Mastering variability in the wild : on object-oriented variability implementations and variability-aware build systems

Johann Mortara

UNIVERSITÉ CÔTE D'AZUR

ÉCOLE DOCTORALE
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

THÈSE DE DOCTORAT

Maîtriser la variabilité enfouie dans les systèmes orientés objet et les systèmes de construction logicielle

Johann MORTARA

Laboratoire d'Informatique, de Signaux et Systèmes de Sophia Antipolis (I3S)
UMR7271 CNRS

Présentée en vue de l'obtention du grade de docteur en Informatique **d'**Université Côte d'Azur

Dirigée par : Philippe COLLET, Professeur des Universités, Université Côte d'Azur
Soutenue le : 20 décembre 2022

Devant le jury, composé de :
Romain ROUVOY, Professeur des Universités, Université de Lille
Xavier BLANC, Professeur des Universités, Université de Bordeaux
Jean-Marc JÉZÉQUEL, Professeur des Universités, Université de Rennes 1
Anne-Marie DERY-PINNA, Maîtresse de Conférences, Université Côte d'Azur
Xhevahire TËRNAVA, Chercheuse postdoctorale, Université de Rennes 1

i3s sophia antipolis

MAÎTRISER LA VARIABILITÉ ENFOUIE DANS LES SYSTÈMES ORIENTÉS OBJET ET LES SYSTÈMES DE CONSTRUCTION LOGICIELLE

Mastering Variability in the Wild: On Object-Oriented Variability Implementations and Variability-Aware Build Systems

Johann MORTARA

Jury :

Président du jury
Romain ROUVOY, Professeur des Universités, Université de Lille

Rapporteurs
Xavier BLANC, Professeur des Universités, Université de Bordeaux
Jean-Marc JÉZÉQUEL, Professeur des Universités, Université de Rennes 1

Examinateurs
Anne-Marie DERY-PINNA, Maîtresse de Conférences, Université Côte d'Azur

Directeur de thèse
Philippe COLLET, Professeur des Universités, Université Côte d'Azur

Membres invités
Xhevahire TËRNAVA, Chercheuse postdoctorale, Université de Rennes 1

Université Côte d'Azur

Johann MORTARA

Maîtriser la variabilité enfouie dans les systèmes orientés objet et les systèmes de construction logicielle

xv+203 p.

Maîtriser la variabilité enfouie dans les systèmes orientés objet et les systèmes de construction logicielle

Résumé

La demande sans cesse croissante de solutions logicielles nouvelles et récentes oblige les professionnels du logiciel à développer et à maintenir des systèmes logiciels personnalisables tout en garantissant un niveau élevé de qualité et de fiabilité. Si les lignes de produits logiciels (LPLs) constituent une solution pour atteindre cet objectif, de nombreux systèmes logiciels riches en variabilité ne sont pas organisés de cette manière. Ils augmentent progressivement leurs parties variables, en s'appuyant sur les multiples mécanismes existants pour mettre en œuvre leur variabilité dans le code et leur chaîne d'outils de construction. Dans leur mise en œuvre, les systèmes orientés objet (OO) gèrent principalement leur variabilité dans une base de code unique en utilisant des mécanismes OO tels que l'héritage et les patrons de conception. En raison de leur nature, ces implémentations sont enfouies dans la base de code, ce qui nuit à la compréhension du système par les développeurs et, par conséquent, à sa maintenance et à son évolution, entraînant des problèmes de qualité. En outre, les grands systèmes logiciels riches en variabilité s'appuient souvent sur des systèmes de construction complexes pour sélectionner les éléments du code. Comme il s'agit de systèmes de construction ad hoc réutilisant des outils standard, aucune représentation globale du mécanisme de résolution de la variabilité n'est disponible, et des conflits peuvent survenir et causer des anomalies. Dans cette thèse, nous proposons tout d'abord les bases et les techniques pour identifier et visualiser les implémentations de la variabilité dans les grands systèmes logiciels OO riches en variabilité. Ces implémentations sont abstraites en termes de points de variation et de variantes et identifiées en s'appuyant sur la notion de densité de symétries dans les structures OO. En reprenant la métaphore d'une ville, elles sont ensuite visualisées sous la forme d'un ensemble connecté de bâtiments 3D combinés à des métriques sur leur qualité. Cela permet de distinguer les zones concentrant les implémentations de variabilité et présentant potentiellement une dette technique. Ces propositions ont été validées par un prototype sur de grands systèmes logiciels OO open-source et hautement variables, ainsi que par une étude d'utilisabilité avec deux groupes distincts de développeurs débutants. La thèse introduit également un cadre de modélisation et de raisonnement pour caractériser les anomalies dans les systèmes de construction gérant de la variabilité, permettant de raisonner sur les relations entre les actifs du code, et d'identifier toutes ces anomalies au grain le plus fin. Le framework a été instancié et partiellement implémenté à la fois sur le système de construction du noyau Linux, démontrant sa généralité sur les nombreuses détections distinctes sur ce sujet très étudié, et sur une chaîne d'outils de construction récemment étudiée de la fondation Mozilla, démontrant son applicabilité.

Mots-clés : Génie logiciel, Lignes de produits logiciels, Variabilité logicielle, Modèle de variabilité, Rétro-ingénierie, Visualisation.

Mastering Variability in the Wild: On Object-Oriented Variability Implementations and Variability-Aware Build Systems

Abstract

The constantly increasing demand for new and up-to-date software solutions compels software practitioners to develop and maintain customizable software systems while assuring a high level of quality and reliability. While Software Product Lines (SPLs) are a solution towards this goal, many variability-rich software systems are not organized as such. They progressively grow their variable parts, relying on existing multiple mechanisms to implement their variability in code and their building toolchain. In their implementation, object-oriented (OO) systems mainly manage their variability in a single codebase using OO mechanisms such as inheritance and patterns. Due to their nature, these implementations are buried in the codebase, hampering the system's comprehension for developers and thus its maintenance and evolution, causing quality issues. Additionally, large variability-rich software systems often rely on complex build systems to select code assets. As they are widely ad hoc build systems reusing off-the-shelf tools, no global representation of the overall variability resolution mechanism is available, and conflicts may happen and cause anomalies. In this thesis, we first propose the foundations and techniques to identify and visualize variability implementations in large OO variability-rich software systems. These implementations are abstracted in terms of variation points and variants and identified relying on the notion of density of symmetries in OO structures. Following a city metaphor, they are then visualized in the form of a connected set of 3D buildings together with metrics on their quality. This helps distinguish zones concentrating variability implementations and potentially exhibiting technical debt. These proposals have been validated by a prototyped application on large open-source and highly-variable OO software systems, as well as a usability study with two separate groups of newcomer developers. The thesis also introduces a modeling and reasoning framework to characterize anomalies in variability-aware build systems, allowing to reason on code assets relationships, and identify all these anomalies at the finest grain. The framework was instantiated and partially tooled on both the Linux kernel build system, demonstrating its generality over the many separate detections on this heavily studied subject, and a newly studied build toolchain from the Mozilla foundation, showing applicability.

Keywords: Software engineering, Software product lines, Software variability, Variability modeling, Reverse-engineering, Visualization.

Acknowledgements

Je n'ai pas les mots pour exprimer toute la gratitude que j'ai envers Philippe qui a eu la lourde charge de me supporter pendant un peu plus de 3 ans. Le fait que nous étions d'accord sur une multitude de points a grandement contribué à la mise en place et au maintien de notre relation. Si je devais en citer deux, je dirais une exigence forte dans la qualité de travail et une exigence faible dans la qualité des vannes. Faisant toujours preuve de gentillesse et de bienveillance, il est pour moi un modèle tant en ce qui concerne la recherche que l'enseignement. J'ai énormément grandi en tant que scientifique mais également en tant que personne à ses côtés, et je souhaite à quiconque de l'avoir comme directeur de thèse.

Je souhaite également remercier deux personnes sans qui je n'aurais jamais fait cette thèse : Sébastien Mosser et Xhevahire Tërnava. Seb m'a permis de faire mes premiers pas dans le monde de la recherche en me proposant un stage de deux mois dans l'équipe pour finir ma 4^{e} année d'école d'ingénieur, avant de me laisser entre les mains de Philippe pour mon PFE et mon stage en dernière année. C'est alors que j'ai commencé à échanger avec Xheva sur ses premières intuitions qui ont été à l'origine d'une longue collaboration et des travaux de cette thèse. Je les remercie pour m'avoir fait découvrir le monde de la recherche et m'avoir apporté ces expériences qui m'ont donné envie de continuer en doctorat.

Je ne peux que souhaiter à tout étudiant qui souhaite se lancer dans l'aventure d'un doctorat d'intégrer une équipe de recherche comme Muscat. Je regretterai de ne plus partager mon quotidien avec :

- Anne-Marie qui, grâce à son don pour détecter quand quelqu'un ne va pas avant même qu'il ne le sache, arrive en un rien de temps à remonter le moral des troupes en transmettant des bonnes ondes qui nous font relativiser les petits tracas du quotidien ;
- Mireille, avec qui j'ai partagé bon nombre d'intenses sessions de brainstorming sur des sujets scientifiques et philosophiques (et pas que !) mais que je ne réussirai jamais à battre au sprint avec des talons ;
- Nassim, qui a occupé pendant un an et demi 50% du bureau et 90% du tableau mais qui en contrepartie y a apporté de la vie, des bonnes blagues et des sucreries (je suis plutôt satisfait du deal finalement) ;
- Yassine, qui a participé à la hausse de ma glycémie en nous apportant une forte variabilité de pâtisseries que j'ai dû toutes essayer par esprit scientifique ;
- Yann, que l'on n'a visiblement pas assez traumatisé en Master pour qu'il décide de rester, et grâce à qui j'ai pu étendre mon état de l'art des boulangeries et restaurants de Nice.

Je souhaite bon courage à Yassine et à Nassim pour la fin de leur thèse ainsi qu'à Yann pour la fin de son alternance. Mais je ne m'inquiète pas, vous êtes entre de bonnes mains !

J'ai une pensée pour toutes les personnes qui m'ont accompagné chacune à leur manière au cours de cette aventure. Je ne peux toutes les citer, mais en voici quelques unes.

- Thibaut, qui fut mon alibi pour prendre des pauses café au labo au cours desquelles nous partageâmes un nombre indécent de *tikafés* avec des *tigatos* en échangeant sur des sujets aussi variés qu'intéressants ;
- Tim, avec qui les *"eh salut !"* rapides en passant dans le couloir se transformaient assez régulièrement en discussions philosophiques, ce qui a considérablement allongé la durée moyenne de mes pauses café ;
- Sara, Pietro et Piergiorgio dont les soirées jeux de plateau qui m'ont permis d'étendre de manière assez stupéfiante mon catalogue de jurons en italien ;
- Nassim (oui, toujours le même) et Zaineb avec qui j'ai partagé bon nombre d'aventures qu'il m'est impossible de détailler ici et qui m'ont forcé à sortir la tête de l'eau à des moments où j'en avais vraiment besoin ;
- Sabine qui fait un travail extraordinaire pour organiser et suivre les missions, et grâce à qui j'ai pu rentrer chez moi quand une grève des contrôleurs aériens s'est déclarée alors que j'étais en Autriche ;
- L'OR, Côte d'Or et Lindt pour les cafés et chocolats consommés au cours des pauses de 7h45, 10h, 13h, 14h30 et 16h que j'ai partagées avec plusieurs des personnes sus-citées.

J'ai également une pensée pour les étudiants à qui j'ai eu le plaisir d'enseigner, car c'est à ce moment que je me suis aperçu que je n'avais aucune maîtrise des concepts que j'enseignais. J'espère qu'ils ne s'en sont pas rendu compte.

Je remercie mes parents qui m'ont toujours apporté leur soutien et, bien que n'étant pas sûrs de comprendre ce que je fais, savent que des bons petits plats et desserts maison sont un excellent moyen de me réconforter quand le moral baisse.

Enfin, je remercie Antonia dont la présence dans ma vie est l'une des plus belles choses qui me soit arrivé. Tu m'as fait voir la vie autrement et découvrir des choses que je n'imaginais pas. Merci d'avoir été là dans tous les moments difficiles que j'ai eus à traverser et d'avoir contribué à en créer des meilleurs.

Pour être honnête, si je devais résumer ma thèse aujourd'hui avec vous, je dirais que c'est d'abord des rencontres, des gens qui m'ont tendu la main, peut-être à un moment où je ne pouvais pas, où j'étais seul chez moi. Et c'est assez curieux de se dire que les hasards, les rencontres forgent une destinée... Parce que quand on a le goût de la chose, quand on a le goût de la chose bien faite, le beau geste, parfois on ne trouve pas l'interlocuteur en face, je dirais, le miroir qui vous aide à avancer. Alors ce n'est pas mon cas, comme je le disais là, puisque moi au contraire, j'ai pu ; et je dis merci à la vie, je lui dis merci, je chante la vie, je danse la vie... Je ne suis qu'amour ! Et finalement, quand beaucoup de gens aujourd'hui me disent : « Mais comment fais-tu pour avoir cette humanité ? » Eh bien je leur réponds très simplement, je leur dis que c'est ce goût de la science, ce goût donc qui m'a poussé aujourd'hui à entreprendre une thèse, mais demain, qui sait, peut-être simplement à me mettre au service de la communauté, à faire le don, le don de soi.

Contents

List of Acronyms

vp(-s)	Variation Point(s)
CPP	C Preprocessor
FM	Feature Model
LoC	Lines of Code
OO	Object-Oriented
SPL	Software Product Line
VM	Variability Model

Chapter 1

Introduction

We build our computer systems the way we build our cities: over time, without a plan, on top of ruins.

— Ellen Ullman, Life in Code: A Personal History of Technology.

1.1 Context

The constantly increasing demand for software solutions constrains software practitioners to develop and maintain customizable software systems. Such systems, which can range from small-scale embedded systems to large-scale systems of systems, are qualified as variability-intensive [Hilliard, 2010; Galster et al., 2013; Galster, 2019]. Software variability is the capacity of a software system to be tailored for a given need or context [Capilla et al., 2013]. In order to limit the time and effort spent in the development and maintenance of such systems, organizations rely on approaches based on software reuse [Krueger, 1992; Jacobson et al., 1997] to manage their code assets such as Software Product Lines (SPLs).

An SPL is defined as *"a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way"* [Clements and Northrop, 2002; Northrop et al., 2007]. Its engineering process is illustrated on Figure 1.1. Features represent *"prominent or distinctive user-visible aspects, qualities, or characteristics of a software system or systems"* [Kang et al., 1990] and can be organized in a feature model (FM) [Batory, 2005] during the *domain analysis* phase (*cf.* Figure 1.1). Features are typed as mandatory, alternative, or optional, allowing to model commonalities and variations between the different products that can be derived from the SPL (*cf.* Section 2.1). Then, a mapping between each feature and the code assets implementing it is built relying on one or more variability implementation techniques such as CPP directives (often called `ifdefs`) [Liebig et al., 2010; Hunsen et al., 2016], or other forms of annotations [Couto et al., 2011]. While such techniques can be incorporated in existing code assets, others isolate code assets implementing each feature by organizing them in feature modules [Apel et al., 2006; Takeyama and Chiba, 2013] that can be implemented with the use of aspects [Griss, 2000; Voelter and Groher, 2007; Figueiredo et al., 2008] (*domain implementation*). Therefore, code assets can be separated into three parts: core, commonalities, and variations [Turner et al., 1999; Coplien, 1999; Bachmann and Clements, 2005]. The core part represents the code assets that are not mapped to any feature and are therefore included in every product of the SPL [Turner et al., 1999]. A commonality is a common part between the related variations of given code assets, while variations indicate how and when should code assets vary [Hilliard,

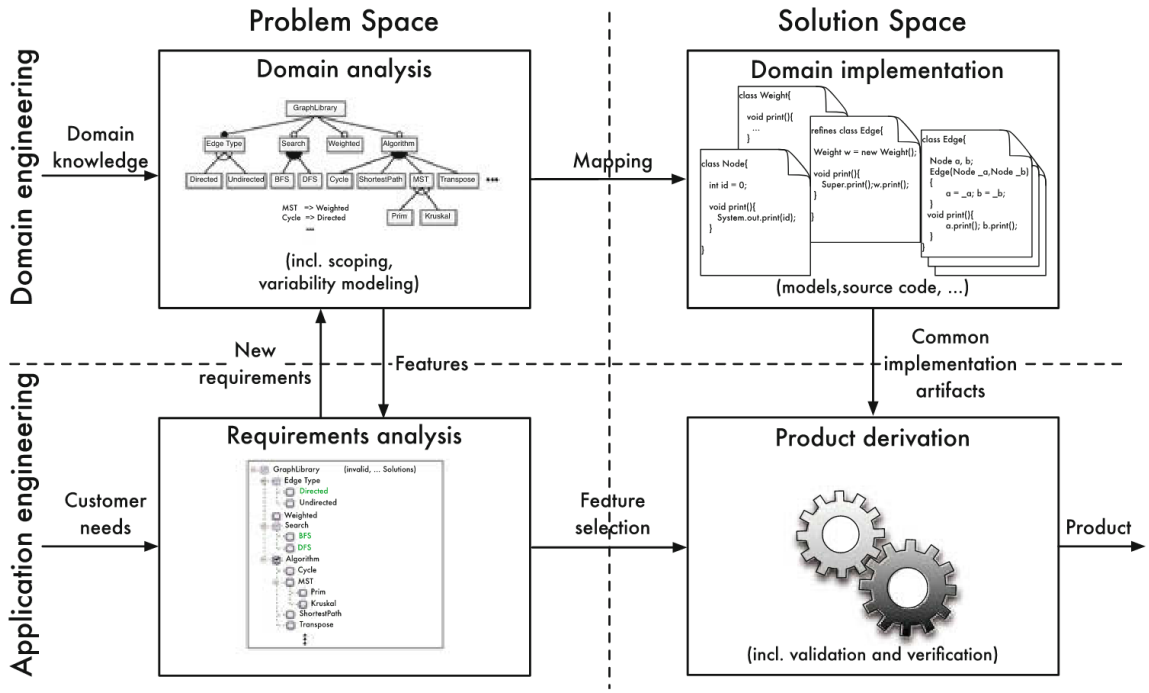


Figure 1.1: Software Product Line engineering process (extracted from [Nadi, 2015])

2010]. Such commonalities and variations are usually abstracted in terms of variation points (*vp*-s) with *variants*, respectively [Jacobson et al., 1997; Czarnecki et al., 2012; Rabiser, 2019], which are related to concrete elements in code assets [John et al., 2007].

