colorful Alloy Analyzer, namely allow the user to control the search for instances or counter-examples in specific variants, for example, adding a feature selector to enable the user to quickly focus on a variant. We also intend to explore the usage of model merging techniques or variational SAT solving (Young et al., 2020) to be able to present the user with a single, feature annotated, counter-example instance that succinctly captures which features are causing errors.

Concerning the clone migration technique, we intend to conduct a more extensive evaluation, with more examples and measuring other aspects of model quality, in order to assess if the positive results achieved in the preliminary evaluation still hold. We also intend to implement a full SAT-based decision procedure for testing the preconditions of the refactoring laws.

BIBLIOGRAPHY

Andreas Abele, Yiannis Papadopoulos, David Servat, Martin Törngren, and Matthias Weber. The CVM framework - A prototype tool for compositional variability management. In *Proceedings of the 4th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS'10)*, volume 37, pages 101–105. Universität Duisburg-Essen, 2010.

Michal Antkiewicz and Krzysztof Czarnecki. Featureplugin: feature modeling plug-in for eclipse. In *Proceedings of the 2004 OOPSLA workshop on Eclipse Technology eXchange (ETX'04)*, pages 67–72. ACM, 2004.

Sven Apel and Christian Kästner. An overview of feature-oriented software development. *Journal of Object Technology*, 8:49–84, 2009.

Sven Apel, Wolfgang Scholz, Christian Lengauer, and Christian Kastner. Detecting dependences and interactions in feature-oriented design. In *Proceedings of the IEEE 21st International Symposium on Software Reliability Engineering (ISSRE'10)*, pages 161–170. IEEE Computer Society, 2010.

Sven Apel, Christian Kästner, and Christian Lengauer. Language-independent and automated software composition: The FeatureHouse experience. *IEEE Transactions on Software Engineering*, 39(1):63–79, 2013.

Wesley KG Assunção, Roberto E Lopez-Herrejon, Lukas Linsbauer, Silvia R Vergilio, and Alexander Egyed. Reengineering legacy applications into software product lines: a systematic mapping. *Empirical Software Engineering*, 22:2972–3016, 2017.

Wesley KG Assunção, Silvia R Vergilio, and Roberto E Lopez-Herrejon. Automatic extraction of product line architecture and feature models from UML class diagram variants. *Information and Software Technology*, 117:106198, 2020.

Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT press, 2008.

Kacper Bak, Zinovy Diskin, Michal Antkiewicz, Krzysztof Czarnecki, and Andrzej Wasowski. Clafer: unifying class and feature modeling. *Software & Systems Modeling*, 15:811–845, 2016.

Don Batory. Feature models, grammars, and propositional formulas. In *Proceedings of the 9th International Conference on Software Product Lines (SPLC'05)*, volume 3714, pages 7–20. Springer, 2005.

Don S. Batory and Egon Börger. Modularizing theorems for software product lines: The Jbook case study. *Journal of Universal Computer Science*, 14:2059–2082, 2008.

Thomas Bednasch. Konzept und implementierung eines konfigurierbaren metamodells für die merkmalmodellierung. Master's thesis, Fachhochschule Kaiserslautern, Standort Zweibrücken, 2002.

Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on software engineering*, 33:577–591, 2007.

David Benavides, Pablo Trinidad, and Antonio Ruiz Cortés. Using constraint programming to reason on feature models. In *Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering (SEKE'05)*, volume 5, pages 677–682, 2005a.

David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. Automated reasoning on feature models. In *Proceedings of the 17th International Conference on Advanced Information Systems Engineering (CAiSE'05)*, volume 3520, pages 491–503. Springer, 2005b.

David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz Cortés. Fama: Tooling a framework for the automated analysis of feature models. In *the 1st International Workshop on Variability Modelling of Software-intensive Systems (VAMOS'07)*, volume 2007-01, pages 129–134, 2007.

David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information systems*, 35:615–636, 2010.

Fabian Benduhn, Thomas Thüm, Malte Lochau, Thomas Leich, and Gunter Saake. A survey on modeling techniques for formal behavioral verification of software product lines. In *Proceedings of the 9th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'15)*, page 80. ACM, 2015.

Julien Brunel, David Chemouil, Alcino Cunha, and Nuno Macedo. The Electrum analyzer: Model checking relational first-order temporal specifications. In *33rd ACM/IEEE International Conference on Automated Software Engineering (ASE'18)*, page 884–887. ACM, 2018.

Muffy Calder and Alice Miller. Feature interaction detection by pairwise analysis of LTL properties - A case study. *Formal Methods in System Design*, 28:213–261, 2006.

Felix Sheng-Ho Chang and Daniel Jackson. Symbolic model checking of declarative relational models. In *Proceedings of the 28th International Conference on Software Engineering (ICSE'06)*, page 312–320. ACM, 2006.

Philipp Chrszon, Clemens Dubslaff, Sascha Klüppelholz, and Christel Baier. Profeat: feature-oriented engineering for family-based probabilistic model checking. *Formal Aspects of Computing*, 30:45–75, 2018.

Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. NuSMV: A new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2:410–425, 2000.

Dave Clarke, Radu Muschevici, José Proença, Ina Schaefer, and Rudolf Schlatte. Variability modelling in the ABS language. In *Proceedings of the 9th International Symposium on Formal Methods for Components and Objects (FMCO'10)*, volume 6957, pages 204–224. Springer, 2010.

A. Classen, M. Cordy, P. Y. Schobbens, P. Heymans, A. Legay, and J. F. Raskin. Featured transition systems: Foundations for verifying variability-intensive systems and their application to LTL model checking. *IEEE Transactions on Software Engineering*, 39:1069–1089, 2013.

Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay, and Jean-François Raskin. Model checking lots of systems: efficient verification of temporal properties in software product lines. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1 (ICSE'10)*, pages 335–344. ACM, 2010.

Andreas Classen, Quentin Boucher, and Patrick Heymans. A text-based approach to feature modelling: Syntax and semantics of TVL. *Science of Computer Programming*, 76:1130 – 1143, 2011a.

Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. Symbolic model checking of software product lines. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*, pages 321–330. ACM, 2011b.

Andreas Classen, Maxime Cordy, Patrick Heymans, Axel Legay, and Pierre-Yves Schobbens. Model checking software product lines with SNIP. *International Journal on Software Tools for Technology Transfer*, 14: 589–612, 2012.

Andreas Classen, Maxime Cordy, Patrick Heymans, Axel Legay, and Pierre-Yves Schobbens. Formal semantics, modular specification, and symbolic verification of product-line behaviour. *Science of Computer Programming*, 80:416 – 439, 2014.

Matthias Clauß. Generic modeling using UML extensions for variability. In *Proceedings of the Workshop on Domain Specific Visual Languages, Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'01)*, volume 2001, 2001.

Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, and Axel Legay. Behavioural modelling and verification of real-time software product lines. In *Proceedings of the 16th International Software Product Line Conference (SPLC'12)*, pages 66–75. ACM, 2012.

Maxime Cordy, Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. Provelines: a product line of verifiers for software product lines. In *Proceedings of the 17th International Software Product Line Conference co-located workshops (SPLC'13 workshops)*, pages 141–146. ACM, 2013.

Rui Couto, José Creissac Campos, Nuno Macedo, and Alcino Cunha. Improving the visualization of Alloy instances. In *Proceedings of the 4th Workshop on Formal Integrated Development Environment (F-IDE'18)*, volume 284 of *EPTCS*, pages 37–52, 2018.

Alcino Cunha. Bounded model checking of temporal formulas with Alloy. In *Proceedings of the 4th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z (ABZ'14)*, volume 8477, page 303–308. Springer, 2014.

Alcino Cunha, Nuno Macedo, and Chong Liu. Validating multiple variants of an automotive light system with Electrum. In *Proceedings of the 7th International Conference on Rigorous State-Based Methods*, pages 318–334. Springer, 2020.

Krzysztof Czarnecki and Michal Antkiewicz. Mapping features to models: A template approach based on superimposed variants. In *Proceedings of the 4th International Conference on Generative Programming and Component Engineering (GPCE'05)*, volume 3676, pages 422–437. Springer, 2005.

Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative programming - methods, tools and applications*. Addison-Wesley, 2000.

Krzysztof Czarnecki and Krzysztof Pietroszek. Verifying feature-based model templates against well-formedness OCL constraints. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE'06)*, page 211–220. ACM, 2006.

Krzysztof Czarnecki and Andrzej Wasowski. Feature diagrams and logics: There and back again. In *Proceedings of the 11th International Software Product Line Conference (SPLC'07)*, pages 23–34. IEEE Computer Society, 2007.

Krzysztof Czarnecki, Thomas Bednasch, Peter Unger, and Ulrich W. Eisenecker. Generative programming for embedded software: An industrial experience report. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE'2002)*, volume 2487, pages 156–172. Springer, 2002.

Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10:7–29, 2005.

Xavier Devroey, Maxime Cordy, Gilles Perrouin, Eun-Young Kang, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay, and Benoit Baudry. A vision for behavioural model-driven validation of software product lines. In *Proceedings of the 5th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'12)*, volume 7609, pages 208–222. Springer, 2012.

Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. An exploratory study of cloning in industrial software product lines. In *Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR'13)*, pages 25–34. IEEE Computer Society, 2013.

Jonathan Edwards, Daniel Jackson, and Emina Torlak. A type system for object models. In *Proceedings of the 12th International Symposium on Foundations of Software Engineering (SIGSOFT'04/FSE-12)*, volume 29, page 189–199. ACM, 2004.

Holger Eichelberger and Klaus Schmid. A systematic analysis of textual variability modeling languages. In *Proceedings of the 17th International Software Product Line Conference (SPLC'13)*, pages 12–21. ACM, 2013.

Alessandro Fantechi and Stefania Gnesi. Formal modeling for product families engineering. In *12th International Software Product Line Conference (SPLC'08)*, pages 193–202. IEEE, 2008.

Janet Feigenspan, Christian Kästner, Sven Apel, Jörg Liebig, Michael Schulze, Raimund Dachselt, Maria Papendieck, Thomas Leich, and Gunter Saake. Do background colors improve program comprehension in the# ifdef hell? *Empirical Software Engineering*, 18:699–745, 2013.

Wolfram Fenske, Jens Meinicke, Sandro Schulze, Steffen Schulze, and Gunter Saake. Variant-preserving refactorings for migrating cloned products to a product line. In *Proceedings of the 24th International Conference on Software Analysis, Evolution and Reengineering (SANER'17)*, pages 316–326. IEEE Computer Society, 2017.

Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

Marcelo F. Frias, Juan P. Galeotti, Carlos López Pombo, and Nazareno Aguirre. DynAlloy: upgrading Alloy with actions. In *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*, pages 442–450. ACM, 2005.

Marcelo F. Frias, Carlos G. Lopez Pombo, Juan P. Galeotti, and Nazareno M. Aguirre. Efficient analysis of DynAlloy specifications. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17: 1–34, 2007.

Rohit Gheyi. *A Refinement Theory for Alloy*. PhD thesis, Universidade Federal de Pernambuco, 2007.

Rohit Gheyi and Paulo Borba. Refactoring Alloy specifications. *Electronic Notes in Theoretical Computer Science*, 95:227–243, 2004.