To derive a product variant of the SPL, customers define their own configuration by selecting a set of features corresponding to their needs (*requirements analysis*) that is then used by a build system that checks its validity before assembling the variant by selecting the code assets mapped to the features (*product derivation*).

1.2 "Wild" variability in software systems

In a standard SPL approach, a variability implementation mechanism establishes the mapping between each domain feature and the code assets implementing it. When deriving a variant, the build system is then in charge of the resolution of this variability, conducting to the selection of the code assets that will constitute the final product. Some systems, however, are not implemented in such a way although they are variability-intensive.

1.2.1 Wild variability implementations in object-oriented systems

Due to the characteristics of their variability implementation mechanisms, object-oriented (OO) systems often do not follow a fully-fledged software product line approach. [Pohl et al., 2005; Apel et al., 2013]. They often implement these *vp*-s and variants relying on the mechanisms provided by these languages, namely inheritance, overloading of methods and constructors, and some design patterns [Gacek and Anastasopoules, 2001; Svahnberg et al., 2005; Capilla et al., 2013;

Tërnava and Collet, 2017a]. As opposed to other implementation techniques such as annotative approaches, mechanisms used to implement variability in OO systems do not allow making explicit the domain variability in the code assets. Moreover, such systems hardly document the domain variability [Krüger et al., 2019b]. Therefore, it results that although they are variability-rich, many OO software systems do not follow a fully-fledged software product line approach [Pohl et al., 2005; Apel et al., 2013] as the mapping between the domain variability and its implementation is either lost or non-existent [Krüger et al., 2019b].

Problem Consequently, the comprehension of the implemented variability is hampered and eventually leads to a decrease in the ability to maintain the system and make it evolve [Favre, 1996; Kröher et al., 2022], creating technical debt (*i.e.*, expedient but costly on the long term implementation constructs, primarily hampering maintainability and evolvability) [Avgeriou et al., 2016; Li et al., 2015]. Monitoring quality being crucial for the maintenance and evolution of such systems [Martini and Bosch, 2015], it is of prime importance for software developers and architects to comprehend the variability that is implemented in their system. In other words, **there is a need to comprehend the variability implemented in OO software systems**.

1.2.2 Wild variability implementations in variability-aware build systems

Due to their important size, large variability-rich software systems commonly rely on multiple mechanisms to implement their variability at multiple granularities in a single codebase. While in a traditional SPL approach variability is implemented using tools explicitly designed for this purpose that enable a global vision of the implemented variability and ensures a safe mapping with domain features [Meinicke et al., 2017; Beuche, 2019; Krueger and Clements, 2013], these build systems often rely on combinations of handcrafted tools and existing solutions adapted to be used as variability resolution tools. When deriving a variant of the system, a variability-aware build system is then in charge of invoking the different tools to resolve each type of variability implementation based on a given configuration. For example, the Linux kernel's build system[1] is divided in three steps [Nadi and Holt, 2014]:

- a configuration step, KCONFIG, that builds a configuration based on user input and constraints between features;
- two derivation steps, the KBUILD and CPP, that define conditions on features to select code assets at two levels of granularity (source files and code blocks respectively). If the configuration satisfies a condition, the asset is selected.

Problem In such build systems, the execution of a derivation step might influence the execution of another one (*e.g.*, for a given configuration, if a source file is not selected, neither of the code blocks it contains can be). As different tools are used to implement variability in the different steps, each derivation step is uncorrelated from the others. This prevents checking the consistency of the whole build system and can lead to anomalies, such as parts of the code that can never be selected due to features in conflict (*e.g.*, no configuration selecting a code block selects the file containing it) [Nadi et al., 2013; Nadi, 2014], potentially creating bugs in derived product variants. In other words, **there is a need to comprehend the variability managed by build systems**.

[1] https://www.kernel.org/doc/html/latest/kbuild/index.html

1.3 Taming wild variability

In this thesis, we aim to improve the comprehension of variability in systems that are not fully-fledged SPLs by focusing our attention on (*i*) variability implementations relying on OO mechanisms and (*ii*) their management in complex ad-hoc build systems. Hereafter, we detail the challenges to be addressed in order to reach these goals.

A. Comprehending variability implemented in OO software systems

A1. Identifying variability implemented in OO software systems

OO variability implementations are of diverse natures [Lozano, 2011; Tërnava and Collet, 2017a] and the used mechanisms (*i.e.*, inheritance, overloading of methods and constructors, and design patterns) do not allow for keeping trace of domain information. Therefore, with the evolution of the system, information on the location of the variability is lost, hampering their capacity to evolve and to be comprehended. Moreover, such mechanisms are also used to structure the code [Gamma et al., 1993, 1995]. Variability implementations are thus buried in the implementation code, increasing the difficulty of their identification.

State-of-the-art techniques to identify variability implementations in OO systems (detailed in Section 3.1.1) mostly target *clone-and-own* systems [Roy and Cordy, 2007; Michelon et al., 2019; Linsbauer et al., 2022]. However, in our case, OO mechanisms allow implementation of variability in a single codebase, thus preventing the application of these techniques. Other techniques rely on traces obtained by executing the system [Walkinshaw et al., 2007] or the unit tests [Eisenberg and De Volder, 2005], which is not always possible for large systems. There is therefore a need for a technique allowing static identification of variability implemented using OO mechanisms in systems managed in a single codebase.

A2. Making the identified variability implementations comprehensible

When variability is implemented relying on OO mechanisms, *vp*-s and variants can be implemented at multiple granularities [Tërnava et al., 2019]. At class level, *vp*-s are classes or interfaces having as variants their subclasses or implementations. At method level, *vp*-s are overloaded methods and constructors having for variants their overloads. Additionally, these *vp*-s and variants compose each other through the use of attributes or method parameters to build more advanced structures as design patterns. This diversity of mechanisms involved causes the organization of OO variability implementations to be highly complex. Consequently, their identification cannot consist of a reverse-engineered list of features, imposing to rely on a semi-automatic approach extracting information and metrics on the presence of mechanisms involved in variability implementations. These metrics are then used as a support to a software developer or architect having the domain knowledge to identify the implemented variability. However, on large systems, such an identification technique would output a too important volume of data to be understandable in a textual format. As software metrics are often comprehended through the use of visualizations [Domingue, 1998; Knight and Munro, 2000; Diehl, 2007; Wettel et al., 2010], we advocate that a visualization support for metrics on OO variability implementation would help the comprehension of the variability implemented in such large variability-rich systems.

A3. Understanding the quality of the implemented variability

Variability implementations are known to bring additional complexity to the system [Galster et al., 2017]. Consequently, its maintainability is threatened and its capacity to evolve hampered [Favre, 1996; Kröher et al., 2022], eventually leading to technical debt [Avgeriou et al., 2016; Li et al., 2015]. Technical debt due to variability implementations has been largely studied [Mordahl et al., 2019], leading to new definitions [Fenske and Schulze, 2015] and a recent characterization of *variability debt* by Wolfart et al. [2021]. As OO systems reuse the traditional OO mechanisms to implement their variability, there is no dedicated implementation mechanism, causing the variability to be intertwined with the implementation (*cf.* Challenge A1). Consequently, these systems are prone to variability debt in their implementation. As monitoring technical debt and, more globally, the quality of the system is crucial for its maintenance and evolution [Martini and Bosch, 2015], there is a need for a solution combining quality metrics and OO variability metrics (Challenge A2), allow identifying the technical debt that can be induced by these variability implementations.

B. Comprehending the variability managed by build systems

B1. Making explicit the derivation mechanism of build systems

As detailed in Section 1.2.2, the selection of an asset in a step of the build system has an influence on the selection of other assets, being in the same step or not. In order to master the variability managed by build systems and prevent anomalies, it is important to understand, for a given asset, which other assets manipulated by the build system can influence its selection. As opposed to systems for which the variability is designed using variability modeling approaches [Meinicke et al., 2017], build systems often rely on adapted off-the-shelf solutions and do not allow characterizing these dependencies between assets. While multiple works focus on checking the selection of an asset in the different steps of the Linux kernel [Hengelein, 2015; Nadi and Holt, 2012; Sincero et al., 2010], they do not provide a fine-grain representation of the whole build system mechanism modeling precisely the conditions under which a code asset is present in a derived variant. Such a representation is needed (*i*) to understand with precision the dependencies that exist between code assets in the whole build system and (*ii*) to apply the contributions from the state-of-the-art, that are focused on the Linux kernel's build system, to other build systems.

B2. Characterizing and identifying anomalies in build systems

The interactions between dependent code assets can lead to unwanted side effects. For example, code assets can become *dead* if they can never be selected (*e.g.*, a code block whose selection condition is incompatible with the one of its parent file). On the opposite, code assets can also become *core* if they are always selected (*e.g.*, a code block whose selection condition is always true when the one of its parent file also is). Such consequences are called *anomalies* as they change the nature of the implemented variability (*e.g.*, a *core* anomaly leads to a code asset being part of the core of the system while it is a variation according to the implemented variability). While multiple software systems make use of build systems relying on common concepts (*i.e.*, definition and validation of a configuration followed by one or more variability resolution steps), the Linux kernel is the one that received the most attention. Anomalies in and between each of the three steps have been defined [Hengelein, 2015; Nadi and Holt, 2012; Sincero et al., 2010] together with tooled support identifying the anomalies in the kernel's code assets. While these tooled ap-

proaches demonstrated successful identification of anomalies, they all focus on isolated steps of the build system, and the formal definitions of the presented anomalies show little detail about the constraints issuing from other steps. Among contributions, similar concepts have different denominations, common anomalies have different definitions, and all rely on different formalisms. This diversity prevents (*i*) a deep comprehension of the anomalies and how to identify them, and (*ii*) the application of these contributions to other similar build systems that are prone to similar anomalies, such as the Mozilla build system that also makes use of CPP directives to select code blocks in its source files. There is therefore a need to synthesize these contributions and the anomalies they present by uniformizing them under a single representation that can be instantiated on build systems to identify the anomalies.

1.4 Methodology

This Ph.D. work has been conducted relying on an experimental approach. Each contribution provided new observations that allowed the definition of new challenges and incrementally improve the approach. As a consequence, Chapters 5 to 7 have similar plans illustrating this methodology. The proposed contribution is implemented on top of the previous ones and evaluated on a set of subject systems. The observations arising from these conducted experiments exhibit the limitations of the contribution, driving the motivations for the following one.

The work of this Ph.D. thesis started as a Master's thesis directed by Philippe Collet in collaboration with Xhevahire Tërnava and consisted in designing a tool automating an identification approach for OO variability implementations she designed. This joint work led to two contributions that we use as a starting point for this work.

- In [Tërnava et al., 2019], we define *vp*-s and variants in the context of OO variability implementations and propose a technique to identify these variability implementations relying on the notion of symmetry in code assets [Zhao and Coplien, 2002, 2003].
- The *symfinder* toolchain [Mortara et al., 2019] automates this identification on systems in Java and provides a visualization of the identified potential *vp*-s and variants in the shape of a graph.

More details on these contributions are given in Section 3.1.2.

As a result, Chapters 5 to 7 successively increment this preliminary work, that we first assess in Chapter 4 to evaluate to what extent it provides first elements of answers to Challenges A1 and A2.

1.5 Plan

This thesis is organized as follows:

- Chapter 2 presents elements of background on software product lines and how variability is modeled and implemented in this context. We also present an overview of the different techniques allowing to implement variability in software systems and introduce variability-aware build systems, focusing on two examples: the Linux kernel build system and the Mozilla build system.

- Chapter 3 details the state of the art on techniques to identify and comprehend the variability implemented in software systems and its quality. Then, we detail work studying the organization of variablity-aware build systems and finally focus on work characterizing anomalies in these build systems.
- In Chapter 4, we assess the identification technique proposed by Tërnava et al. [2019] and the associated tool support *symfinder* [Mortara et al., 2019] on three aspects: (*i*) we extend the approach to C++ constructs to validate the use of symmetry to identify OO variability implementations, (*ii*) we map the identified potential *vp*-s and variants to existing traces of features in the implementation of two systems to validate the relevance of these *vp*-s and variants, and (*iii*) we evaluate the interpretability of the graph visualization proposed by *symfinder* by exchanging with a software architect on his experience using *symfinder* on his software.
- In Chapter 5, we extend the identification approach by taking into account usage relationships (*i.e.*, composition/aggregation) between *vp*-s and variants and rely on them to characterize a measure of density of variability implementations. Consequently, the visualization has also been improved and shows these two additional information. We implement this new approach in *symfinder-2* and evaluate it by applying it on multiple open source software systems and reporting on the newly revealed dense zones of variability implementations.
- Chapter 6 presents *VariCity*, a new visualization approach adapting the metaphor of the city to represent the variability implementations identified by *symfinder-2* and assist the exploration of a system by revealing dense zones of variability implementations. We evaluate the capabilities of the visualization by relying on onboarding scenarios on multiple open source software systems, and report on the results of a controlled experiment with 49 students aiming to compare *VariCity* to standard tools used in code comprehension activities.
- Chapter 7 introduces *VariMetrics*, an extension of the *VariCity* visualization to support software quality metrics and reveal critical zones concentrating variability implementations prone to cause variability debt in a single OO codebase. We define quality metrics allowing to identify variability debt in the context of OO variability implementations and evaluate our approach by visually identifying indebted zones of variability implementations on multiple open source software systems. We validate the relevance of identified zones in one system by comparing its quality before and after applying maintenance actions to the indebted code assets.
- Chapter 8 details existing work characterizing anomalies in the Linux kernel build system and proposes a model unifying these contributions. This model provides a detailed representation of the conditions determining the selection of a code asset in a variability-aware build system. We demonstrate the completeness of our representation by instantiating the definitions of anomalies from the studied contributions in the proposed representation and exhibit incoherences between them.
- Chapter 9 presents a generalization of the Linux-centered model introduced in Chapter 8 by taking into account the diversity of mechanisms exhibited by the Mozilla build system. A framework implementing this model and allowing the identification of anomalies as well as their detailed representation is proposed and evaluated on the code assets of both the Linux kernel build system and the Mozilla build system.

- Chapter 10 summarizes the contributions presented in this thesis and the answers they provide to the challenges presented in Section 1.3 before detailing perspectives on the short and long terms.

1.6 Contributions

Johann Mortara and Philippe Collet. Capturing the diversity of analyses on the Linux kernel variability. In *Proceedings of the 25th ACM International Systems and Software Product Line Conference - Volume A*, SPLC '21, page 160–171, New York, NY, USA, September 2021a. Association for Computing Machinery. ISBN 978-1-4503-8469-8. doi: 10.1145/3461001.3471151. URL https://hal.archives-ouvertes.fr/hal-03283627.

Johann Mortara and Philippe Collet. Capturing the diversity of analyses on the Linux kernel variability – Companion Technical Report. July 2021b. URL https://hal.archives-ouvertes.fr/hal-03283633.

Johann Mortara and Philippe Collet. How I Met Your Implemented Variability: Identification in Object-Oriented Systems with symfinder. In *25th ACM International Systems and Software Product Line Conference - Volume A (SPLC '21)*, Leicester, United Kingdom, September 2021c. doi: 10.1145/3461001.3472733. URL https://hal.archives-ouvertes.fr/hal-03274636.

Johann Mortara, Xhevahire Tërnava, and Philippe Collet. symfinder: A Toolchain for the Identification and Visualization of Object-Oriented Variability Implementations. In *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume B*, SPLC '19, pages 5–8, New York, NY, USA, September 2019. Association for Computing Machinery. ISBN 978-1-4503-6668-7. doi: 10.1145/3307630.3342394. URL https://hal.archives-ouvertes.fr/hal-02342730.

Johann Mortara, Philippe Collet, and Xhevahire Tërnava. Identifying and Mapping Implemented Variabilities in Java and C++ Systems using symfinder. In *Proceedings of the 24th ACM International Systems and Software Product Line Conference - Volume B*, SPLC '20, page 9–12, New York, NY, USA, October 2020a. Association for Computing Machinery. ISBN 978-1-4503-7570-2. doi: 10.1145/3382026.3431251. URL https://hal.archives-ouvertes.fr/hal-02908531.

Johann Mortara, Xhevahire Tërnava, and Philippe Collet. Mapping Features to Automatically Identified Object-Oriented Variability Implementations: The Case of ArgoUML-SPL. In *Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems*, VaMoS '20, pages 1–9, New York, NY, USA, February 2020b. Association for Computing Machinery. ISBN 978-1-4503-7501-6. doi: 10.1145/3377024.3377037. URL https://hal.archives-ouvertes.fr/hal-02421353.

Johann Mortara, Philippe Collet, and Anne-Marie Dery-Pinna. Visualization of Object-Oriented Variability Implementations as Cities. In *2021 Working Conference on Software Visualization (VISSOFT)*, pages 76–87, Luxembourg (virtual), Luxembourg, September 2021a. ISBN 978-1-6654-3144-6. doi: 10.1109/VISSOFT52517.2021.00017. URL https://hal.archives-ouvertes.fr/hal-03312487.

Johann Mortara, Xhevahire Tërnava, Philippe Collet, and Anne-Marie Dery-Pinna. Extending the Identification of Object-Oriented Variability Implementations using Usage Relationships. In *SPLC 2021 - 25th ACM International Systems and Software Product Line Conference*, volume Volume B, pages 1–8, Leicester, United Kingdom, September 2021b. ACM. doi: 10.1145/3461002.3473943. URL https://hal.archives-ouvertes.fr/hal-03284626.