Rohit Gheyi, Tiago Massoni, and Paulo Borba. A theory for feature models in Alloy. In *1st Alloy workshop*, pages 71–80. Citeseer, 2006.

Stefania Gnesi and Marinella Petrocchi. Towards an executable algebra for product lines. In *Proceedings of the 16th International Software Product Line Conference (SPLC'12)*, pages 66–73. ACM, 2012.

Hassan Gomaa. Object oriented analysis and modeling for families of systems with UML. In *Proceedings of the 6th International Conference on Software Reuse (ICSR-6)*, volume 1844, pages 89–99. Springer, 2000.

Ali Gondal, Mike Poppleton, Michael Butler, and Colin Snook. Feature-oriented modelling using Event-B. In *Proceedings of the International Conference on Software Engineering Theory and Practice (SETP'10)*, pages 100–106, 2010.

Joel Greenyer, Amir Molzam Sharifloo, Maxime Cordy, and Patrick Heymans. Efficient consistency checking of scenario-based product-line specifications. In *Proceedings of the 20th IEEE International Requirements Engineering Conference (RE'12)*, pages 161–170. IEEE Computer Society, 2012.

Martin L. Griss, John M. Favaro, and Massimo D'Alessandro. Integrating feature modeling with the rseb. In *Proceedings of the 5th International Conference on Software Reuse (ICSR'1998)*, pages 76–85. IEEE Computer Society, 1998.

Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta modeling for software architectures. In *Proceedings of the Dagstuhl Workshop on Model-Based Development of Embedded Systems (MBEES'11)*, pages 1–10. fortiss GmbH, München, 2011.

Adithya Hemakumar. Finding contradictions in feature models. In *Proceedings of the International Workshop on Analyses of Software Product Lines (ASPL'08)*, pages 183–190. Lero Int. Science Centre, University of Limerick, Ireland, 2008.

Gerard Holzmann. *Spin Model Checker, the: Primer and Reference Manual*. Addison-Wesley Professional, first edition, 2003. ISBN 0-321-22862-6.

Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23:279–295, 1997.

Daniel Jackson. *Software Abstractions: logic, language, and analysis*. MIT press, 2012.

Jean-Marc Jézéquel. Model-driven engineering for software product lines. *International Scholarly Research Notices*, 2012:1–24, 2012.

Edson A. Oliveira Junior, Itana Maria de Souza Gimenes, and José Carlos Maldonado. Systematic management of variability in UML-based software product lines. *Journal of Universal Computer Science*, 16(17):2374–2393, 2010.

Paulius Juodisius, Atrisha Sarkar, Raghava Rao Mukkamala, Michal Antkiewicz, Krzysztof Czarnecki, and Andrzej Wasowski. Clafer: Lightweight modeling of structure, behaviour, and variability. *The Art, Science, and Engineering of Programming*, 3:2, 2018.

Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, 1990.

Christian Kästner and Sven Apel. Integrating compositional and annotative approaches for product line engineering. In *Proceedings of the Workshop on Modularization, Composition, and Generative Techniques for Product Line Engineering (McGPLE'08)*, pages 35–40, 2008.

Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in software product lines. In *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, page 311–320. ACM, 2008.

Charles W. Krueger. Easing the transition to software mass customization. In *Proceedings of the 4th International Workshop on Software Product-Family Engineering (PFE'01)*, pages 282–293. Springer-Verlag, 2001.

Duc Minh Le, Hyesun Lee, Kyo Chul Kang, and Lee Keun. Validating consistency between a feature model and its implementation. In *Proceedings of the international Conference on Software Reuse (ICSR'13)*, pages 1–16. Springer, 2013.

Chong Liu, Nuno Macedo, and Alcino Cunha. Simplifying the analysis of software design variants with a Colorful Alloy. In *Proceedings of the International Symposium on Dependable Software Engineering: Theories, Tools, and Applications (SETTA'19)*, pages 38–55. Springer, 2019.

Chong Liu, Nuno Macedo, and Alcino Cunha. Merging cloned Alloy models with colorful refactorings. In *Proceedings of the 23rd Brazilian Symposium on Formal Methods (SBMF'20)*, pages 173–191. Springer, 2020.

Malte Lochau, Stephan Mennicke, Hauke Baller, and Lars Ribbeck. Deltaccs: A core calculus for behavioral change. In *Proceedings of the 6th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'14)*, volume 8802, pages 320–335. Springer, 2014.

Nuno Macedo, Alcino Cunha, and Tiago Guimarães. Exploring scenario exploration. In *Proceedings of the 18th International Conference on Fundamental Approaches to Software Engineering (FASE'15)*, volume 9033, pages 301–315. Springer, 2015.

Nuno Macedo, Julien Brunel, David Chemouil, Alcino Cunha, and Denis Kuperberg. Lightweight specification and analysis of dynamic systems with rich configurations. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'16)*, pages 373–383. ACM, 2016.

Nuno Macedo, Alcino Cunha, José Pereira, Renato Carvalho, Ricardo Silva, Ana C. R. Paiva, Miguel S. Ramalho, and Daniel Castro Silva. Sharing and learning Alloy on the web. *CoRR*, abs/1907.02275, 2019.

Kenneth L. McMillan. *Symbolic model checking*. Kluwer, 1993. ISBN 978-0-7923-9380-1.

Marcílio Mendonça, Moises Branco, and Donald D. Cowan. S.P.L.O.T.: software product lines online tools. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications (OOPSLA'09)*, pages 761–762. ACM, 2009a.

Marcílio Mendonça, Andrzej Wasowski, and Krzysztof Czarnecki. Sat-based analysis of feature models is easy. In *Proceedings of the 13th International Software Product Line Conference (SPLC'09)*, volume 446, pages 231–240. ACM, 2009b.

Aleksandar Milicevic and Daniel Jackson. Preventing arithmetic overflows in Alloy. *Science of Computer Programming*, 94:203–216, 2014.

Joseph P. Near and Daniel Jackson. An imperative extension to Alloy. In *Proceedings of the International Conference on Abstract State Machines, Alloy, B and Z (ABZ'10)*, volume 5977, pages 118–131. Springer, 2010.

Tim Nelson, Salman Saghafi, Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. Aluminum: Principled scenario exploration through minimality. In *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)*, page 232–241. IEEE, 2013.

Natsuko Noda and Tomoji Kishi. Aspect-oriented modeling for variability management. In *Proceedings of the 12th International Conference on Software Product Lines (SPLC'08)*, pages 213–222. IEEE Computer Society, 2008.

Malte Plath and Mark Ryan. Feature integration using a feature construct. *Science of Computer Programming*, 41:53–84, 2001.

Klaus Pohl, Günter Böckle, and Frank J van Der Linden. *Software product line engineering: foundations, principles and techniques*. Springer Science & Business Media, 2005.

Hendrik Post and Carsten Sinz. Configuration lifting: Verification meets software configuration. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE'08)*, pages 347–350. IEEE, 2008.

Christian Prehofer. Feature-oriented programming: A fresh look at objects. In *European Conference on Object-Oriented Programming*, pages 419–443. Springer, 1997.

Iris Reinhartz-Berger and Arnon Sturm. Comprehensibility of UML-based software product line specifications - A controlled experiment. *Empirical Software Engineering*, 19(3):678–713, 2014.

Matthias Riebisch, Kai Böllert, Detlef Streitferdt, and Ilka Philippow. Extending feature diagrams with uml multiplicities. In *Proceedings of the 6th Conference on Integrated Design & Process Technology (IDPT'02)*, volume 23, pages 1–7, 2002.

Shamim Ripon, Keya Azad, Sk. Jahir Hossain, and Mehidee Hassan. Modeling and analysis of product-line variants. In *Proceedings of the 16th International Software Product Line Conference (SPLC'12)*, pages 26–31. ACM, 2012.

Silva Robak, Bogdan Franczyk, and Kamil Politowicz. Extending the UML for modeling variabilities for system families. *International Journal of Applied Mathematics and Computer Science*, 12:285–298, 2002.

Julia Rubin and Marsha Chechik. Combining related products into product lines. In *Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering (FASE 2012)*, volume 7212, pages 285–300. Springer, 2012.

Julia Rubin and Marsha Chechik. Quality of merge-refactorings for product lines. In *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering (FASE 2013)*, volume 7793, pages 83–98. Springer, 2013.

Julia Rubin, Krzysztof Czarnecki, and Marsha Chechik. Cloned product variants: From ad-hoc to managed software product lines. *International Journal on Software Tools for Technology Transfer*, 17:627–646, 2015.

Hamideh Sabouri and Ramtin Khosravi. Delta modeling and model checking of product families. In *Proceedings of the 5th International Conference on Fundamentals of Software Engineering (FSEN'13)*, volume 8161, pages 51–65. Springer, 2013.

Ina Schaefer. Variability modelling for model-driven development of software product lines. In *Proceedings of the 4th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS'10)*, volume 10, pages 85–92. Universität Duisburg-Essen, 2010.

Ina Schaefer and Reiner Hähnle. Formal methods in software product line engineering. *IEEE Computer*, 44(2):82–85, 2011.

Sandro Schulze, Thomas Thüm, Martin Kuhlemann, and Gunter Saake. Variant-preserving refactoring in feature-oriented software product lines. In *Proceedings of the 6th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS'12)*, pages 73–81. ACM, 2012.

Sergio Segura. Automated analysis of feature models using atomic sets. In *1st Workshop on Analyses of Software Product Lines (SPLC'08)*, pages 201–207, 2008.

Samuel Sepúlveda, Carlos Cares, and Cristina Cachero. Towards a unified feature metamodel: A systematic comparison of feature languages. In *Proceedings of the 7th Iberian Conference on Information Systems and Technologies (CISTI'12)*, pages 1–7, 2012.

José Serna, Nancy A. Day, and Sabria Farheen. DASH: A new language for declarative behavioural requirements with control state hierarchy. In *Proceedings of the 25th International Requirements Engineering Conference Workshops (REW'17)*, pages 64–68. IEEE, 2017.

Pourya Shaker, Joanne M. Atlee, and Shige Wang. A feature-oriented requirements modelling language. In *Proceedings of the 20th IEEE International Requirements Engineering Conference (RE'12)*, pages 151–160. IEEE Computer Society, 2012.

Shin'ichi Shiraishi. An AADL-Based approach to variability modeling of automotive control systems. In *Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems (MODELS'10)*, page 346–360. Springer-Verlag, 2010.

Anjali Sree-Kumar, Elena Planas, and Robert Clarisó. Analysis of feature models using Alloy: A survey. In *Proceedings of the 7th International Workshop on Formal Methods and Analysis in Software Product Line Engineering (FMSPLE'16)*, volume 206, pages 46–60, 2016.

Harald Störrle. Towards clone detection in UML domain models. *Software & Systems Modeling*, 12: 307–329, 2013.

Chang-ai Sun, Rowan Rossing, Marco Sinnema, Pavel Bulanov, and Marco Aiello. Modeling and managing the variability of web service-based systems. *Journal of Systems and Software*, 83:502–516, 2010.

Maurice H. ter Beek and Erik P. de Vink. Towards modular verification of software product lines with mcrl2. In *Proceedings of the 6th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'14)*, volume 8802 of *LNCS*, pages 368–385. Springer, 2014a.

Maurice H. ter Beek and Erik P. de Vink. Using mcrl2 for the analysis of software product lines. In *Proceedings of the 2nd FME Workshop on Formal Methods in Software Engineering (FormaliSE'14)*, pages 31–37. ACM, 2014b.