Johann Mortara, Philippe Collet, and Anne-Marie Dery-Pinna. Customizable Visualization of Quality Metrics for Object-Oriented Variability Implementations. In *Proceedings of the 26th ACM International Systems and Software Product Line Conference - Volume A*, SPLC '22, page 43–54, New York, NY, USA, September 2022a. Association for Computing Machinery. ISBN 978-1-4503-9443-7. doi: 10.1145/3546932.3547073. URL https://hal.archives-ouvertes.fr/hal-03717858.

Johann Mortara, Philippe Collet, and Anne-Marie Dery-Pinna. IDE-assisted visualization of indebted OO variability implementations. In *Proceedings of the 26th ACM International Systems and Software Product Line Conference - Volume B*, SPLC '22, page 74–77, New York, NY, USA, September 2022b. Association for Computing Machinery. ISBN 978-1-4503-9206-8. doi: 10.1145/3503229.3547066. URL https://hal.archives-ouvertes.fr/hal-03717874.

CHAPTER 2

Background

This chapter provides the reader with the fundamental concepts used throughout the thesis by introducing the background knowledge needed for its understanding. We first detail how domain and implementation variability are modeled in a software product line (SPL) context (Section 2.1). We then provide an overview of the different techniques allowing to implement variability in variability-rich systems (Section 2.2.1) before focusing on variability implementations in OO systems (Section 2.2.2). Finally, we present variability-aware build systems and detail how they reuse existing tools to manage variability by relying on two examples: the Linux kernel build system and the Mozilla build system (Section 2.3).

2.1 Modeling and implementing variability in Software Product Lines

The domain and realization of SPLs rely on the notion of feature, being a user-visible aspect of a software system [Kang et al., 1990]. After they have been determined, they are organized in a variability model that, most often, has the shape of a feature model [Batory, 2005]. An example of feature model describing the domain of a library manipulating graphs is given in Figure 2.1. The abstract feature `GraphLibrary` represents conceptually the SPL domain, which has a single compound mandatory feature `EdgeType` and three compound optional features `Search`, `Weighted` and `Algorithm`, and all four have variant features. Features having variant features represent subdomains of the system by factorizing the commonality between features of the system. For example, while the different types of algorithms could have been directly linked to `GraphLibrary`, they have been grouped under an `Algorithm` feature. While at least one feature in an Or group needs to be selected (*e.g.*, if the `Search` variant is selected, at least one of `BFS` and `DFS` also needs to be selected), exactly one feature can be selected in an Alternative group (*e.g.*, the `EdgeType` can be `Directed` or `Undirected` but not both). Additionally, cross-tree constraints define conditions on features constraining their selection and are commonly expressed using propositional logic. For example, the selection of the `Cycle` algorithm implies the selection of the `Directed` type of edges as this algorithm cannot be applied to graphs with undirected edges.

The realization of the variability in the code assets is usually separated into three parts: core, commonalities, and variabilities. The core corresponds to the implementation that remains when no feature is selected [Turner et al., 1999]. The commonalities represent the common parts in the implementation of each feature of a subdomain. When factorized, these commonalities become part of the core, unless the implementation is related to an optional feature. Finally, the variabilities represent the implementation specific to each variant. The implementation of the variability is

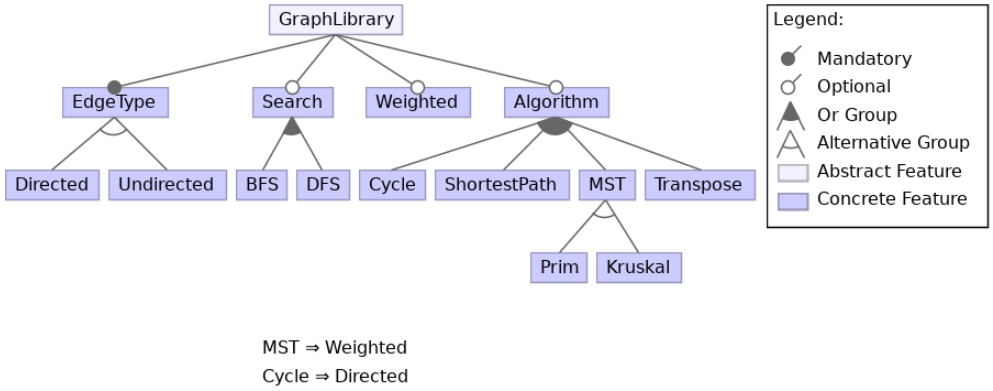


Figure 2.1: Feature model of a library manipulating graphs

often abstracted in terms of variation points (*vp*-s) and variants. A *vp* is defined by *"one or more locations at which the variation will occur"* [Jacobson et al., 1997], while the variants characterize how this *vp* varies. Multiple techniques allow their implementation.

2.2 Variability implementation techniques in large software systems

2.2.1 Usual variability implementation techniques

The implementation of the variability can be done in numerous ways. In a *clone-and-own* approach [Rubin et al., 2013], variability is achieved by duplicating the code assets and adapting them to create a new variant. In practice, this technique is often achieved by managing forks of an original codebase repository [Li et al., 2016; Businge et al., 2018]. Each product variant has a dedicated codebase containing the commonalities (*i.e.*, the code common to all products) and its variabilities (*i.e.*, the code specific to this variant). This technique, however, impedes the maintainability of the system as it evolves. As the core is duplicated for every product, every modification to those assets (being implementation, tests, specification...) needs to be replicated in all the variants, which can become cumbersome with an important number of variants. Additionally, the difficulty increases with time as each variant has its own evolution pace [Pate et al., 2013]. Consequently, software practitioners now increasingly rely on other techniques allowing them to manage variability in a single codebase.

On their side, annotative approaches allow incorporating features traces in a textual way in the code assets to indicate the features they implement. Such techniques are used to delimit code blocks to be selected when generating the variant, as with CPP annotations [Kernighan and Ritchie, 1988], often called `ifdefs`. While `ifdefs` are heavily used in C based systems like the Linux kernel [Hunsen et al., 2016; Le et al., 2013; Liebig et al., 2010; Tartler et al., 2012; Sincero et al., 2010], annotations are also used to reengineer variable software implemented in other languages, independently of their paradigm, in SPLs [Couto et al., 2011; Martinez et al., 2017a]. Although their widespread use is due to the easiness of their implementation, extensive use of such techniques in a code is seen as a "pollution" [Liebig et al., 2010]. As they can be uniformly used to manage variability at various levels, the overabundance of annotations in the code hampers the

understanding and management of both the system and the implemented variability, eventually becoming a "hell" [Le et al., 2011; Medeiros et al., 2017].

Other paradigms have therefore been designed with the main objective to prevent such entanglement. The feature-oriented programming (FOP) [Prehofer, 1997; Apel et al., 2005, 2013] paradigm aims to organize the implementation in feature modules [Apel et al., 2006; Takeyama and Chiba, 2013] and obtain a strict alignment between the implementation *vp*-s with variants and the domain commonalities and variabilities respectively. For example, aspect-oriented programming [Kiczales et al., 1997; Morin et al., 2009; Parra et al., 2011] can be used in an FOP context by implementing the variabilities in the aspects [Mezini and Ostermann, 2004; Kastner et al., 2007; Figueiredo et al., 2008] that will then be merged with the core implementation in a preprocessing step. While aspects contain actual implementation code, deltas in delta-oriented programming [Schaefer et al., 2010; Clarke et al., 2010] define patches (additions, deletions or modifications of assets at different granularities) that are applied to the core implementation when compiling the code assets for the desired variant. Although such techniques present the advantage of allowing a clear separation between the code assets implementing the different features, they cannot be applied to codebases for which variability is undocumented, which is often the case of OO systems [Krüger et al., 2019b].

2.2.2 Variability in code assets of an object-oriented system

Although the previously cited approaches are applicable OO systems, in practice, they often rely on OO mechanisms to achieve variability implementation [Coplien et al., 1998; Gacek and Anastasopoules, 2001; Patzke and Muthig, 2002; Svahnberg et al., 2005], namely inheritance, overloading of methods and constructors, and design patterns such as Strategy and Factory [Gamma et al., 1993]. Listing 2.1 shows an illustrative example of implementation in Java for two different types of geometric shapes, `Circle` and `Rectangle`. The abstract class `Shape` factorizes the behavior common to `Rectangle` and `Circle`, relying on inheritance as a variability implementation technique [Jacobson et al., 1997; Coplien et al., 1998]. The `Point origin` attribute and the `newOrigin` method are accessible by both subclasses, and the two abstract methods `area` and `perimeter` are overridden in each subclass, defining their own way to compute these values. Additionally, in the `Rectangle` class, the two `draw` methods lines 13–16 and 18–21 allow implementing two ways of drawing a rectangle, depending on how the point's coordinates are passed as a parameter (*i.e.*, as two integers or encapsulated in a `Point` object).

Relying on the definitions of *vp*-s and variants given in Section 2.1, the behavior implemented in the `Shape` class being commonly shared by its subclasses, it is characterized as a class level *vp* with two variants being its subclasses `Rectangle` and `Circle`. Analogously, two methods named `draw` have a common name but different parameters and bodies. These methods are therefore variants of a method level *vp* called `draw` [Tërnava et al., 2019].

In this thesis, we set our focus on large OO variability-rich systems that are not architected as SPLs and progressively implement their variability in a single codebase using OO mechanisms.

2.3 Variability-aware build systems

Build systems are responsible for *"scheduling and executing all build-related tasks, which may include running generators, compiling source code, running tests, and creating and copying deliverable units"* [Apel et al., 2013]. In its simplest version, a build system can be a simple script

```java
/* Class level variation point, vp_Shape */
public abstract class Shape {
  // Point defined
  private Point origin;
  // Constructor omitted
  public void newOrigin(Point o) {
   origin.setPoint(o.getX(), o.getY());
  }
  public abstract double area();
  public abstract double perimeter();
}
```

```java
/* First variant, v_Circle, of vp_Shape */
public class Circle extends Shape {
  private final double radius;
  // Constructor omitted
  public double area() {
    return Math.PI * Math.pow(radius, 2);
  }
  public double perimeter() {
    return 2 * Math.PI * radius;
  }
}
```

```java
/* Second variant, v_Rectangle, of vp_Shape  */
public class Rectangle extends Shape {
  private final double width, length;
  // Constructor omitted
  public double area() {
    return width * length;
  }
  public double perimeter() {
    return 2 * (width + length);
  }
  /* Method level variation point, vp_Draw  */
  /* Variant v_drawCoordinates of vp_Draw  */
  public void draw(int x, int y) {
  // rectangle at (x, y, width, length)
   System.out.println("Rectangle at (" + x + ", " + y + ")");
  }
  /* Variant v_drawPoint of vp_Draw  */
  public void draw(Point p) {
  // rectangle at (p.x, p.y, width, length)
   System.out.println("Rectangle at (" + p.getX() + ", " + p.getY() + ")");
  }
}
```

Listing 2.1: Example of variability implementations. The `vp_Shape` and `vp_draw` represent two *vp*-s at the class and method levels, respectively.

executing build tools. However, as they manage which assets are built and how, they are also often in charge of managing compile-time variability [Apel et al., 2013]. When they manage variability, they allow resolving it by relying on command-line parameters or by reading from a configuration. While systems that are fully engineered as SPLs would rely on tools integrated by the modeling frameworks [Meinicke et al., 2017; Beuche, 2019; Krueger and Clements, 2013], other variability-intensive systems often design their build system by relying on existing tools. For example, Make[1] is a tool designed to control the build of an executable. The build process is described in the form of Makefiles that are then executed to build the system. In the features it proposes, Make allows to conditionally select code assets to build variants of the executable. It is therefore a tool that allows the implementation of variability at compile-time. In their simplest form, variability-aware build systems consist of one such tool. However, when the size and complexity of a system increase, they often rely on multiple of these tools. For example, the files selected by the Makefiles may contain variability implemented using `ifdef` directives that are resolved at the compilation of the sources invoked by the build system.

[1] https://www.gnu.org/software/make/

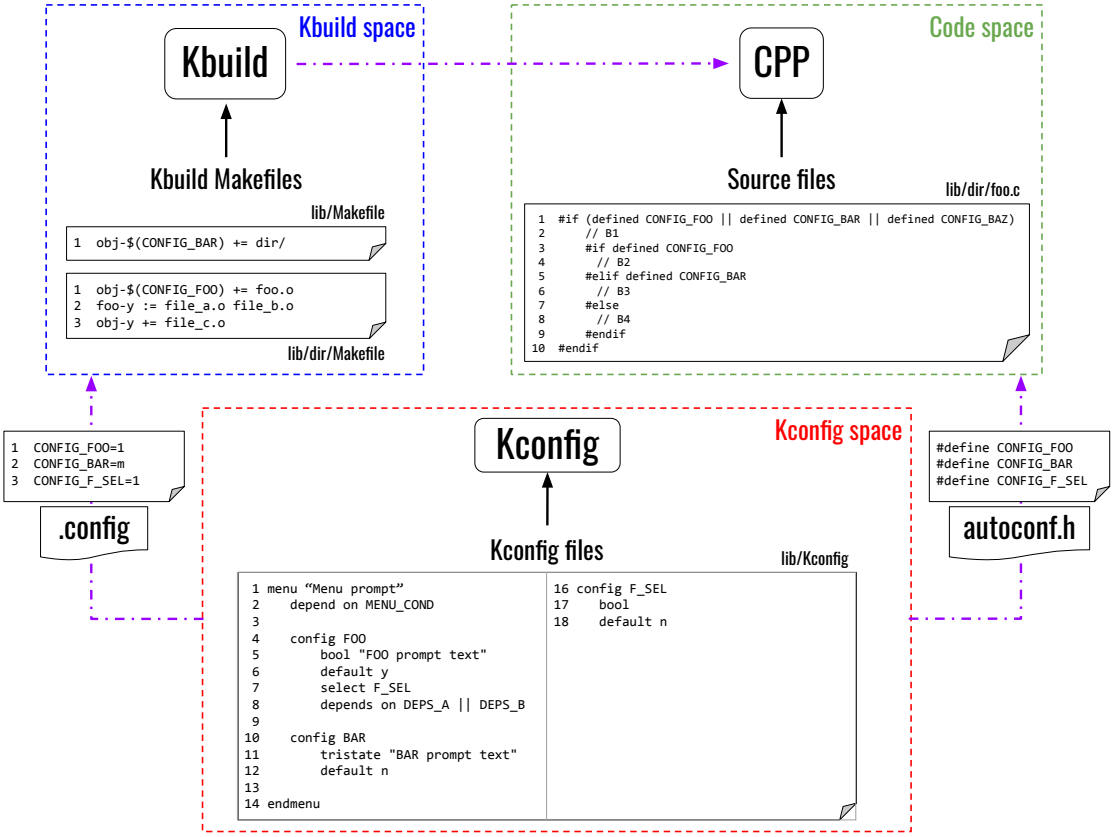


Figure 2.2: Linux build process. Violet dotted arrows represent constraints propagation between spaces.

In this thesis, we set our focus on variability-aware build systems reusing off-the-shelf tools to manage variability at multiple granularities in several steps. One example of such build systems is the Linux kernel build system.

2.3.1 The Linux kernel build system

The widespread use of the Linux kernel in a variety of contexts and, consequently, platforms (*e.g.*, servers, personal computers, smartphones running the Android operating system[2]) leads to numerous specific implementations to support this diversity of ecosystems. The Linux kernel build system is responsible for the configuration of the kernel (*i.e.*, selection of the desired features) and the selection of the respective code assets that are then compiled to build the final bootable kernel. This build system has three steps, illustrated in Figure 2.2.

KCONFIG KCONFIG files are present in multiple directories of the codebase and define configuration options (also called *symbols*) representing features. Each configuration option is defined as a `config` entry and can be of six different types: `bool`, `tristate`, `string`, `hex`, or `int`.

[2] https://www.android.com/

A default value for the feature can be set with the `default` entry. Features can be selected directly by the user via a prompt (present in an individual `prompt` entry or attached to the type of the feature), or by constraints on other features (defined in a `depends on` entry). Menus allow grouping features. If a feature is defined within a `menu` item that itself has a `depends on` entry, this condition is appended by KCONFIG to the `depends on` condition of the feature[3]. A feature can also force the selection of another feature with the `select` entry. For example, in the `lib/Kconfig` file presented in Figure 2.2, feature `FOO` (line 4) is a feature of type `bool` whose default value is `y` but which can be modified by the user via a prompt. To be selected, `DEPS_A` or `DEPS_B` need to be selected, and `MENU_COND` needs to be satisfied. The selection of `FOO`, will also force the selection of `F_SEL`. KCONFIG checks for the consistency of the constraints between the selected features and outputs two files containing the list of selected features in two formats: `.config` will be read by the KBUILD Makefiles, and `autoconf.h` is a C header file that will be appended to every source file during compilation. The KCONFIG is often considered by the scientific community as a variability model [Sincero and Schröder-Preikschat, 2008; Oh et al., 2019] as the `depends on` and `select` statements implement behavior analog to cross-tree constraints between features (*cf.* Section 2.1).

KBUILD The KBUILD system is made of multiple Makefiles present in multiple directories throughout the project, which select objects for the compilation. Three types of objects exist: object files, directories, and composite objects. Object files (such as `file_c.o` in `lib/dir/Makefile`) represent objects generated during the compilation from existing `.c` files in the codebase. Therefore, a `file_c.c` file should be present in the codebase. Added directories (such as `dir/` in `lib/Makefile`) will have their KBUILD Makefile parsed to select files from this subtree. Composite objects associate multiple files in one single object. For example, `foo.o` in `lib/dir/Makefile` is a composite object defined at line 2 combining `file_a.o` and `file_b.o` and used at line 1.

Selection is done by adding the object files generated at the precompilation to lists. For example, in `lib/dir/Makefile`, the `file_c.o` object is added to the `obj-y` list. In this case, the object will always be selected. The selection of an object can also be conditioned by the value of a feature, as for the `foo.o` object. `CONFIG_FOO` refers to the `FOO` feature defined in the KCONFIG file `lib/Kconfig`. `FOO` is a boolean feature, therefore if it has for value `y`, the object will be added to the `obj-y` list. The same mechanism applies for the `dir` directory in `lib/Makefile`, with the small difference that `BAR` is a `tristate` feature, allowing an extra `m` value. The object added to the `obj-m` list will be compiled as a module. If a feature is not defined, the name of the list becomes `obj-` and is ignored.

CPP Variability in the source files is implemented using CPP directives. Code in conditional blocks declared with `#if`, `#elif`, `#ifdef`, or `#ifndef` directives (referred to as `ifdef` directives) is selected only if the condition of the directive is satisfied. For example, in `lib/dir/foo.c`, the selection of `B1` implies that the condition line 1 is true. A nested block can only be selected if its parent block is selected (the selection of `B1` implies that the condition line 3 is true and that `B1` is also selected). Finally, code defined in a block declared with `#elif` or `#else` can only be selected if the `ifdef` blocks preceding it are not selected (the selection of `B3` implies that the condition line 5 is true and that `B2` is not selected, and the selection of `B4`

[3] https://www.kernel.org/doc/html/latest/kbuild/kconfig-language.html#menu-structure

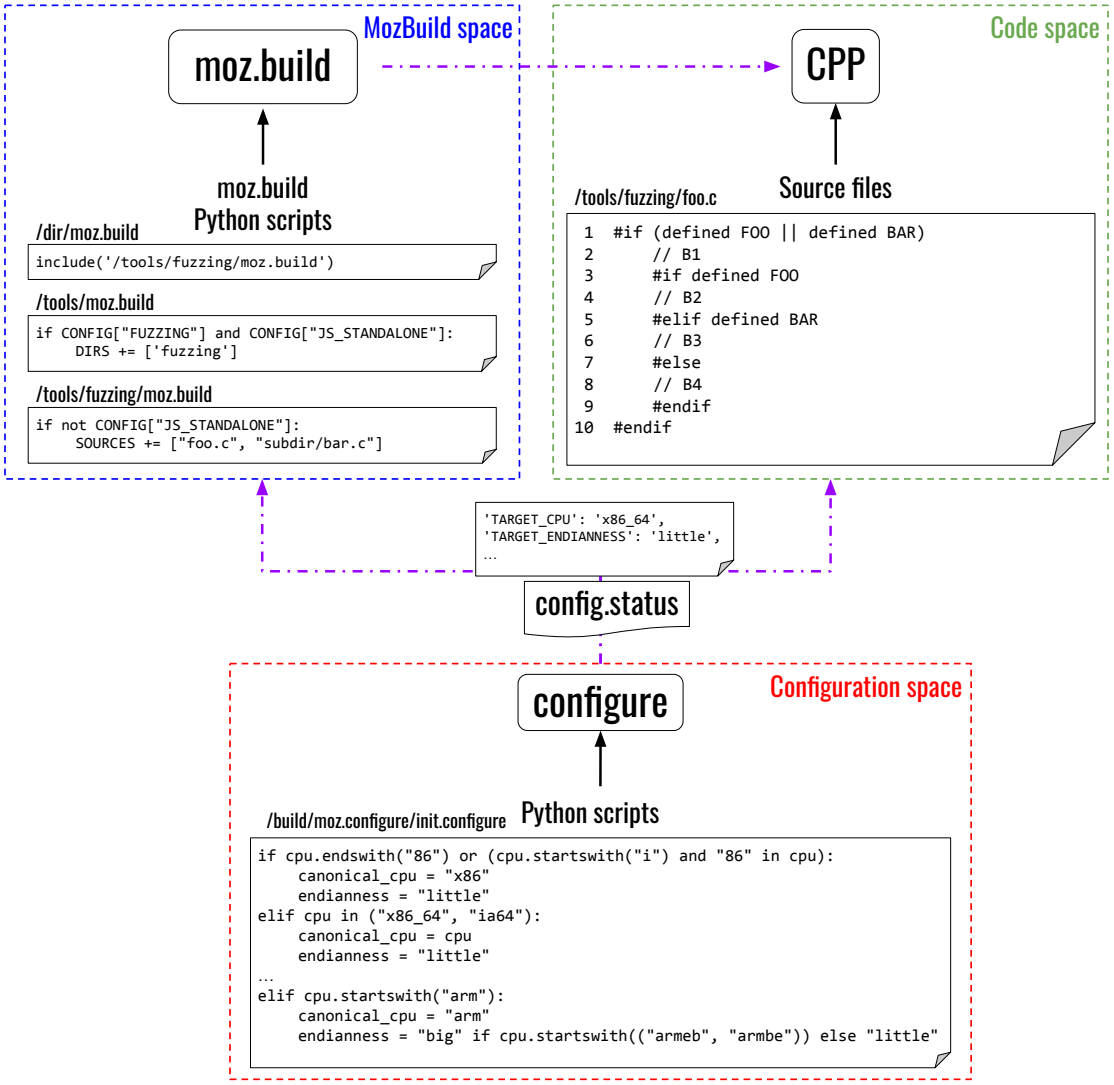


Figure 2.3: Mozilla build process.

implies that neither `B2` nor `B3` is selected). Naturally, at the build system level, the selection of a code block is also conditioned by the selection of the source file containing it, itself depending on the selection of its parent directory by the KBUILD.

In the Linux kernel, the management of the variability is therefore done in three steps: a hand-crafted configuration step, that has then been reused in other build systems [Fernandez-Amoros et al., 2019; Oh et al., 2019], followed by two variability resolution steps reusing and adapting off-the-shelf tools for that purpose. The Linux kernel is however not an isolated case, and similar build systems reuse other approaches, such as the Mozilla build system.

2.3.2 The Mozilla build system

The codebase for the Mozilla Gecko rendering engine is split in two repositories. `mozilla-central`[4] contains all the implementation for the Mozilla Firefox web browser[5], the Android Firefox application[6] and the SpiderMonkey suite[7], while `comm-central`[8] contains the code specific to the Thunderbird mail client[9] and the SeaMonkey suite[10]. Analogously to the Linux kernel, these products are used on a plethora of platforms and operating systems[11], and a build system is in charge of determining the configuration of the system that will run the product and selecting the respective code assets. The Mozilla build system is composed of three distinct steps, illustrated in Figure 2.3.

configure A `configure.in` file processed with `Autoconf`[12] generates the `configure` script. This script, together with a set of Python scripts, analyzes the host system to extract information and create the `config.status` file containing the list of configuration options used to configure Gecko and their values. The sample script shown in the *Configuration space* box (Figure 2.3) is an excerpt of one of the Python scripts determining the features used in the build system, `build/moz.configure/init.configure`[13], and exhibits the definition of the `TARGET_CPU` and `TARGET_ENDIANNESS` options relying on the information given by `Autoconf`. In the showcased example, the `x86_64` value returned by `Autoconf` leads to `TARGET_CPU` being set to `x86_64` and `TARGET_ENDIANNESS` to `little` in the `config.status` file.

MOZBUILD The MOZBUILD system consists of multiple `moz.build` files written in Python disseminated in multiple directories of the project, whose goal is to select source files for compilation (and determine other build parameters such as compilation flags or target libraries) based on conditions on features from the configuration[14]. The codebase possesses multiple root MOZBUILD files for each of the different software products that can be derived from the codebase.

Symbols represent lists or constants. Depending on their nature, the code assets are added for compilation in different list symbols. C++ source files are added to the `SOURCES` list, as shown in the `/tools/fuzzing/moz.build` file on Figure 2.3, while C compilation flags are added to the `HOST_CFLAGS` list. If the `JS_STANDALONE` feature is not set, the `foo.c` file in this directory and the `bar.c` in the `subdir` subdirectory relatively to the `moz.build` file will be selected for compilation. Adding a directory path to the `DIRS` list indicates to the MOZBUILD to parse and evaluate the `moz.build` file in this directory. For example, in `/tools/moz.build`, if both `FUZZING` and `JS_STANDALONE` features are set, the `/tools/fuzzing/moz.build` file will be evaluated. Another way to select directories is to use the `include` directive that

[4] https://hg.mozilla.org/mozilla-central/
[5] https://www.mozilla.org/en-US/firefox/
[6] https://www.mozilla.org/en-US/firefox/browsers/mobile/android/
[7] https://spidermonkey.dev/
[8] https://hg.mozilla.org/comm-central/
[9] https://www.thunderbird.net/en-US/
[10] https://www.seamonkey-project.org/
[11] https://data.firefox.com/dashboard/hardware
[12] https://www.gnu.org/software/autoconf/
[13] https://hg.mozilla.org/mozilla-central/file/a66dcaea419641c5483a43aa7f577b70908d147f/build/moz.configure/init.configure#l503
[14] https://firefox-source-docs.mozilla.org/build/buildsystem/mozbuild-symbols.html

allows to include other `moz.build` files in unrelated directories. For example, evaluating `/dir/moz.build` will also lead to the evaluation of `/tools/fuzzing/moz.build`.

Therefore, as opposed to the kernel's KBUILD Makefiles, the MOZBUILD files do not only manage files in their own directories, and multiple inclusions of a same source file of `moz.build` file thus becomes possible. This is the case for `/tools/fuzzing/moz.build` that is always evaluated whenever `/dir/moz.build` is, but is included with a specific condition in `/tools/moz.build`.

CPP The `config.status` file is also used as input to CPP to preprocess the source files and select code blocks. The CPP stage is similar as in the Linux kernel build system and is already described in Section 2.3.1.

The steps to build a product from the Mozilla codebase are therefore similar to the ones building a Linux variant. Moreover, although the tools used are different (apart from CPP), they are in both cases solutions built by adapting existing tools that are originally not designed to manage variability. This causes each step to be completely independent and unaware of the variability implemented by the others, and any modification regarding the variability in a step can have unexpected consequences in others. For example, in the KCONFIG file presented on Figure 2.2, adding a `depends on !FOO` constraint to the `BAR` feature would prevent the selection of the `foo.c` file in the `lib/dir/Makefile` (line 1) as entering the `lib/dir` directory implies that `BAR` is selected. Such problems are called *anomalies* [Sincero et al., 2010; Nadi et al., 2013]. In this case, the file (and consequently the code blocks it contains) are never selectable for any variant of the system and are therefore called *dead* files/blocks. While a variability modeling tool having the view of the overall variability would detect the problem, build systems made of ad hoc tools, such as the Linux kernel build system and the Mozilla build system do not have this capacity.

CHAPTER 3

State of the Art

In Chapter 2, we introduced the main concepts related to software variability and its implementation in code assets, with a particular focus on OO systems. We also presented how variability-aware build systems adapt tools to manage variability at compile-time. In this chapter, we provide an overview of the state of the art on work providing approaches related to the challenges tackled in this thesis. Proposed techniques to identify variability implementations (Challenge A1) are detailed in Section 3.1, while tools and visualizations for variability and software comprehension (Challenge A2) are presented in Section 3.2. Work on the characterization and the comprehension of quality and its relationship with variability implementations (Challenge A3) is detailed in Section 3.3. Finally, contributions studying the derivation mechanism of build systems (Challenge B1) and characterizing the anomalies they can suffer from (Challenge B2) are showcased in Sections 3.4 and 3.5 respectively.

3.1 Variability identification techniques

Multiple techniques have been proposed to identify variability implementations in the code assets of software systems that are not fully-fledged SPLs. We first present a panel of proposed feature location and identification techniques (Section 3.1.1) before focusing on an approach specifically designed for OO systems in a single codebase (Section 3.1.2).

3.1.1 Feature location and identification techniques

Multiple approaches have been proposed to reengineer legacy software systems in an SPL [Assunção et al., 2017] and are commonly divided into two categories. On one side, feature location approaches aim to recover the traceability of some pre-existing features to the reusable code assets in an SPL [Rubin and Chechik, 2013; Dit et al., 2013; Krüger et al., 2019a; Michelon et al., 2022]. However, as the domain variability is hardly documented in OO variability-rich systems [Krüger et al., 2019b], such approaches are not applicable to our context. On the other side, feature identification approaches aim to identify the common and varying units, as potential features, among some related software systems [Ziadi et al., 2012; Martinez et al., 2017b]. While they do not rely on domain features, they often target clone-and-own systems [Roy and Cordy, 2007; Michelon et al., 2019; Linsbauer et al., 2022], thus managing their variability in multiple codebases, or systems relying on `ifdefs` [Sincero et al., 2010; Liebig et al., 2010] or other annotative approaches [Couto et al., 2011]. When applicable to single OO codebases, they rely on dynamic analysis results such as traces obtained from executing the system [Walkinshaw et al., 2007] or the unit tests [Eisenberg and De Volder, 2005] that can be used to enhance static analysis results [Michelon et al., 2021a].

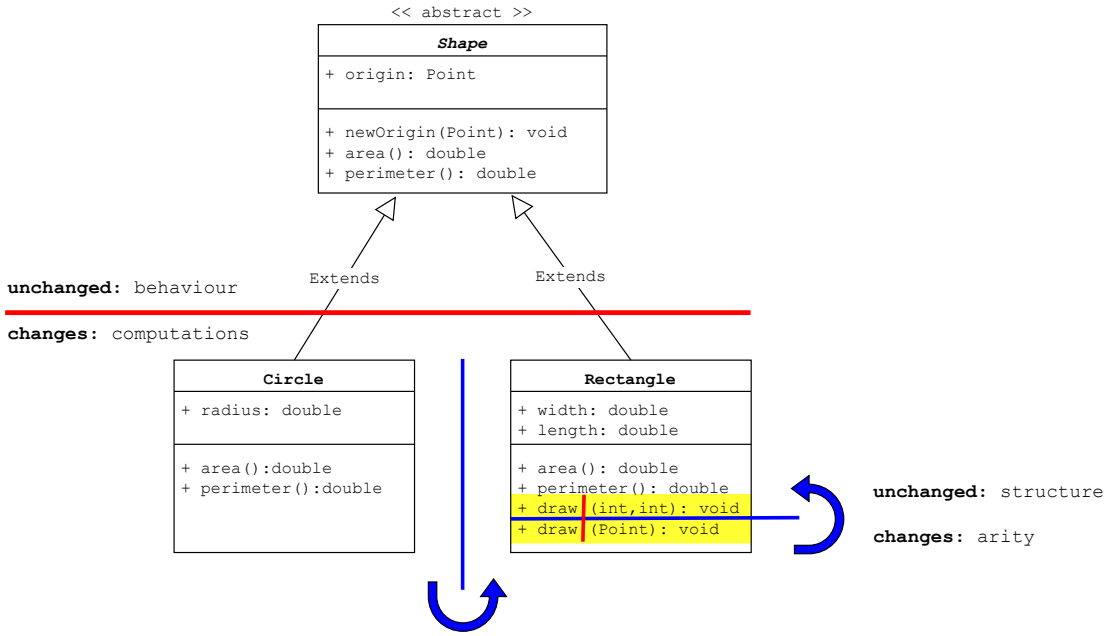


Figure 3.1: UML representation of Listing 2.1 exhibiting local symmetries

Table 3.1: Five object-oriented software constructs and their symmetries

Software construct	Commonality / Unchange	Variability / Change	Symmetry transformation
Class subtyping	Superclass / Type	Subclasses	Substitution
Method/constructor overloading	Structure	Signatures / Arity	Substitution
Strategy pattern	Strategy interface	Algorithms	Substitution
Factory pattern	Abstract Creator and product	Concrete creators and products	Factory

As we aim to tackle large systems that are possibly not executable locally and since unit tests are not always available and executable, we restrain ourselves to a static analysis of the code assets.

3.1.2 Identifying OO variability implementations

Considering this lack of approach to tackle the problem of statically identifying variability implemented using OO mechanisms, Tërnava et al. [2019] proposed an identification approach relying on the notion of symmetry in OO code assets and proposed *symfinder* [Mortara et al., 2019], a toolchain automating the identification on single Java codebases using this technique[1]. We detail both the identification technique and the toolchain hereafter.

[1]During my Master's thesis, I collaborated with Xhevahire Tërnava and designed the *symfinder* toolchain as an automation of the symmetry-based identification approach she designed. Consequently, although I am the first author of the paper on *symfinder* toolchain [Mortara et al., 2019], Xhevahire is the first author of the paper describing the identification approach [Tërnava et al., 2019].

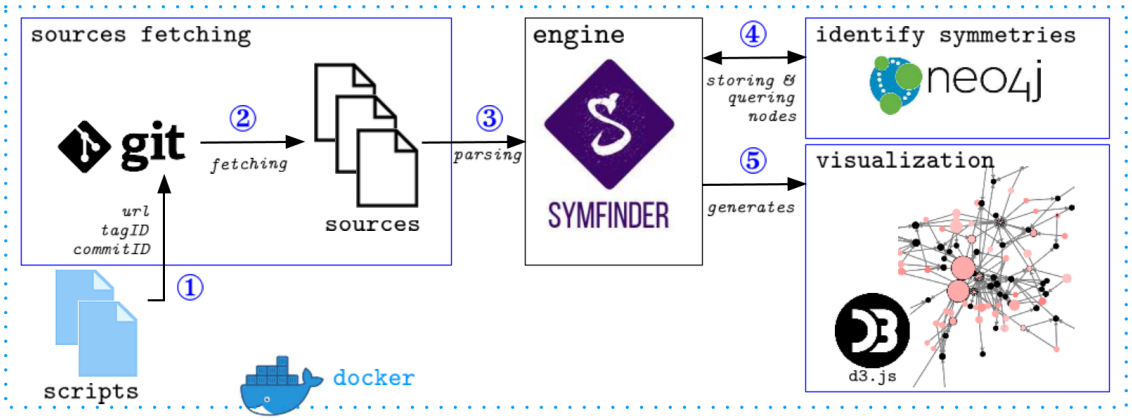


Figure 3.2: The *symfinder* toolchain

Symmetry in OO software constructs Inspired by Alexander's theory of centers [Alexander, 2002], several works show that OO techniques and software design patterns exhibit a form of symmetry [Coplien and Zhao, 2000a; Coplien, 2001; Zhao and Coplien, 2003, 2002; Henney, 2003; Zhao, 2008]. From the natural sciences, the symmetry of an object is defined as *the immunity to a possible change* [Rosen, 1995, 2008] and relies on (1) the possibility of change and (2) the immunity to change. Considering a whole codebase, the OO techniques involved in variability implementations can be seen as local symmetries [Tërnava et al., 2019; Tërnava et al., 2022], which allow a part of the code to change while another part remains unchanged.