Maurice H. ter Beek, Alberto Lluch-Lafuente, and Marinella Petrocchi. Combining declarative and procedural views in the specification and analysis of product families. In *Proceedings of the 17th International Software Product Line Conference co-located workshops (SPLC'13 workshops)*, pages 10–17. ACM, 2013.

Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. Featureide: An extensible framework for feature-oriented software development. *Science of Computer Programming*, 79:70–85, 2014.

Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07)*, pages 632–647. Springer, 2007.

Arie Van Deursen and Paul Klint. Domain-specific language design requires feature descriptions. *Journal of computing and information technology*, 10:1–17, 2002.

Hai H Wang, Yuan Fang Li, Jing Sun, Hongyu Zhang, and Jeff Pan. Verifying feature models using OWL. *Journal of web semantics*, 5:117–129, 2007.

Jeffrey M. Young, Eric Walkingshaw, and Thomas Thüm. Variational satisfiability solving. In *Proceedings of the 24th ACM Conference on Systems and Software Product Line (SPLC '20)*, pages 18:1–18:12. ACM, 2020.

Atulan Zaman, Iman Kazerani, Medha Patki, Bhargava Guntoori, and Derek Rayside. Improved visualization of relational logic models. Technical Report CS-2013-04, University of Waterloo, 2013.

COLORFUL EXAMPLES

```
1  module Ecommerce
2  fact FeatureModel {
3    2 1 some none 1 2    -- Hierarchical requires Categories
4    3 1 some none 1 3    -- Multiple requires Categories
5  }
6  sig Product {
7   images      : set Image,
8   1 catalog    : one Catalog 1 ,
9   1 3 category : some Category 3 1 ,
10  1 3 category : one Category 3 1  }
11
12 5 sig onSale in Product {} 5
13
14 sig Image {}
15
16 sig Catalog {
17   4 thumbnails: set Image 4  }
18
19 1 2 sig Category {
20  inside : one Catalog+ Category
21 } 2 1
22 1 2 sig Category {
23  inside : one Catalog
24 } 2 1
25
26 1 2 fact Acyclic {
27  all c : Category | c not in c.^inside
28 } 2 1
29 4 fact Thumbnails {
30   all c : Catalog |
31       c.thumbnails in ( 1 category.(( 2 inside 2  + 2 ^inside 2 ).c) 1  + 1 catalog.c 1 ).images
32  } 4
33 4 5 fact OnSaleThumbnails {
34   all p: onSale, c: catalogs[p]| some p.images & c.thumbnails
35 } 5 4
36
37 pred Scenario{
38   some Product.images and 1 all c : Category | lone category.c 1
39 }
40 run Scenario with exactly 1
41
42 fun catalogs [p : Product] : set Catalog {
43  1 p.catalog 1  +  1 p.category.( 2 inside 2  +  2 ^inside 2 ) &  2 Catalog 2 1
44 }
45 assert AllCataloged {
46   all p : Product | some catalogs[p]
47 }
48 check AllCataloged for 10
```

Figure 92: E-commerce specification in Colorful Alloy.

```
1   module Graph
2   /*
3   * ① Multigraph
4   * ② Undirected
5   * ③ DAG
6   * ④ Tree
7   * ⑤ Vertex Labeled
8   * ⑥ Binary Tree
9   */
10
11  fact FeatureModel {
12   -- DAG incompatible with Undirected
13   ②③some none③②
14   -- Tree requires DAG
15   ④③some none③④
16   -- Binary Tree requires Tree
17   ⑥④some none④⑥
18   -- Binary Tree requires Vertex Labeled
19   ⑥⑤some none⑤⑥
20  }
21
22  sig Node {
23   adj : set Node,
24   ⑤label : one Label⑤
25  }
26  ④lone sig Root extends Node {}④
27
28  ⑤sig Label {}⑤
29
30  sig Edge {
31   src, dst : one Node
32  }
33  ⑥sig Left, Right extends Edge {}⑥
34
35  fact {
36    -- Auxiliary relation to help with visualization and specification
37    adj = ~src.dst
38    -- No multiple edges
39    ①all disj e, e' Edge | e.src != e'.src or e.dst != e'.dst①
40    -- The adjency relation is symetric
41    ②adj = ~adj②
42    -- No cycles
43    ③all n : Node | n not in n.^adj③
44    -- All nodes except the root have one parent
45    ④all n : Node-Root | one adj.n④
46    -- Labels are unique
47    ⑤label in Node lone -> Label⑤
48    -- In binary trees all edges are either left or right
49    ⑥Edge = Left+Right⑥
50    -- Each node has at most one left and one right adjacent
51    ⑥all n : Node | lone (src.n & Left).dst and lone (src.n & Right).dst⑥
52    -- The left and right adjacent nodes must be distinct
53    ⑥all n : Node | no (src.n & Left).dst & (src.n & Right).dst⑥
54  }
55  run {} with exactly ④ for 5 expect 0
56
57  ④assert Connected {
58    -- All nodes descend from root
59    Node in Root.*adj
60  }④
61  check Connected with exactly ③,④ for 8
62
63  ③assert SourcesAndSinks {
64    -- If the graph is not empty there is at least one source and one sink node
65    some Node implies (some n : Node | no adj.n and some n : Node | no n.adj)
66  }③
67  check SourcesAndSinks with ③ for 7
```