To illustrate these symmetries, let us consider the UML diagram given in Figure 3.1, representing the example code given in Listing 2.1. The class `Shape` is a class level *vp* with two variants, `Circle` and `Rectangle`. They can be abstracted as `vp_Shape`, `v_Circle`, and `v_Rectangle`, respectively. Following the symmetry definition, inheritance defines a *substitution symmetry* for its subtypes. Here, the *possibility of a change* in the superclass `Shape` materializes in its potential different subtypes, such as `Circle` and `Rectangle`, which vary regarding the type of geometric shape. Still, they also *preserve* and conform to the common behavior of their superclass. Analogously at the method level, the two `draw` methods exhibit a substitution symmetry as their signatures and parameters vary, while their structure remains immune to the change. Table 3.1 illustrates all five mechanisms allowing to implement variability detailed in Section 2.2.2 the common and variable parts of the symmetry they exhibit. Additionally, as the coherence of a structure or object is related to a density of local symmetries [Alexander and Carey, 1968; Alexander, 2002], the density of *vp*-s with variants has been proposed as a way to locate and describe the most intense places concentrating variability implementations in a system.

symfinder The proposed identification technique relying on symmetries and their density has been implemented in a tooled approach, *symfinder*, providing an automatic identification and visualization of potential *vp*-s with *variants* in code assets of a Java-based variability-rich system [Mortara et al., 2019]. The organization of the toolchain is depicted in Figure 3.2. After fetching the sources of the Java system to analyze, the code assets are parsed and the potential *vp*-s with variants identified, relying on a graph representation of the assets and their identified symmetries

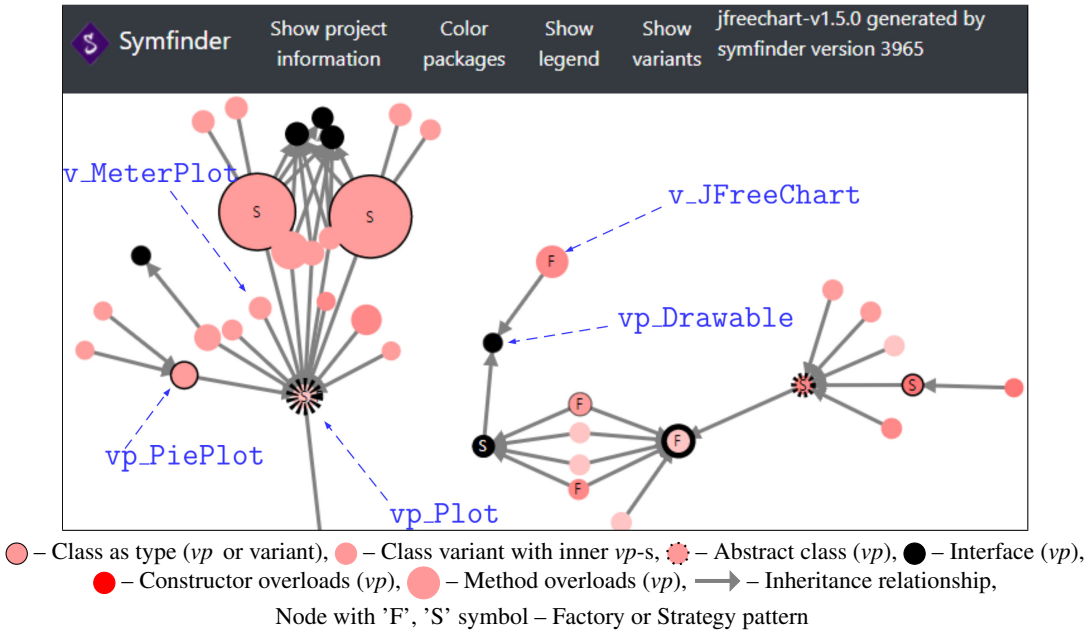


Figure 3.3: Identified *vp*-s and variants using *symfinder* for the excerpt of JFreeChart given in Figure 3.1.

in a Neo4j database[2]. *symfinder* also provides a visual representation of the potential *vp*-s and variants and their metrics (*i.e.*, for each class level *vp*, the number of method level *vp*-s with the number of variants). Figure 3.3 shows the visualization of identified potential *vp*-s with variants[3] in JFreeChart[4], a charting library written in Java. Each class level *vp* or variant is visualized as a circle node that points out the used implementation technique while edges show their inheritance relationship. The size and shades of the red color of nodes indicate the number of method level *vp*-s with variants[5]. Consequently, the visualized variability forms a disconnected graph based on inheritance links, while the visual representation of the metrics associated with each node (*e.g.*, size, color intensity) creates zones that can be easily distinguished by their different density of symmetries[6]. For example, the left part of Figure 3.3 is denser than the right part, indicating that these classes concentrate more mechanisms involved in variability implementations.

Tërnava et al. [2019] evaluated this identification technique by applying it on eight Java open source systems and visually assessing the relevance of the identified zones concentrating *vp*-s with variants. As the authors were able to discern relevant variability implementations, *symfinder* and its symmetry-based identification technique seem to be a first answer to Challenge A1. However, this evaluation exhibits two limitations. First, the applicability of the symmetries in OO constructs has been demonstrated in a single language, Java. Second, the relevance of the identified *vp*-s and

[2] https://neo4j.com/

[3] Blue names and arrows in Figure 3.3 have been manually added for illustration.

[4] https://www.jfree.org/jfreechart/

[5] `PiePlot` is a variant of `Plot`, however, it is also a *vp* as it has two variants, thus is described as `vp_PiePlot` on the figure.

[6] It should be noted that the identification is based on symmetry in implementation techniques while the visualization on their density. Hence, one should not confuse the two and expect any kind of symmetry in the visualization.

variants has been done visually, based on the authors' knowledge of these systems. Therefore, confirming the hypothesis according to which *symfinder* answers Challenge A1 requires assessing the method on these aspects.

3.2 Helping the comprehension of the implemented variability

Multiple approaches have been proposed to help the comprehension of variability-intensive software systems, heavily relying on visualizations.

Tool supports and visualizations for SPLs. When such systems are developed as SPLs, multiple tools help the implementation and the management of the variability, such as FeatureIDE [Meinicke et al., 2017], pure::variants [Beuche, 2019; pure-systems GmbH, 2020] or BigLever Gears [Krueger and Clements, 2013]. These tools rely on visual representations to assist the user. For example, pure::variants provides a visualization of the realized variability into a *family model*, including code assets by using a hierarchical "file explorer style", iconography for types of elements and "feature" states, and a matrix view. Other visual approaches have been proposed to help the comprehension of the variability of a variability-intensive systems but mainly focus on SPLs and feature models. They aim to facilitate the configuration process over features [Lopez-Herrejon et al., 2018], or assist the onboarding of developers on SPLs [Azanza et al., 2021], and therefore do not concern the implementation.

Visualizations for variability implementations. Regarding variability at the code level, multiple works rely on integrated development environments (IDEs) to visualize feature traces [Kästner et al., 2008; Andam et al., 2017; Martinson et al., 2021] or `ifdefs` [Feigenspan et al., 2011a, 2013] in the code assets and help to understand the interactions between features and code [Greevy et al., 2005; Bergel et al., 2021]. Other approaches target software clones [Hammad et al., 2020] and provide visualization approaches to measure the similarity that exists between them, using for example bar charts [Duszynski and Becker, 2012]. These techniques targeting *clone-and-own* systems or requiring features traces or annotations, they are therefore not applicable to our case.

Although *symfinder* already proposes a graph visualization for the identified OO variability implementations (*cf.* Section 3.1.2), the 2D nature of the visualization limits the information on the variability that can be displayed. Additionally, the challenge of tackling large codebases requires a scalable visualization, which is known to be a weak point of graph visualizations [Pienta et al., 2015]. Hence, to the extent of our knowledge, no scalable visualization for OO variability implementations has yet been proposed, leaving Challenge A2 open.

The city metaphor. While, it could be possible to make *symfinder*'s graph visualization evolve and improve its scalability [Burch et al., 2011; Beck et al., 2017], multiple visualizations rely on metaphors as they bring an understandable graphical representation to concepts [Knight and Munro, 2000]. For example, the metaphor of the city [Knight and Munro, 1999] has been applied to multiple types of metrics on software systems: dynamic behavior (such as concurrency between classes [Waller et al., 2013] or memory consumption of heaps [Weninger et al., 2020]), and static properties such as dependency and communication links between components [Fittkau et al., 2017].

At a finer grain, software cities to understand OO software systems have been proposed, the first of them being CodeCity [Wettel and Lanza, 2008a, 2007] which uses buildings to represent classes, grouping them in districts representing packages. These principles were enhanced by adding a temporal dimension in the analysis to visualize the evolution of the metrics through multiple versions of the system, first in CodeCity [Wettel and Lanza, 2008b] and also in a more recent approach called M3TRICITY [Pfahler et al., 2020]. The Evo-Streets [Steinbrückner and Lewerentz, 2013] approach also aims at visualizing the evolution of the software but uses streets to represent the package decomposition (instead of nested boxes in CodeCity). Multiple approaches also reuse the city metaphor by adding other visual dimensions such as arcs between buildings [Dashuber et al., 2021; Dashuber and Philippsen, 2022a] or by adapting it to more immersive techniques, such as virtual reality for CodeCity [Moreno-Lumbreras et al., 2023] and VRCity [Vincur et al., 2017], or Minecraft for CodeMetropolis [Balogh and Beszedes, 2013].

The popularity of this metaphor and the fact that CodeCity showed to help complete program comprehension tasks [Wettel et al., 2010, 2011] lead us to the hypothesis that such a visualization adapted for OO variability implementations could help their comprehension, and therefore be a way to answer Challenge A2. However, while the graph representation provided by *symfinder* allowed the authors to visually identify relevant zones concentrating variability implementations [Tërnava et al., 2019], no evaluation of the help that density brings to comprehend the implemented variability has been conducted. This preliminary step is required before designing a new visualization.

3.3 Identifying and comprehending indebted variability implementations

Multiple approaches have been proposed targeting the observation and characterization of technical debt in the context of variable systems.

Quality of variability implementations. Implementing variability in a system is known to bring additional complexity [Galster et al., 2017]. For example, extensive use of annotative approaches such as CPP directives is seen as a "pollution" [Liebig et al., 2010] as it hampers the understanding and management of the implemented variability, eventually becoming a "hell" [Medeiros et al., 2017; Le et al., 2011].

Consequently, the maintainability of the system is threatened and its capacity to evolve hampered [Favre, 1996; Kröher et al., 2022], eventually leading to technical debt [Avgeriou et al., 2016; Li et al., 2015]. Monitoring quality being crucial for the maintenance and evolution of such systems [Martini and Bosch, 2015], technical debt due to variability implemented using annotative approaches has been largely studied, leading to new definitions (as variability bugs, being errors and warnings caused by `ifdefs` interactions [Abal et al., 2014; Mordahl et al., 2019]) and adaptations of standard code smells definitions [Fenske and Schulze, 2015; Souza et al., 2019] to consider variability mechanisms. By targeting annotative approaches, these definitions cannot be used directly to measure the quality of OO variability implementations.

Variability debt and OO systems. Very recently, Wolfart et al. [2021] defined variability debt as *"Technical debt caused by defects and sub-optimal solutions in the implementation of variability*

management in software systems". They studied 52 industrial case studies reporting technical debt issues on variable software systems with the following main results:

1. the absence of known variability implementation mechanisms is prone to cause artifact duplication, an increase of code complexity, and a "disappearance of links between implementation artifacts to business values" [Ebert and Smouts, 2003];
2. the lack of knowledge of the implemented variability, as well as the absence of traceability, causes variability debt;
3. variability debt mainly impacts source code artifacts;
4. variability debt causes an inability to systematically deal with customization and poor overall internal quality, complicating maintenance for the development team.

OO systems are directly reusing traditional mechanisms and variability code is then intertwined with the rest of the implementation code [Tërnava and Collet, 2017b] (*cf.* Section 2.2.2), providing no traceability with potential domain knowledge. They are therefore prone to variability debt in the source code artifacts.

Defining OO variability debt. Wolfart et al. [2021] introduced a catalog of ten forms of variability debt, detailing for each of them its cause(s), consequence(s), and concerned type(s) of artifacts. In the following analysis, these forms are written in italics.

As OO variability implementations rely solely on standard OO mechanisms, the availability of the source code is the only requirement to identify them. Finding *Code duplication* is therefore possible, as well as *System-level structure quality issues* in the implementation. Most often, tests sources are provided along with the source code, enabling identification of *Lack of tests*.

However, other information is not always available, especially in the case of open-source systems, such as the documentation, leaving aside *Out-of-date or incomplete documentation* and *Duplicate documentation*. Identifying *Architectural anti-patterns* needs information on the domain and the associated design choices (*e.g.*, we cannot say if a Strategy pattern has the desired behavior solely by analyzing its structure). Covering *Poor test of feature interactions* would require a list of features and their mapping with their implementations, which are often not available in our case, while covering *Old technology in use* and *Multi-version support* implies having information about the versions of the supporting language and used libraries. Finally, identifying *Expensive tests* implies determining whether test cases have been formally defined or not [Shah et al., 2013], thus requiring test cases definitions.

It results that relying on the source code and its tests, OO software systems are prone to *Code duplication*, *Lack of tests*, and *System-level structure quality issues* in the implementation. There is therefore a need for a technique to identify and comprehend these types of variability debt in the context of OO variability implementations.

Identifying and comprehending technical debt in OO software systems. In the domain of OO systems, technical debt is often measured relying on metrics related to the OO implementations such as the number of lines of code (LoCs) of a class or method, the quantity of duplicated code or the unit tests coverage. Multiple works focus on determining software quality metrics [McCabe, 1976; Kafura and Henry, 1981; Rosenberg and Hyatt, 1997; Fowler, 2018; Campbell, 2018; Misra

et al., 2018], measuring their evolution [Sato et al., 2007; Hecht et al., 2015], and validating the relevance of these metrics [Khan et al., 2007; Pantiuchina et al., 2018]. Tools have been developed to automatically analyze OO codebases and extract quality metrics [Lenarduzzi et al., 2018], such as SonarQube[7], one of the most frequently used open-source code analysis tools, adopted by more than 200K developer teams, including more than 250K public open-source projects on its cloud version SonarCloud[8]. Not only the metrics are extracted, but a set of customizable rules gives more precise insights into the defects detected, and how to correct them [Pellegrini et al., 2018; Lenarduzzi et al., 2020]. Finally, a set of plugins complete the tool to provide improved exploitation of the extracted metrics and improve their comprehension, such as advanced visualization solutions. One of them is SoftVis3D[9], which embeds the city-based CodeCity [Wettel and Lanza, 2008a] and Evo-Streets [Steinbrückner and Lewerentz, 2010] visualizations. As such visualizations have proven to help the comprehension of a system's quality [Wettel et al., 2011], multiple other city-based visualization approaches for quality have been proposed [Wettel and Lanza, 2008b; Fittkau et al., 2017; Pfahler et al., 2020].

It results that in the context of OO systems, technical debt is identified relying on OO metrics, while their comprehension is achieved through visualizations. However, none of the existing identification or visualization approaches consider the variability implemented in such systems, leading to Challenge A3 (*"Understanding the quality of the implemented variability"*) to be open. Variability debt being a particular type of technical debt, there is a need to (*i*) determine relevant OO metrics to identify OO variability debt and (*ii*) design an adapted visualization to comprehend it.

3.4 Studies conducted on build systems

As detailed in Section 2.3, multiple large software systems rely on build systems to manage their build process, as the Mozilla products codebase and the Linux kernel, and have been studied on multiple aspects.

3.4.1 Studies on the Mozilla build system

For example, the Mozilla build system has been used as a case study to evaluate the impact of having a build system on the maintenance effort [McIntosh et al., 2011]. Lampel et al. [2021] explore the causes of intermittent test failures in the system's continuous integration and Maudoux and Mens [2019] report on the application of a third-party build system, Tup[10], as a replacement of Make. Finally, de Jonge [2005] examined the structure of the build system step selecting directories and source files (managed at the time of this study with Autotools) and noticed that *"Mozilla uses a dedicated build system.* [...] *It requires a special directory layout.* [...] *As a consequence, one cannot easily integrate Mozilla's source directories or build process in other software systems."*, confirming the ad hoc and complex nature of Mozilla's build system. However, to the extent of our knowledge, no work studies this build system from the variability point of view.

[7] https://www.sonarqube.org/
[8] https://sonarcloud.io/explore/projects
[9] https://softvis3d.com/
[10] https://gittup.org/tup/

3.4.2 Studies on the Linux kernel build system

On the opposite, the impressive figures of over 15,000 configurable features, 28 million lines of code in more than 60K files, 900,000 commits from more than 2K authors [Larabel, 2020], the Linux kernel has been a constant subject of study for the software engineering community on a plethora of aspects including software evolution [Antoniol et al., 2002; Padioleau et al., 2008; Israeli and Feitelson, 2010] and maintenance [Jiang et al., 2013; Abal et al., 2014] issues. More specifically, it has become emblematic over the years in the variability research community [Sincero et al., 2007] as mastering its variability is still an open challenge [Thüm, 2020]. Consequently, it naturally became a center of interest for build system variability studies.

As detailed in Section 2.3.1, the variability management architecture of the kernel relies on a model-based configuration tool (KCONFIG), CPP preprocessor directives in the code, and a configuration-aware build system (KBUILD).

Studies on the KCONFIG. Naturally, variability management in the KCONFIG part was deeply investigated. While Sincero and Schröder-Preikschat [2008] established a first mapping between the KCONFIG language and feature modeling concepts, She et al. [2010] investigated the inverse mapping and built a model for the KCONFIG language constructs. She and Berger [2010] described the semantics of the KCONFIG language, used as a basis for multiple tools [Tartler et al., 2011; Kästner, 2016; She, 2013; El-Sharkawy et al., 2015]. Zengler and Küchlin [2010] achieved a translation of KCONFIG's constraints in a single logic formula, later reused with SAT-solving by Walch et al. [2015] to analyze the consistency of KCONFIG files, while Fernandez-Amoros et al. [2019] provide a translation to propositional logic. In his Master's thesis, Hengelein [2015] analyses defects in KCONFIG. As it can be seen as a feature model [She et al., 2010], it can have its defects, *i.e.*, dead features [Kang et al., 1990], or false optional features [Zhang et al., 2004]. Based on these previous works, CONFIGFIX has been proposed as a variability-aware tool to help the configuration of the kernel [Franz et al., 2021]. Besides, as the Linux kernel is a living ecosystem, the evolution of its variability model has also been extensively studied [Lotufo et al., 2010; Passos et al., 2013; Dintzner et al., 2017; Passos et al., 2018; Kröher et al., 2018c].

Studies on the variability implemented with CPP. Closer to the code, Sincero et al. [2010] defined *presence conditions* to identify inconsistencies in the constraints defined by `ifdef` directives in the kernel, and proposed an implementation with the UNDERTAKER toolchain. While Tartler et al. [2009] introduced the problem of inconsistencies between KCONFIG files and `ifdef` directives, they extended UNDERTAKER to add constraints from the KCONFIG files and identify inconsistencies between the two spaces [Tartler et al., 2011, 2012]. Other tools relying on presence conditions have been developed to reason on `ifdef` directives for type checking, such as TypeChef [Kenner et al., 2010], and use the Linux code base as a robustness trial [Kästner et al., 2011]. Santos and Santana de Almeida [2015] identified more than 36,000 inconsistencies in the Linux code assets with their checking technique between FM concepts and their translation using `ifdef` directives. El-Sharkawy et al. [2017] analyze the causes of *configuration mismatches* in the kernel, being cases where the build system behaves differently from the constraints expressed in the variability model.

Studies on the KBUILD. Multiple tooled approaches have also been proposed to parse, analyze, and reason on KBUILD Makefiles, as KBUILDMINER [Berger et al., 2010], MAKEX [Nadi and

Table 3.2: Terminologies used by a panel of work for each space of the Linux build system

Paper	KCONFIG files	KBUILD Makefiles	CPP / Source files
Tartler [2013]	Feature Modeling Configuration	Build system	Generator Preprocessor
Passos et al. [2013]	Variability Model	Mapping	Implementation
El-Sharkawy et al. [2017]	Problem space	Solution space	
Abal et al. [2014]	Problem space Model	/	Solution space Code
Nadi [2014]	Configuration space	Build space	Code space
Work characterizing anomalies			
Sincero et al. [2010]	Problem space Model	/	Solution space Implementation
Tartler et al. [2011]	Configuration space	/	Implementation space
Hengelein [2015]	Feature Modeling Configuration	Build system	Generator Preprocessor
Nadi and Holt [2011]	Configuration space	Compilation space	Implementation space
Nadi and Holt [2012]	Kconfig space	Make space	Code space
Tartler et al. [2009]	Model level	Generation level	Source code level
Tartler et al. [2012]	Configuration space	Implementation variant	Implementation space
Nadi and Holt [2014]	Kconfig space	Make space	Code space

Holt, 2014], GOLEM [Dietrich et al., 2012] (and its extension MINIGOLEM [Ruprecht, 2015]), and KMAX [Gazzillo, 2017]. Berger et al. [2010] analyzed the KBUILD Makefiles to extract a mapping between features and code assets in the shape of presence conditions on the features. Other tools for analyzing standard Makefiles have been applied to KBUILD files, such as MAKAO [Adams et al., 2007], which builds a dependency graph from them. This tool is used in more recent work on the identification of *unspecified dependencies* in Make-based systems, also applied to KBUILD [Bezemer et al., 2017]. More recent approaches use symbolic execution to recover build conditions in KBUILD files [Nguyen and Nguyen, 2020]. Finally, after studying the internal consistency of the KBUILD Makefiles through three types of defects, Nadi and Holt [2011] built a third extension of UNDERTAKER to add constraints from the KBUILD files and identify inconsistencies in the three spaces [Nadi and Holt, 2012].

However, all the contributions previously detailed focus on isolated steps of the Linux build system. While the KernelHaven [Kröher et al., 2018a,b] workbench aggregates multiple tools previously cited in a single framework and allows to analyze properties of the whole kernel build system and extract metrics, there is no representation of the interactions between the different steps of the build system, and each one of them is also characterized differently. Table 3.2 summarizes for multiple of the previously cited work the different terminologies used to refer to the different parts of the build system. Except for a journal extension [Nadi and Holt, 2014] and a

Ph.D. thesis [Tartler, 2013], every paper has its own terminology, and some of them even use multiple terminologies [Tartler, 2013; Hengelein, 2015; Abal et al., 2014; Tartler et al., 2011]. One paper [El-Sharkawy et al., 2017] groups the KBUILD and CPP in a single *Solution space*, denomination used by Abal et al. [2014] and Sincero et al. [2010] to refer only to the CPP constraints. This heterogeneity prevents the application of these contributions to other build systems as the Mozilla build system that, although they are equally complex, have been less studied, leaving Challenge B1 (*"Making explicit the derivation mechanism of build systems"*) open.

3.5 Anomalies in build systems

As for contributions related to variability management in build systems presented previously, existing work on the definition and identification of anomalies in build systems are focused on the Linux kernel. Eight of the contributions studying the build system introduced in Section 2.3 and presented in Table 3.2 characterize anomalies in the build system and provide definitions for them. Appendix A lists the anomalies defined in five of these works. We discarded the last three publications being journal extensions [Tartler et al., 2012; Nadi and Holt, 2014] or publications by the same authors [Tartler et al., 2009] that do not extend the characterizations of anomalies.

In the implementation, anomalies due to interfering CPP constraints have been characterized by Sincero et al. [2010] (Appendix A.1) and extended to consider the KCONFIG constraints by Tartler et al. [2011] (Appendix A.4). Relying on this last work, Hengelein [2015] analyzes the internal consistency of the KCONFIG and proposes definitions to describe these anomalies (Appendix A.2). Finally, Nadi and Holt [2011] (Appendix A.3) define anomalies happening inside the KBUILD and extended their work to consider all three spaces of constraints [Nadi and Holt, 2012] (Appendix A.5).

Although this panel of work allows covering anomalies in and between all spaces of constraints in the build system, as shown on Figure 3.4, all concepts and denominations are different among contributions, with the same properties being described with varied formalisms and sometimes different definitions, at different levels, or with no easy relationship between them.

By analysing the definitions of anomalies from the selected papers, we can pinpoint multiple elements bringing confusion. First, multiple definitions are redundant between papers, but their expression and their names differ. For example, *Dead block* defined by Sincero et al. [2010] in Anomaly 1, *Configurability defect* defined by Tartler et al. [2011] in Anomaly 13 and *Code–KCONFIG anomalies* defined by Nadi and Holt [2012] in Anomaly 19 express the same formula. On the opposite, some anomalies with identical names may not express the same type of defect. This is the case for the formulas to detect *dead* blocks in Anomalies 13 and 15, which are equivalent, while the characterizations of *undead* blocks are inconsistent. Using the example shown

```
1 #if defined A
2     //block 1
3 #   if defined A
4         //block 2
5 #   endif
6 #endif
```

Listing 3.1: Two nested CPP code blocks

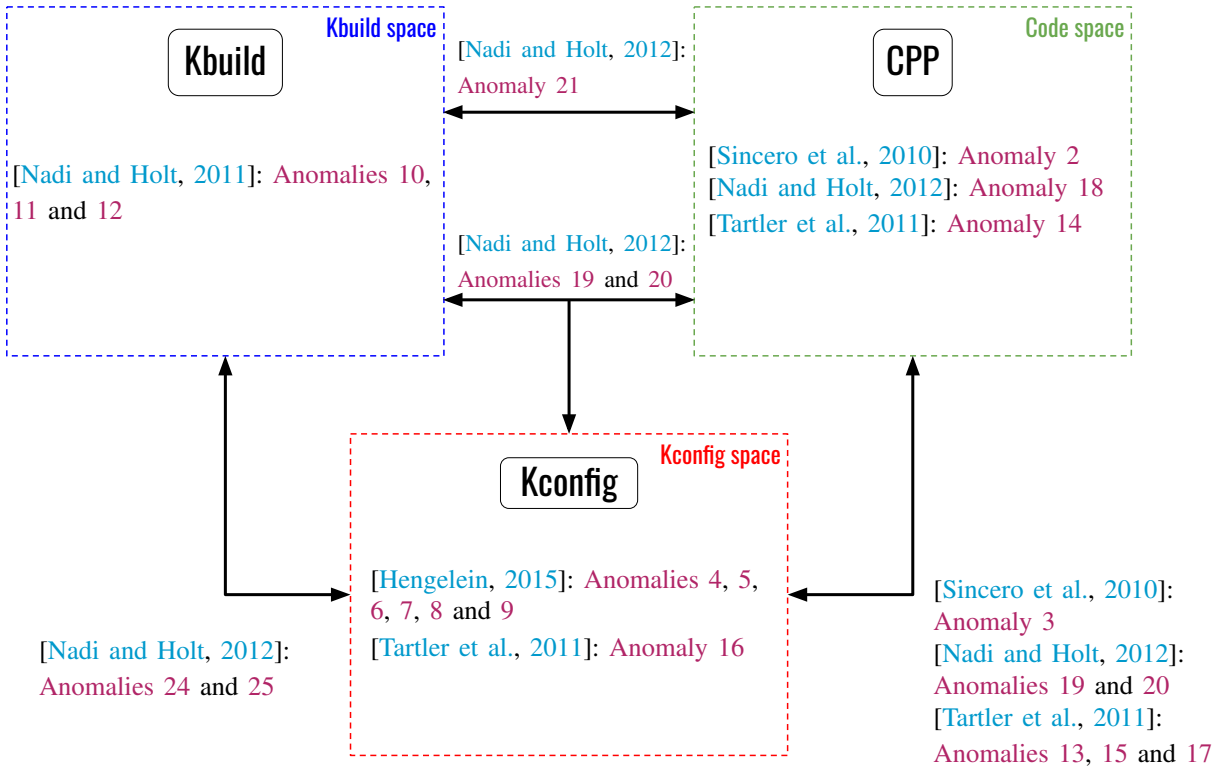


Figure 3.4: Synthetic map of inconsistencies analyses of the Linux Kernel

in Listing 3.1, Block 2 is *undead* according to Anomaly 13, as the selection of its parent (Block 1) implies its selection. However, if the `A` variable is not defined, then Block 2 is not *undead* according to Anomaly 15 as it is not always included.

Another limitation issues from the fact that the formalisms given for anomalies are at coarse grain. Sincero et al. [2010] presented a first implementation of UNDERTAKER for CPP implementing their formalism on CPP. Tartler et al. [2011] then improved it by adding a second level to UNDERTAKER for reasoning over the KCONFIG constraints, but the authors considered the constraints from CPP in the tool as a black box, altering the previously defined formalism. Consequently, the fine-grained comprehension of the feature–block link is then lost. The same issue can be found in the work from Nadi and Holt [2012] on the KBUILD space and the third extension they provide for UNDERTAKER. Moreover, spaces can also be named differently, sometimes with the same letter representing two different spaces in two definitions (C represents the CPP constraints in [Sincero et al., 2010; Nadi and Holt, 2012] and the KCONFIG constraints in [Tartler et al., 2011]). A summary of these differences is given in Table 3.3.

These issues and the lack of a uniform vision over the different analyses on the kernel variability hamper the understanding of both the issues and the proposed solutions, as well as their transfer in the future evolution of the build system. Furthermore, a uniform and consistent model could be applied to other highly configurable systems, such as the MOZBUILD. These limitations cause Challenge B2 (*"Characterizing and identifying anomalies in build systems"*) to be open.

Table 3.3: Notation mapping for constraints in the three spaces in three papers

Paper	CPP	KBUILD	KCONFIG
Sincero et al. [2010]	C	/	K
Tartler et al. [2011]	I	/	C
Nadi and Holt [2012]	C	M	K

3.6 Summary

Current state-of-the-art approaches do not allow deep comprehension of two aspects of variability implementations in large software systems. First, in the implementation of OO software systems, and second, in their management in ad hoc build systems. In this thesis, we tackle separately challenges related to the identification and comprehension of OO variability implementations and their quality (Part I) from the challenges related to the comprehension of the mechanisms resolving variability in build systems and the induced anomalies (Part II).

PART I

Comprehending the implemented variability in object-oriented code assets

CHAPTER 4

Assessing the *symfinder* method

This chapter shares material with the VaMoS 2020 paper *"Mapping Features to Automatically Identified Object-Oriented Variability Implementations – The case of ArgoUML-SPL"* [Mortara et al., 2020b] and the SPLC 2020 paper *"Identifying and Mapping Implemented Variabilities in Java and C++ Systems using symfinder"* [Mortara et al., 2020a]. Some elements are also extracted from the AUSE journal paper *"Identification and visualization of variability implementations in object-oriented variability-rich systems: a symmetry-based approach"* [Tërnava et al., 2022] which extends the work presented in [Tërnava et al., 2019] and [Mortara et al., 2020b].

The approach proposed by Tërnava et al. [2019] to identify variability implemented in OO software systems has been applied to multiple open source variability-rich systems written in Java. While the proposed graph visualization allowed on these systems to visually distinguish zones concentrating variability implementations, assessing the relevance of the approach requires additional steps. First, the *symfinder* toolchain [Mortara et al., 2019] automating the identification only supports Java codebases, therefore the use of symmetry in OO structures as a technique to identify OO variability implementation has been validated on this language only. Then, as the relevance of the identified *vp*-s and variants has been evaluated visually by the authors, evaluating whether they actually represent domain features requires a comparison between identified *vp*-s and variants with known domain features and their traces in the code assets. Finally, evaluating whether these identified potential variability implementations and their representation actually helps the comprehension of the implemented variability requires an evaluation by a third-party user.

In this chapter, we evaluate those two aspects to determine to what extent *symfinder* approach tackles Challenges A1 and A2 (*"Identifying variability implemented in OO software systems"* and *"Making the identified variability implementations comprehensible"* respectively). We assess the relevance of the *symfinder* approach by addressing the following questions:

1. **Is the identification approach adapted to other OO languages?** To address this question, we apply the notion of symmetry in OO constructs to C++ structures and extend *symfinder* to support C++ codebases. We then evaluate the approach on multiple C++ open source systems (Section 4.1).

2. **Do the identified *vp*-s and variants correspond to domain features?** To address this question, we apply *symfinder* on two systems for which domain features and their traces in

the code assets are available and measure the accuracy of the symmetry-based identification by adapting precision and recall metrics (Section 4.2).

3. **To what extent does the *symfinder* approach and its visualization help comprehension of the implemented variability?** To address this question, we report on an evaluation by Daniel Le Berre on his use of *symfinder* on the Sat4j project, of which he is the main architect (Section 4.3).

4.1 Reproducibility of the symmetry-based identification technique on C++ code assets

In order to assess the reproducibility of the *symfinder* approach, we apply on C++ systems the same protocol introduced by Tërnava et al. [2019] to evaluate the approach on Java systems. We first select a set of open source software systems implemented in C++ of various sizes and that can implement variability (Section 4.1.1). We then apply an adapted version of the *symfinder* toolchain to parse and identify symmetries in C++ codebases (Section 4.1.2) and report on the visually visible zones concentrating variability implementations (Section 4.1.3).

4.1.1 Subject systems

Table 4.1 synthesizes the five selected subject systems. **Zelda-Game** is a text based game made as a final project for an academic course. We selected it as its documentation states that it uses *"concepts of OOP like Inheritance, Composition, Association, Polymorphism"*, that is, mechanisms used to implement OO variability. **Decaf-Compiler** is a compiler for the Decaf language. As it is a small project designed as an academic project for a Compilers course, we can expect from it a clear code structure exhibiting the different elements composing the Decaf language. **PCSX2** is a PlayStation 2 emulator, thus we await variability concerning the different configuration options it can offer such as CPU tuning or video settings. **MuseScore** is a music notation and composition software. As it provides an editor for musical score, we can presume variability related to elements composing a sheet, such as the key, the tempo, or textual annotations. Finally, we applied *symfinder* on the **Mozilla Firefox** codebase as it is a large and widely use project, giving us confidence in the rigor of its implementation and allowing us to measure the scalability of our approach.

Table 4.1: The five studied variability-rich subject systems and their numbers of *vp*-s and variants identified by *symfinder*.