Figure 93: Graph specification in Colorful Alloy.

```
1   module vending
2   /*
3   * (1) Free drinks
4   * (2) Multiple Selection
5   * (3) Cancel
6   * (4) Multiple Prices
7   */
8
9   open util/ordering[Time]
10  open util/natural as nat
11
12  fact FeatureModel {
13   -- Free drinks incompatible with Multiple Selection
14  some none
15   -- Free drinks incompatible with Multiple Prices
16  some none
17  }
18
19  abstract sig State {}
20  one sig Ready,Selected,Served extends State {}
21  one sig Paid extends State {}
22  one sig Done extends State {}
23
24  sig Product {
25   price : one  Natural
26  }
27
28  sig Time {
29   state     : one State,
30   stock     : set Product,
31   selection : set Product,
32   total     : one Natural,
33   coins     : one Natural,
34   balance : one Natural
35  }
36
37  pred Pay [pre, pos : Time] {
38    pre.state in Ready+Paid
39    pos.state = Paid
40    pos.stock = pre.stock
41    pos.selection = pre.selection
42    pos.total = pre.total
43    pos.coins = inc[pre.coins]
44    pos.balance = pre.balance
45  }
46
47  pred Select [pre,pos : Time, p : Product] {
48    pre.state in Ready+Paid +Selected
49    p in pre.stock - pre.selection
50    pos.total= inc[pre.total]
51    pos.total= add[pre.total, p.price]
52    pos.state = Selected
53    pos.stock = pre.stock
54    pos.selection = pre.selection + p
55    pos.coins = pre.coins
56    pos.balance = pre.balance
57  }
58
59  pred Cancel [pre,pos : Time] {
60    pre.state = Selected
61    pos.state = Done+ Ready
62    pos.stock = pre.stock
63    no pos.selection
64    pos.total = Zero
65    pos.coins = pre.coins
66    pos.balance = pre.balance
67  }
```

Figure 94: Vending Machine specification in Colorful Alloy (part 1).

```
1   pred Serve [pre,pos : Time] {
2     pre.state = Selected
3     pos.state = Served
4     ①gte[pre.coins, pre.total]①
5     pos.stock = pre.stock - pre.selection
6     pos.selection = pre.selection
7     ①pos.total = pre.total①
8     ①pos.coins = sub[pre.coins, pre.total]①
9     ①pos.balance = add[pre.balance, pre.total]①
10  }
11
12  pred Open [pre,pos : Time] {
13    pre.state = Served
14    pos.state = ①Done① +①Ready①
15    pos.stock = pre.stock
16    no pos.selection
17    ①pos.total = Zero①
18    ①pos.coins = pre.coins①
19    ①pos.balance = pre.balance①
20  }
21
22  ①pred Change [pre, pos : Time] {
23    pre.state = Done
24    pos.state = Ready
25    pos.stock = pre.stock
26    pos.selection = pre.selection
27    pos.total = pre.total
28    pos.coins = Zero
29    pos.balance = pre.balance
30  }①
31
32  pred Nop [pre,pos : Time] {
33    pre.state = Ready
34    no pre.stock
35    pos.state = pre.state
36    pos.stock = pre.stock
37    pos.selection = pre.selection
38    ①pos.total = pre.total①
39    ①pos.coins = pre.coins①
40    ①pos.balance = pre.balance①
41  }
42
43  fact Behaviour {
44    first.state = Ready
45    no first.selection
46    some first.stock
47    ①first.coins = Zero①
48    ①first.balance = Zero①
49    ①first.total = Zero①
50    all pre : Time-last, pos : pre.next {
51        ①Pay[pre, pos]① or (some p : Product | Select[pre,pos,p]) or ③Cancel[pre,pos]③ or
52        Serve[pre,pos] or Open[pre,pos] or ①Change[pre, pos]① or Nop[pre,pos]
53      }
54  }
55
56  ①assert Balance {
57    all t : Time | t.state = Served implies gte[t.balance, Zero]
58  }①
59  check Balance with exactly ① for  5 but 20 Time
60
61  assert Selection {
62    all t : Time, p : Product | p not in t.stock implies all u : t.nexts |
63      p not in u.selection
64  }
65  check Selection with exactly ① for 5 but 20 Time
66
67  pred NoStock {
68    some t : Time | no t.stock
69  }
70  run NoStock for 5 but 10 Time
```

Figure 94: Vending Machine specification in Colorful Alloy (part 2).

```
module Bestiary
/*
* ① Injective
* ② Functional
* ③ Total
* ④ Surjective
*/

sig A {
  r : set B
}

sig B {}

fact Bestiary {
  ①r in A lone -> B①
  ②r in A -> lone B②
  ③r in A -> some B③
  ④r in A some -> B④
}

assert Injective {
  r.~r in iden
}

check Injective with ① for 25

assert Simple {
  ~r.r in iden
}

check Simple with ② for 25

assert Associative {
  r.(~r.r) = (r.~r).r
}

check Associative for 6
```

Figure 95: Bestiary specification in Colorful Alloy.

```
1  fact FeatureModel {
2  -- ② requires ①
3  ②①some none①②
4  }
5
6  abstract sig Person {
7    father: lone Man,
8    mother: lone Woman
9  }
10
11 sig Man extends Person {
12   wife: lone Woman
13 }
14 sig Woman extends Person {
15   husband: lone Man
16 }
17
18 ②fact {
19   no p: Person | p in p.^(mother+father)
20   wife = ~husband
21 }②
22 ①②fact Biology { no p: Person | p in p.^(mother+father) }②①
23 ①②fact Terminology { wife = ~husband }②①
24
25 assert NoSelfFather {
26   no m: Man | m = m.father
27 }
28 check NoSelfFather
29
30 fun grandpas [p: Person] : set Person {
31   ①②p.(mother+father).father②① +
32   ①let parent = mother + father + father.wife + mother.husband |
33        p.parent.parent & Man①
34 }
35
36 pred ownGrandpa [p: Person] {
37   p in p.grandpas
38 }
39 run ownGrandpa for 4 Person
40
41 ①②assert NoSelfGrandpa { no p: Person | p in p.grandpas }②①
42 check NoSelfGrandpa with exactly ①,② for 4 Person
43
44 ①②fact SocialConvention { no (wife+husband) & ^(mother+father) }②①
45 ①②pred SocialConvention1 { no (wife + husband) & ^(mother + father) }②①
46 ①②pred SocialConvention2 {
47   let parent = mother + father {
48     no m: Man | some m.wife and m.wife in m.*parent.mother
49     no w: Woman | some w.husband and w.husband in w.*parent.father }
50 }②①
51
52 ①②assert Same { SocialConvention1 iff SocialConvention2 }②①
53 check Same with exactly ①,②
```