Subject system	Version	C++ LoCs	# *vp*-s	# *variants*
Zelda-Game	54a8197	1,132	1	2
Decaf-Compiler	820448a	1,549	5	32
PCSX2	v1.6.0	798,479	130	234
MuseScore	v3.4.2	482,152	388	930
Mozilla Firefox	53.0.3	5,550,504	5,928	11,681

Execution system: Ubuntu 18.04.2 LTS with Intel Xeon CPU E5-2637v2 @ 3.5GHz and 128Go memory.

```
1  // Template declaration
2  template<typename T>
3  class Value {};      // <--- VP
4
5  // Instantiation
6  class Long : public Value<long> {};      // <--- variant
7  class Int : public Value<int> {};      // <--- variant
8
9  // Specialization
10 template<>
11 class Value<long> {};      // <--- variant
12
13 template<>
14 class Value<int> {};      // <--- variant
```

Listing 4.1: Example of use of template class in C++

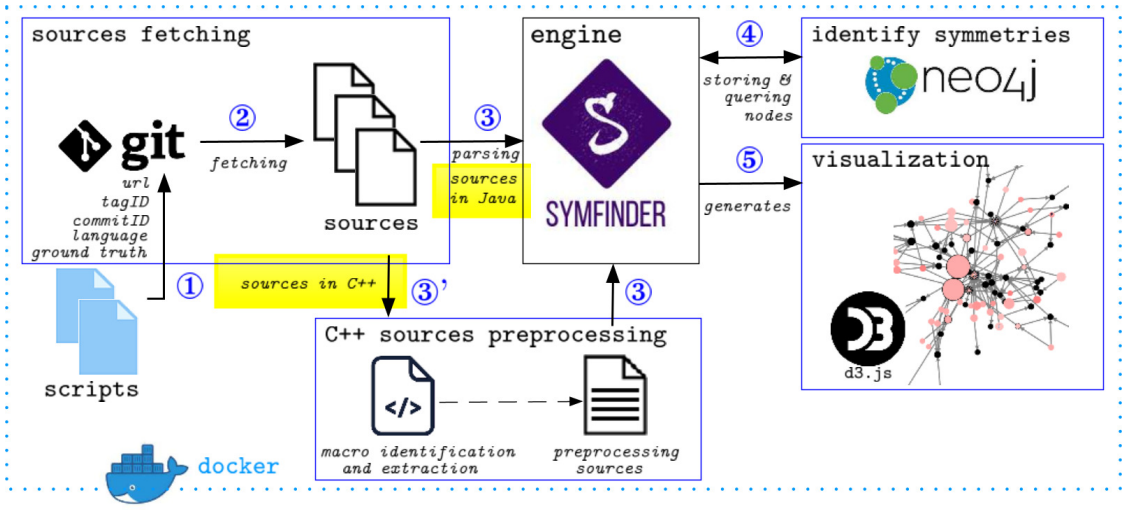
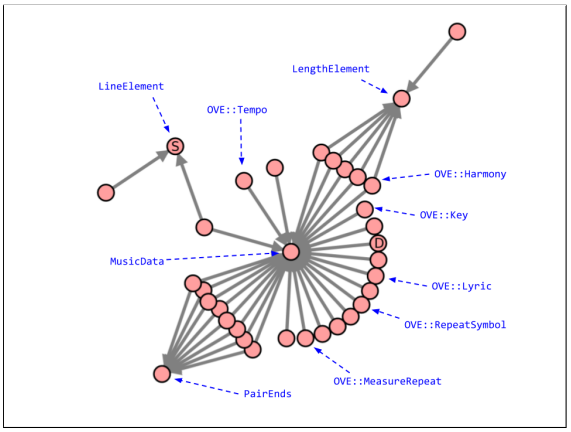


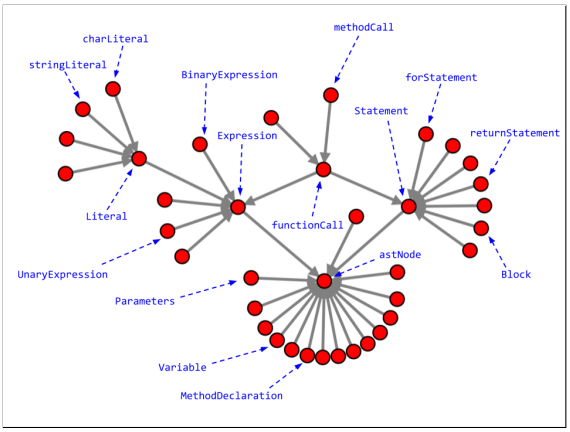
Figure 4.1: The extended *symfinder* toolchain supporting both Java and C++ systems

4.1.2 Adapting *symfinder* to C++ codebases

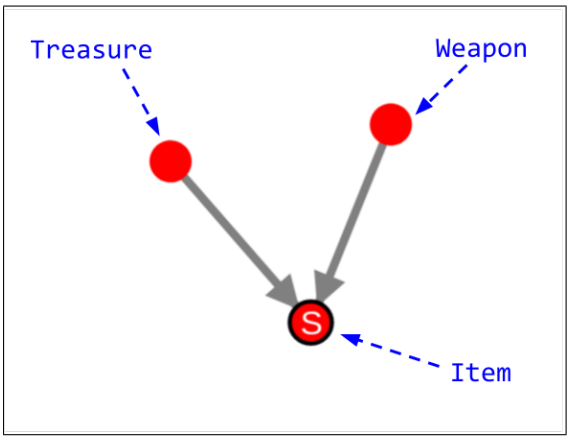
Identification approach As the OO mechanisms implementing variability are similar between Java and C++ systems, the identification of mechanisms common to both languages (listed in Table 3.1) has not been altered apart from few adaptations such as between the interfaces in Java and virtual pure methods in C++. Regarding mechanisms specific to C++, the identification has been extended to the class template mechanism, illustrated in Listing 4.1. Once the template has been declared, it can be *instantiated* in a concrete class or *specialized* in another template with a concrete type. We observe symmetries in both these mechanisms. By considering the template declaration as a center of symmetry, the classes instantiating the template and the specializations of the template are the changeable parts of the symmetry. Therefore, we consider the mechanisms as variability implementations, considering as a *vp* the template declaration and as variants both their instantiations and specializations.



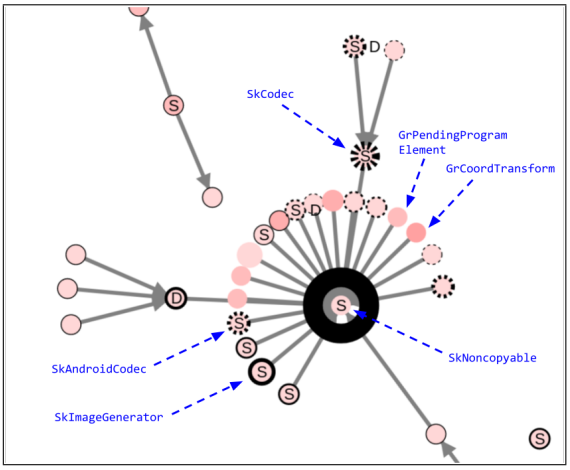
(a) MuseScore visualization excerpt exhibiting the different music sheet elements.



(b) Decaf-Compiler visualization excerpt exhibiting the different concepts of the Decaf language.



(c) Zelda-Game visualization.



(d) Firefox visualization excerpt.

Figure 4.2: Excerpts of visualizations generated by *symfinder*

Implementation Figure 4.1 illustrates the *symfinder* toolchain adapted to support both Java and C++ codebases. We first developed a second parser using the ANTLR parser generator and a grammar supporting C++14. Software written in C++ is known to make use of CPP macros, which are likely to implement variability as well [Liebig et al., 2010; Hunsen et al., 2016]. As a first step, we decided to handle these macros to be able to identify symmetries in all C++ code assets, but without representing the variants potentially created by the preprocessor directives. While this is obviously an interesting feature, we decided to focus first on the symmetry-based approach. Specifically, After fetching the sources of a C++ project, an additional preprocessing step (③') extracts macro definitions in a file and used to precompile the system using the C preprocessor and to expand macros. Then, similar to the Java version, a second analysis in ③ generates an AST, which is used to build a graph representation of the source code, stored in a graph database. It is then queried to identify symmetries in OO variability implementation mechanisms.

4.1.3 Results

Table 4.1 presents the numbers of *vp*-s and variants identified by *symfinder* in the 5 systems. We can notice that on these projects, *symfinder* tends to find more *vp*-s and variants on larger projects, although there is no direct correlation.

We can observe in the generated visualizations dense zones corresponding to expected variability implementations. For example, Figure 4.2a exhibits a *vp*, `MusicData`, surrounded by multiple class level variants linked to it with inheritance relationships (*e.g.*, `OVE::Tempo`, `OVE::Key`, `OVE::Harmony`) corresponding to the expected variability regarding the music sheet elements. While very few *vp*-s and variants have been identified in Decaf-Compiler (Figure 4.2b), they appear as relevant by exhibiting the awaited elements composing the Decaf language such as `Expression` or `functionCall` as well as the different types of `Literals` (`charLiteral`, `stringLiteral`, ...). Analogously to the identification in Java systems, a lowly-variable system will lead to a light visualization, as it is the case for Zelda-Game whose complete visualization is shown in Figure 4.2c and that makes use of inheritance solely to model two types of `Item`. Finally, due to the nature of the OO variability implementations, some dense zones on the visualization might correspond to the use of the mechanisms for implementation purpose, as in the excerpt of the Firefox visualization shown on Figure 4.2d. The obtained visualizations are accessible online[1].

4.1.4 Threats to validity

As our experimentation protocol is analog to the one used by Tërnava et al. [2019], the threats to the validity of this experiment are similar and mainly concern the choice of our subject systems. Although restricted to five projects, the fact that our dataset is made of multiple projects of multiple dimensions (including the popular Mozilla Firefox web browser) gives us confidence in the applicability of our approach.

Although the identified variability implementations observed on some systems correspond to expected variability implementations (*e.g.*, MuseScore), the remaining ones correspond to the use of such mechanisms to organize the code (*e.g.*, in Mozilla Firefox). As for systems written in C, C++ systems may also rely on `ifdef` directives to implement their variability [Liebig et al., 2010; Hunsen et al., 2016]. Therefore, when our approach does not find relevant variability, it can be that it is implemented using these directives.

4.1.5 Summary

The adaptation and application of *symfinder* on C++ systems results in observations similar to the ones obtained by Tërnava et al. [2019] on Java systems, thus confirming the relevance of the identification using symmetries. While it appears that some zones appearing as dense on the visualization actually represent variability implementations, others correspond to the use of such mechanisms for implementation purpose. We now aim to evaluate to what extent identified *vp*-s and variants represent actual variability implementations.

[1] https://deathstar3.github.io/symfinder-demo/cpp-projects.html

Table 4.2: The two subject systems with their respective LoC, total number of potential *vp*-s with variants, and class or method level granularity identified by *symfinder*

Subject system	LoC	Class (C) and Method (M) level				Total	
			#vp-s	#variants	#nodes	#vp-s	#variants
ArgoUML	134,367	C	327	860	85	774	1,976
		M	447	1,116	–		
Sat4j	27,638	C	80	135	10	268	588
		M	188	453	–		

4.2 Automatic mapping of variability implementations

In order to evaluate the relevance of identified *vp*-s and variants, we apply *symfinder* on two Java systems for which domain knowledge is available and compare the results of our identification to the existing features traces in the code assets.

4.2.1 Subject systems

In the following are introduced the ground truths from the ArgoUML-SPL case study and Sat4j that we use to conduct the feature mapping experiment.

4.2.1.1 ArgoUML-SPL

The first project we selected is ArgoUML[2], an open source UML modeling tool implemented in Java language. It is used in the software product line community as a realistic case study for demonstrating the basic challenges for refactoring a single code base system with variability into an SPL [Couto et al., 2011]. The extracted ArgoUML-SPL, with its ground truth, was also recently proposed [Martinez et al., 2018] and used [Cruz et al., 2019; Müller and Eisenecker, 2019; Michelon et al., 2019, 2021a,b] as a benchmark for evaluating the feature location techniques[3]. The considered ArgoUML-SPL ground truth [Martinez et al., 2018] consists of a feature model (FM) and annotations in the code asset providing *traces* with their implementations.

11 features compose ArgoUML-SPL's feature model, given in Figure 4.4. The abstract feature `ArgoUML-SPL` represents conceptually the SPL domain, which has 2 mandatory features (`Diagrams` and `Class`) and 8 optional features (`State`, `Activity`, `Use Case`, `Collaboration`, `Deployment`, `Sequence`, `Cognitive Support`, and `Logging`). In ArgoUML's ground truth [Martinez et al., 2018], each of these 8 optional features has a set of traces to the ArgoUML's code assets, totalizing 714 features traces (without duplication).

Executing *symfinder* on ArgoUML leads to the identification of 2,750 potential *vp*-s with variants at class and method levels. A subset of potential *vp*-s with variants at class level is given in Figure 4.4[4].

[2] https://www.openhub.net/p/argouml

[3] https://variability-challenges.github.io/2018/ArgoUMLSPL/index.html

[4] The whole ArgoUML's visualization is available at https://deathstar3.github.io/symfinder-demo/JRN20/hotspots_version/argoUML-bcae37.html.

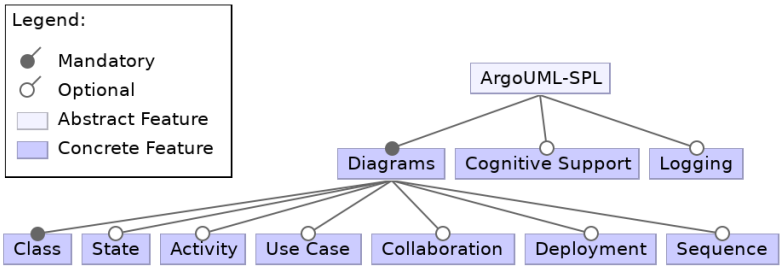


Figure 4.3: Feature model of ArgoUML, adapted from [Couto et al., 2011]

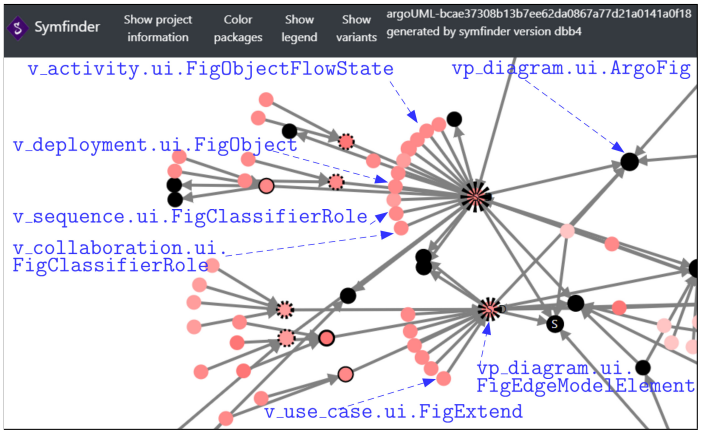


Figure 4.4: An excerpt of the visualization in ArgoUML generated by *symfinder* (from [Tërnava et al., 2022])

4.2.1.2 Sat4j

The second project selected for this experiment is Sat4j. As opposed to ArgoUML-SPL, no ground truth was previously available for this system. We therefore asked its architect, Daniel Le Berre[5], to prepare a ground truth for the purpose of this study. To avoid any possible bias and data manipulation, we first held a meeting with him where we discussed the purpose of the study and the needed data. Then, only him prepared the ground truth by manually defining the domain features, organizing them in a feature model (given in Figure 4.5) and annotating the code assets (classes, interfaces, methods, or fields) in the `org.sat4j.core` module that belong to each domain feature[6] using Java annotations. Extraction for the mapping is then done using an internal utility tool that outputs the list of traces into a Markdown file[7].

13 features compose Sat4j's feature model. The abstract feature `Sat4j-SPL` represents conceptually the Sat4j's variability domain, which has 7 mandatory features (`Reader`, `Solver`, `Constraint`, `Deletion`, `Learning`, `Var Heuristics`, and `Phase Heuristics`) and 5 optional features (`Solution Listener`, `Unit Clause Provider`, `Search Listener`, `Simplifications`, and `Restarts`). All 12 concrete features have a set of traces to its code assets, totalizing 118 features traces.

[5] http://www.cril.univ-artois.fr/~leberre/
[6] Sat4j's code: https://gitlab.ow2.org/sat4j/sat4j/-/tree/master/org.sat4j.core
[7] Sat4j's ground truth: https://deathstar3.github.io/symfinder-demo/JRN20-files/Features.pdf.

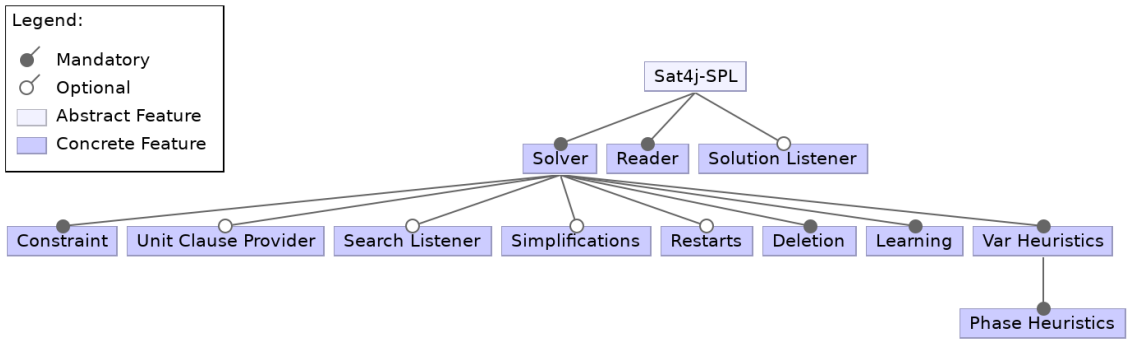


Figure 4.5: Feature model of Sat4j (from [Tërnava et al., 2022])

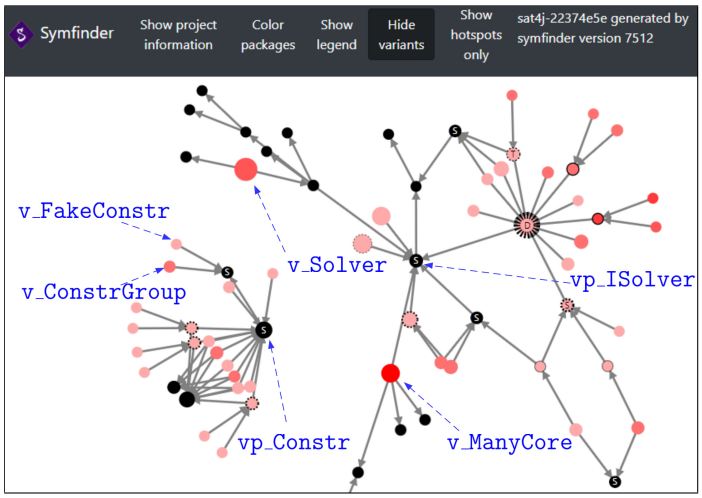


Figure 4.6: An excerpt of the visualization in Sat4j generated by *symfinder* (from [Tërnava et al., 2022])

Executing *symfinder* on Sat4j leads to the identification of 856 potential *vp*-s with variants at class and method levels. A subset of potential *vp*-s with variants at class level is given in Figure 4.6[8].