Figure 96: OwnGrandPa specification in Colorful Alloy.

```
module ringElection
open util/ordering[Time]
open util/ordering[Process]

sig Time {}
sig Process {
  succ: Process,
  toSend: Process -> Time,
  elected: set Time
}

fact ring {
  all p: Process | Process in p.^succ
}

pred init [t: Time] {
  all p: Process | p.toSend.t = p
}

pred step [t, t': Time, p: Process] {
  let from = p.toSend, to = p.succ.toSend |
   some id: from.t {
    from.t' = from.t - id
    to.t' = to.t + (id - p.succ.prevs)
  }
}

fact defineElected {
  no elected.first
  all t: Time-first | elected.t = { p: Process | p in p.toSend.t - p.toSend.(t.prev) }
}

fact traces {
  init [first]
  all t: Time-last |
   let t' = t.next |
    all p: Process |
     step [t, t', p] or step [t, t', succ.p] or skip [t, t', p]
}

pred skip [t, t': Time, p: Process] {
  p.toSend.t = p.toSend.t'
}

pred show { some elected }
run show for 3 Process, 4 Time

assert AtMostOneElected { lone elected.Time }
check AtMostOneElected for 3 Process, 7 Time

pred progress {
  all t: Time - last | let t' = next [t] |
   some Process.toSend.t => some p: Process | not skip [t, t', p]
}

assert AtLeastOneElected {
  progress => some elected.Time
  some t: Time | some elected.t
}
check AtLeastOneElected for 3 Process, 7 Time

pred looplessPath { no disj t, t': Time | toSend.t = toSend.t' }
run looplessPath with for 3 Process, 12 Time
run looplessPath with for 3 Process, 13 Time
```

Figure 97: Ring Election specification in Colorful Alloy.

```
1  module tour/addressBook
2  open util/ordering [Book] as BookOrder
3  fact FeatureModel {
4   -- 2 requires 1
5   some none
6  }
7  sig Name {}
8  sig Addr {}
9  abstract sig Target {}
10 sig Addr extends Target {}
11 abstract sig Name extends Target {}
12 sig Alias extends Name {}
13 sig Group extends Name {}
14 sig Book {
15  addr: Name->lone Addr,
16  names: set Name,
17  addr: names->some Target
18 }
19 fact sigBook{
20  all b: Book |no n: Name | n in n.^(b.addr)
21  all b: Book, a: Alias | lone a.(b.addr)
22 }
23
24 pred add [b, b': Book, n: Name, a: Addr] { b'.addr = b.addr + n->a }
25 pred del [b, b': Book, n: Name] { b'.addr = b.addr - n->Addrn }
26 pred add [b, b': Book, n: Name, t: Target] { b'.addr = b.addr + n->t }
27 pred del [b, b': Book, n: Name, t: Target] { b'.addr = b.addr - n->t }
28
29 fun lookup [b: Book, n: Name] : set Addr { n.(b.addr) }
30 fun lookup [b: Book, n: Name] : set Addr { n.^(b.addr) & Addr }
31
32 pred show [b: Book] { #b.addr > 1 #Name.(b.addr) > 1 }
33 run show with 1,2 for 3 but 1 Book
34
35 pred init [b: Book] { no b.addr }
36 fact traces {
37   init [first]
38   all b: Book-last |
39    let b' = b.next | some n: Name, t: Target |
40     add [b, b', n, t] or del [b, b', n, t]
41 }
42
43 pred showAdd [b, b': Book, n: Name, a: Addr] {
44   add [b, b', n, a] #Name.(b'.addr) > 1
45 }
46 run showAdd with 1,2 for 3 but 2 Book
47
48 assert delUndoesAdd {
49  all b, b', b'': Book, n: Name, a: Addr |
50   no n.(b.addr) and add [b, b', n, a] and del [b', b'', n] implies b.addr = b''.addr
51  all b, b', b'': Book, n: Name, t: Target |
52   no n.(b.addr) and add [b, b', n, t] and del [b', b'', n, t] implies b.addr = b''.addr
53 }
54 check delUndoesAdd for 3
55
56 assert addIdempotent {
57  all b, b', b'': Book, n: Name, a: Addr |
58     add [b, b', n, a] and add [b', b'', n, a] implies b'.addr = b''.addr
59  all b,b',b'': Book,n: Name,t: Target |
60     add [b, b', n, t] and add [b', b'', n, t] implies b'.addr = b''.addr
61 }
62 check addIdempotent for 3
63
64 assert addLocal {
65  all b, b': Book, n, n': Name, a: Addr |
66        add [b, b', n, a] and n != n'implies lookup [b , n'] = lookup [b', n']
67  all b,b': Book,n,n': Name, t: Target |
68        add [b,b',n,t] and n != n'implies lookup [b,n'] = lookup [b',n']
69 }
70 check addLocal for 3 but 2 Book
71
72 assert lookupYields {
73   all b: Book, n: b.names | some lookup [b,n]
74 }
75 check lookupYields with 1,2 for 4 but 1 Book
```

Figure 98: AddressBook specification in Colorful Alloy.

```
1   open util/ordering [Time]
2   open util/ordering [Key]
3   sig Key {}
4   sig Time {}
5   sig Room {
6     keys: set Key,
7     currentKey: keys one -> Time
8   }
9
10  fact DisjointKeySets {
11    Room<:keys in Room lone-> Key
12  }
13
14  one sig FrontDesk {
15    lastKey: (Room -> lone Key) -> Time,
16    occupant: (Room -> Guest) -> Time
17  }
18  sig Guest {
19    keys: Key -> Time
20  }
21
22  fun nextKey [k: Key, ks: set Key]: set Key {
23    min [k.nexts & ks]
24  }
25
26  pred init [t: Time] {
27    no Guest.keys.t
28    no FrontDesk.occupant.t
29    all r: Room | FrontDesk.lastKey.t [r] = r.currentKey.t
30  }
31
32  pred entry [t, t': Time g: Guest, r: Room, k: Key] {
33    k in g.keys.t
34    let ck = r.currentKey |
35     (k = ck.t and ck.t' = ck.t) or
36     (k = nextKey[ck.t, r.keys] and ck.t' = k)
37    noRoomChangeExcept [t, t', r]
38    noGuestChangeExcept [t, t',none]
39    noFrontDeskChange [t, t']
40  }
41
42  pred noFrontDeskChange [t, t': Time] {
43    FrontDesk.lastKey.t = FrontDesk.lastKey.t'
44    FrontDesk.occupant.t = FrontDesk.occupant.t'
45  }
46
47  pred noRoomChangeExcept [t, t': Time, rs: set Room] {
48    all r: Room - rs | r.currentKey.t = r.currentKey.t'
49  }
50
51  pred noGuestChangeExcept [t, t': Time, gs: set Guest] {
52    all g: Guest - gs | g.keys.t = g.keys.t'
53  }
54
55  pred checkout [t, t': Time, g: Guest] {
56    let occ = FrontDesk.occupant {
57     some occ.t.g
58     occ.t' = occ.t - Room -> g
59    }
60    FrontDesk.lastKey.t = FrontDesk.lastKey.t'
61    noRoomChangeExcept [t, t',none]
62    noGuestChangeExcept [t, t', none]
63  }
64
65  pred checkin [t, t': Time, g: Guest , r: Room, k: Key] {
66    g.keys.t' = g.keys.t + k
67    let occ = FrontDesk.occupant {
68     no occ.t [r]
69     occ.t' = occ.t + r -> g }
70    let lk = FrontDesk.lastKey {
71     lk.t' = lk.t ++ r -> k
72     k = nextKey [lk.t [r], r.keys] }
73    noRoomChangeExcept [t, t', none]
74    noGuestChangeExcept [t, t', g]
75  }
```

Figure 99: Hotel specification in Colorful Alloy (part 1).

```
1  (1)abstract sig Event {
2     pre, post: Time,
3     guest: Guest
4  }(1)
5
6  (1)abstract sig RoomKeyEvent extends Event {
7     room: Room,
8     key: Key
9  }(1)
10
11 (1)sig Entry extends RoomKeyEvent {}(1)
12 (1)fact sigEntry {
13    key in guest.keys.pre
14    let ck = room.currentKey |
15    (key = ck.pre and ck.post = ck.pre) or
16    (key = nextKey[ck.pre, room.keys] and ck.post = key)
17    currentKey.post = currentKey.pre ++ room->key
18 }(1)
19
20 (1)sig Checkin extends RoomKeyEvent {}(1)
21 (1)fact SigCheckin {
22    keys.post = keys.pre + guest -> key
23    let occ = FrontDesk.occupant {
24     no occ.pre [room]
25     occ.post = occ.pre + room -> guest }
26    let lk = FrontDesk.lastKey {
27     lk.post = lk.pre ++ room -> key
28     key = nextKey [lk.pre [room], room.keys] }
29 }(1)
30
31 (1)sig Checkout extends Event {}
32 (1)fact sigCheckout {
33    let occ = FrontDesk.occupant { some occ.pre.guest and occ.post = occ.pre - Room -> guest }
34 }(1)
35
36 fact traces {
37    init [first]
38    (1)all t: Time-last | let t' = t.next |some g: Guest, r: Room, k: Key |
39     entry [t, t', g, r, k] or checkin [t, t', g, r, k] or checkout [t, t', g](1)
40    (1)all t: Time-last | let t' = t.next | some e: Event {
41       e.pre = t and e.post = t'
42         currentKey.t != currentKey.t' => e in Entry
43         occupant.t != occupant.t' => e in Checkin + Checkout
44         (lastKey.t != lastKey.t' or keys1.t != keys1.t') => e in Checkin }(1)
45 }
46
47 fact NoIntervening {
48    (1)(2)all t: Time-last | let t' = t.next, t" = t'.next |
49      all g: Guest, r: Room, k: Key |
50       checkin [t, t', g, r, k] = (entry [t', t", g, r, k] or no t")(2)(1)
51    (1)(2)all c: Checkin |
52     c.post = last
53     or some e: Entry { e.pre = c.post e.room = c.room e.guest = c.guest }(2)(1)
54 }
55
56 assert NoBadEntry {
57    (1)all  t : Time, r : Room, g : Guest, k : Key |
58     let t' = t.(next) | let o = r.(FrontDesk.occupant.t) |
59     (entry[t, t', g, r, k]) and some o = g in o(1)
60    (1)all e : Entry |
61      let o = e .room.(FrontDesk.occupant.(e.pre)) | some o =>  e.guest in o(1)
62 }
63 check NoBadEntry for 3 but 2 Room, 2 Guest, 5 Time
```

Figure 99: Hotel specification in Colorful Alloy (part 2).