4.2.2 Mapping process

Before the actual mapping, we normalized the granularity of traces for the domain features with the granularity of the identified *vp*-s with variants, so they all become of a common class level granularity. Specifically, whenever a feature in the ground truth had one of its traces to a class refinement, complete method, or method refinement, we simplified that trace to the whole class. For instance, the feature `State` in the ground truth had one of the trace links to `org.argouml.ui.cmd.GenericArgoMenuBar`

[8]The whole Sat4j's visualization is available at https://deathstar3.github.io/symfinder-demo/JRN20/hotspots_version/sat4j-22374e5e.html.

Table 4.3: The mapping of an identified *vp* with its eight variants at class level to features, visualized also in Figure 4.4.

A *vp* with variants	**Feature**
vp: **`ui.FigNodeModelElement`**	Cognitive, Logging
`use_case.ui.FigActor`	Use Case
`sequence.ui.FigClassifierRole`	Sequence
`static_structure.ui.FigComment`	Logging
`collaboration.ui.FigClassifierRole`	Collaboration
`activity.ui.FigObjectFlowState`	Activity
`ui.FigEdgePort`	-
`deployment.ui.FigObject`	Deployment
`activity.ui.FigPartition`	Activity

`initMenuCreate() Refinement`[9], which is a trace at the statement level within the method `initMenuCreate()`. In such a case, we truncated the trace to the whole class `org.argouml.ui.cmd.GenericArgoMenuBar`. This means that we consider all features' traces, but we only change their granularity to class level. Then, from the identified *vp*-s with variants, we considered only those at the class level but these include also all method level *vp*-s with variants. This normalization is necessary for two reasons. First, *symfinder* records the names only for class level *vp*-s and variants, whereas for those at method level it records only their total number (*cf.* Section 3.1.2). The second reason is that *vp*-s with variants are related only to the structural elements in code, such as classes or methods for now, whereas features in the ground truth have traces mostly to their refinements, where about 73% of them are at statement level.

Our goal is to evaluate for each class having at least a feature trace whether it has been identified as a variability implementation and vice versa. As after this normalization, both the identified potential variability implementations and the features traces are aligned at the class level, we can automatically map the classes identified as variability implementations to the classes having features traces[10]. For instance, we would expect that given their name, potential *vp*-s and variants in Figures 4.4 and 4.6 all have a mapping to domain features in Figures 4.3 and 4.5, respectively. An example of such expected mapping is given in Table 4.3, illustrating the mapping of the `ui.FigNodeModelElement` *vp* and eight from its eighteen class variants (visible on Figure 4.4) with domain features (Figure 4.3). The *vp* itself maps to two features, `Cognitive` and `Logging`, and the eight shown variants map to six features. Seven of the shown variants are mapped to one feature, two of which map to the same `Activity` feature, whereas one variant, `ui.FigEdgePort`, is without a mapping, meaning that this variant does not appear as a trace link in any of the features in the ArgoUML-SPL's ground truth. The complete raw, normalized, and analysed data are available online[11].

[9] https://github.com/but4reuse/argouml-spl-benchmark/blob/master/ArgoUMLSPLBenchmark/groundTruth/STATEDIAGRAM.txt

[10] While the term 'tracing' / 'trace links' is used in the ground truth, we will distinguish from this term in this experiment by using 'mapping' / 'mapping links' for *vp*-s and variants mapped to features, although both of them have the same meaning.

[11] https://deathstar3.github.io/symfinder-demo/mapping_process.html

Table 4.4: Summarized data from the two ground truths and their results [Tërnava et al., 2022]

	ArgoUML	**Sat4j**
#domain features	11	13
#features traces (normalized)	672	113
#local symmetries (class level)	1 272	225
True Positives (TP)	561	113
False Positives (FP)	711	112
False Negatives (FN)	111	0
Precision	44.10%	50.22%
Recall	83.48%	100.00%

4.2.3 Mapping measures

Measuring the results of the mapping is done using two well-known measures, namely *precision* and *recall*.

Precision Let be T_{gt} the set of traces for all features given in the ground truth and I_{vp-v} the set of all identified *vp*-s and variants by *symfinder*. We use *precision* to measure the percentage of the identified *vp*-s and variants that are relevant for the feature mapping. Thus, for the current mapping, the *vp*-s and variants that are mapped to the features in the ground truth are true positives (TP), referred as relevant *vp*-s and variants, whereas the *vp*-s and variants without a mapping are false positives (FP), or irrelevant *vp*-s and variants. Therefore,

$$precision = \frac{TP}{TP + FP} = \frac{|T_{gt} \cap I_{vp-v}|}{|I_{vp-v}|}$$

Recall Through *recall* we measure the percentage of features' traces in the ground truth that are used for the mapping of *vp*-s and variants to features. Thus, the traces that are used for the mapping are true positives (TP), whereas those that are not used are false negatives (FN). Therefore,

$$recall = \frac{TP}{TP + FN} = \frac{|T_{gt} \cap I_{vp-v}|}{|T_{gt}|}$$

4.2.4 Results and discussion

Table 4.4 summarizes the obtained results. In ArgoUML, according to our mapping tool, 561/1,272 identified potential *vp*-s and variants have a mapping to at least one feature, while the remaining 711 appear as unmapped, leading to a precision of 44.10%. Then, out of 672 traces in the code assets, 111 are not mapped to any identified *vp* nor variant leading to a recall of 83.48%. Besides, in Sat4j, 113/225 identified potential *vp*-s and variants have a mapping to at least one feature, while the remaining 112 appear as unmapped, leading to a precision of 50.22%. Then, all traces in the code assets are mapped to identified *vp*-s and variants leading to a recall of 100%.

Although it is expected from such a non-trivial mapping that several local symmetries are without a mapping to features in the ground truth of both systems (and conversely), a considerable number of false positive local symmetries may impact the time spent distinguishing the actual *vp*-s with variants from the irrelevant local symmetries in the visualization. We observed three main reasons for such low precision.

Reason 1. The ArgoUML-SPL ground truth has been extracted by a group of researchers based on their domain knowledge of ArgoUML, and some information regarding the completeness of the features' list is missing. The ground truth may therefore be incomplete and would explain the important number of identified potential *vp*-s and variants without mapping. The slightly greater precision obtained for Sat4j is explained by the fact that we asked its architect to provide a list of features with traces as exhaustive as possible.

Reason 2. Since variable features correspond to configuration units when deriving a product in an SPL, feature identification and location approaches mainly focus on mapping *variable* features to code assets, often disregarding *mandatory* features [Krüger et al., 2018]. Similarly, as ArgoUML-SPL's ground truth has been designed as a benchmark to evaluate extractive SPL approaches, some mandatory features such as `Class` (*cf.* Figure 4.3) have no trace in code assets. However, in our symmetry-based approach, some potential *vp*-s and variants could correspond to mandatory features for which no trace exists (especially *vp*-s as they represent commonalities), explaining further the low precision obtained for ArgoUML-SPL. Regarding Sat4j, although 7/13 features are mandatory (*cf.* Figure 4.5), all of them have traces in code assets, totalizing 69% of the total number of traces, explaining the slightly higher precision.

Reason 3. We can presume that not all places where symmetries have been identified are related to variability implementation, as it is the case for preprocessor directives in C/C++ systems. For example, Zhang et al. [2013] mentioned in an industrial case study that *"from our experience most #ifdef blocks (*e.g.*, 87.6% in the Danfoss SPL) are actually not variability related, but for other purposes such as include guards or macro substitution"*. While, in the case of object-oriented systems, the used mechanisms are different, they are due to their nature likely to be mainly used as a good practice to structure domain objects (*cf.* Challenge A1). Additionally, after we reported the results to the architect, he pointed out that in order to avoid redundant annotations, he annotated only the concrete classes and not the abstract classes they inherit from. Then, only the main variability sources have been annotated. For example, the code contains multiple interfaces having at most two implementations that are not annotated. Further analysis of the false positives show that they are mainly still variability related, but only at the level of internal implementation, and not at the domain level (*cf.* Section 4.3.2.3). To avoid any data manipulation, we decided to present the original genuine experiment with the annotations as decided by the architect.

After further manual investigation of the 17% of traces that could not be mapped to *vp*-s nor variants, it results that they refer to the statements within the initialization classes, such as `Main` classes, or use other external libraries. As *symfinder* does not categorize initialization classes as part of local symmetries and filters out external libraries, they naturally could not be mapped.

4.2.5 Threats to validity

The main threat of this experiment concerns the two subject systems, ArgoUML and Sat4j, as well as their ground truths. As opposed to Sat4j's ground truth that has been established by its software architect and is now part of the codebase on its main branch, ArgoUML's ground truth has been established by a group of researchers. Therefore, another group of researchers or the ArgoUML's developers themselves may identify slightly different features and trace links, thus having a direct impact on the obtained results for precision and recall of our tooled approach on this system. Sat4j's ground truth was established by the software architect after he reported the experience with *symfinder* (Section 4.3). Although it can be seen as a maturation threat, some basic knowledge for variability in his own system was essential to avoid adding meaningless annotations as features traces.

Another threat concerns the normalization applied to the data prior to the mapping (*cf.* Section 4.2.2). During this step, the method level features traces and the identified local symmetries were normalized to class level. Considering method level feature traces and local symmetries may have an impact on the obtained precision and recall and consequently on the quality of the mapping

4.2.6 Related Work

To manage variability in SPLs, most of the existing approaches propose to modularize features into physically separate modules [Apel et al., 2013] or use conditional compilations, such as preprocessors in C/C++ [Liebig et al., 2010; Le et al., 2013; Tartler et al., 2012; Hunsen et al., 2016], or a form of annotations [Heymans et al., 2012; Couto et al., 2011]. In these cases, features have a straightforward mapping in code assets using their naming conventions. However, extensive manual effort is required to add annotations in code assets or refactor them into feature modules. With *symfinder* we provide an automatic approach for identifying variability places in OO code assets. The results of the manual feature mapping conducted in ArgoUML-SPL case study show that these automatically identified places are highly relevant and indeed implement domain features. Similarly to other approaches, this mapping is likely to be automated, although not completely, by simply using the features and *vp*-s naming.

Since Couto et al. [2011] extracted the ArgoUML-SPL, it has been proposed [Martinez et al., 2018] and extensively used [Martinez et al., 2017a; Michelon et al., 2019; Cruz et al., 2019] as a benchmark for reverse engineering variability and evaluating feature location techniques [Rubin and Chechik, 2013; Assunção et al., 2017]. In contrast, our variability identification and visualization approach is more a tool support for understanding implemented variability in forward engineering [Tërnava et al., 2019]. Thus, in addition to its usage in reverse engineering, we show that ArgoUML-SPL can also be used as an interesting study in another context.

Besides a recent mapping study shows that there are several approaches for information visualization in SPL engineering [Lopez-Herrejon et al., 2018]. Only a few of them visualize the variability at code level. From them, the approach for a virtual separation of concerns [Kästner et al., 2008; Kästner, 2010; Feigenspan et al., 2011b] is mostly related to our visualization approach. Similarly, it is used for variability management and relies on a different color per feature to manually map them to code assets. In contrast, we use a graphical visualization of variability by using more visualization parameters, namely position, size, shape, value (lightness), color hue, orientation, and texture. We also automatically visualize the variability and keep it separate from code assets.

4.2.7 Summary

On both studied subject systems, the symmetry-based identification method showed little precision but high recall, meaning that although only around half of them correspond to features traces (44% and 50% for ArgoUML and Sat4j respectively), they implement a high proportion of given domain features (83% and 100% for ArgoUML and Sat4j respectively). Little precision can be explained by the coarse-grain nature of the known features and by the fact that some of the *vp*-s and variants are not variability related. Still, the high recall shows their relevance and demonstrates the feasibility of the *symfinder* approach.

While it appears that the identification method needs to be improved to be more precise, understanding deeply the causes of these results requires information from an architect, who has the knowledge of both the domain and the implementation. We thus deepen our analysis of the results obtained on Sat4j by asking its main architect, Daniel Le Berre, to evaluate *symfinder* both regarding its identification method and the visualization it provides.

4.3 User evaluation of *symfinder*

4.3.1 Experimental setup

For this experiment we selected Sat4j [Le Berre and Parrain, 2010], a variability-rich system and asked its software architect, Daniel Le Berre[12], for his feedback on using the tool. We chose Sat4j not only as it is a Java open source system in a single codebase, but also because it is a research software implemented as an example in teaching software engineering. It therefore heavily relies on object-oriented programming guidelines such as the use of inheritance and design pattern to implement its variability. Its design evolved for about 15 years with regular feature improvements and addition. Out of the four modules that compose Sat4j (`core`, `pb`, `sat`, and `maxsat`), we applied *symfinder* only on `core` as it contains the main features of the system, and made the generated visualization available to the software architect.

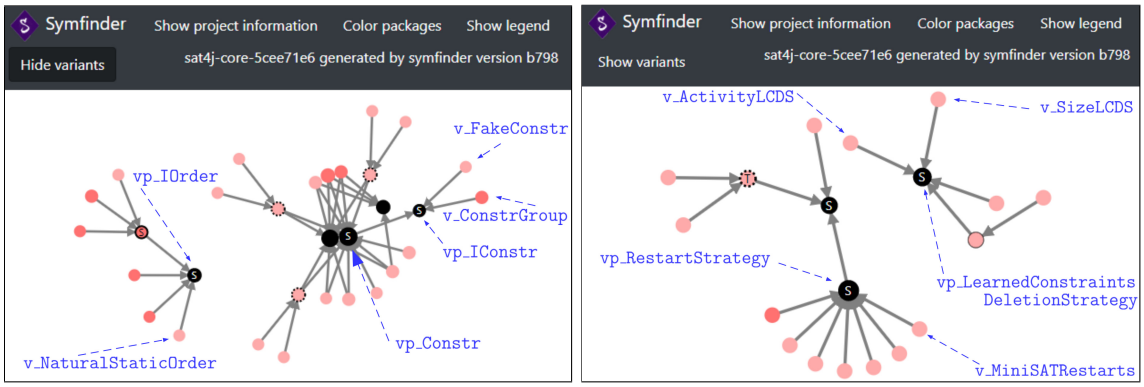
4.3.2 Observations

To further understand the obtained mapping results, we asked the software architect to detail the actual variability implementations identified by *symfinder* (*i.e.*, *true positives*, detailed in Section 4.3.2.1), the variability implementations missed by *symfinder* (*i.e.*, *false negatives*, detailed in Section 4.3.2.2), and finally identified *vp*-s and variants that do not correspond to domain variability (*i.e.*, *false positives*, detailed in Section 4.3.2.3). Then, to evaluate how *symfinder* can help the comprehension of the implemented variability, the architect provided feedback on the general interest of *symfinder* for a software architect (Section 4.3.2.4) and on his particular interest for Sat4j (Section 4.3.2.5), and formulated requests for enhancements of the tooled approach (Section 4.3.2.6).

4.3.2.1 Variability correctly identified by *symfinder*

The feature model given in Figure 4.5 details the different features provided by Sat4j. Their implementation heavily relies on design patterns. Fully customizable Boolean solvers are proposed,

[12] http://www.cril.univ-artois.fr/~leberre/



(a) Identified *vp*-s with variants for features `Heuristics` with `Constraints` in Sat4j

(b) Identified *vp*-s with variants for features `Restarts` with `Constraints database management` in Sat4j

Figure 4.7: The identified *vp*-s with variants for four of the features in Sat4j

whose configuration is implemented using the Strategy design pattern. Decorator patterns allow configuring an optimization problem instead of a decision problem. Finally, multiple Factories expose prebuilt solver configurations. The implementation also considers two levels of abstraction: the `user` level for Java developers that are not familiar with the design of SAT solvers and aim to use the library to solve regular problems, and the `expert` level for advanced users having a deeper understanding of the algorithms (master students, researchers, ...) and that need to customize the solver at fine-grain.

It results that *symfinder* identified most of the domain variability implemented using the strategy design pattern. Figure 4.7a exhibits the `IOrder` and `IConstr` interfaces implementing the *heuristics* and *constraints* features respectively. Figure 4.7b exhibits the `RestartStrategy` and `LearnedConstraintsDeletionStrategy` interfaces implementing the *restarts* and *constraints database management* features respectively. Additionally, some abstractions corresponding to the `user` and `expert` levels can be seen with *symfinder* as the inheritance between two interfaces, see *e.g.*, `IConstr` and `Constr` in Figure 4.7a.

4.3.2.2 Variability missed by *symfinder*

Only two variability implementations could not be retrieved by *symfinder*:

- The `ISimplifier` interface and its implementations, providing various clauses simplifications techniques, were not detected due to their implementation relying on anonymous inner classes inside the solver class. Although unconventional, this design choice is motivated by the fact that a direct access to the state of the solver is required.
- The `PrimeImplicantStrategy` interface, allowing to reduce the model found by the solver to a set of literals required to satisfy all the constraints. This feature being experimental, it does not appear on the feature model. Moreover, the interface is present in a method (and not as a field), while the variant choice is made based on the value of a system property.

It results that in both cases, the non-identification results from an unconvential implementation motivated by efficiency or limited scope reasons.

4.3.2.3 On the remaining identified variability

Some identified *vp*-s not being linked to domain variability actually represent implementation variability. As an example, `IVec` and `IVecInt` are two data structures automatically identified as *vp*-s by being interfaces (*cf.* Figure 4.8a). The heavy use of interfaces to structure Sat4j's implementation explains the important number of identified false positives (*cf.* Table 4.4). Still, many false positive *vp*-s found in Sat4j correspond to variability implementations, but not related to the domain variability envisioned by the architect.

4.3.2.4 General interest of *symfinder* for the software architect

As a first reported feedback, the visualization allowed the architect to quickly spot the main *vp*-s in the code. An unexpected organization of classes on the visualization allows to visualize the evolution of the design and potentially spot unforeseen consequences. The provided global view is the most important feature of *symfinder* as it allows to detect unexpected relationships. Checking the nodes' details then allows to determine whether it is a design error or not. As for each tool, a little adaptation time is needed to get used to the displayed information and to learn the common patterns in the graphs.

4.3.2.5 Concrete interest of *symfinder* for Sat4j

In Sat4j's case, checking the variability was as easy as to look for interfaces marked as strategy and checking their name. Then, a similar observation was made for abstract classes and then the remaining plain interfaces (appearing as black nodes on the visualization) with several implementations, to make sure none is missed.

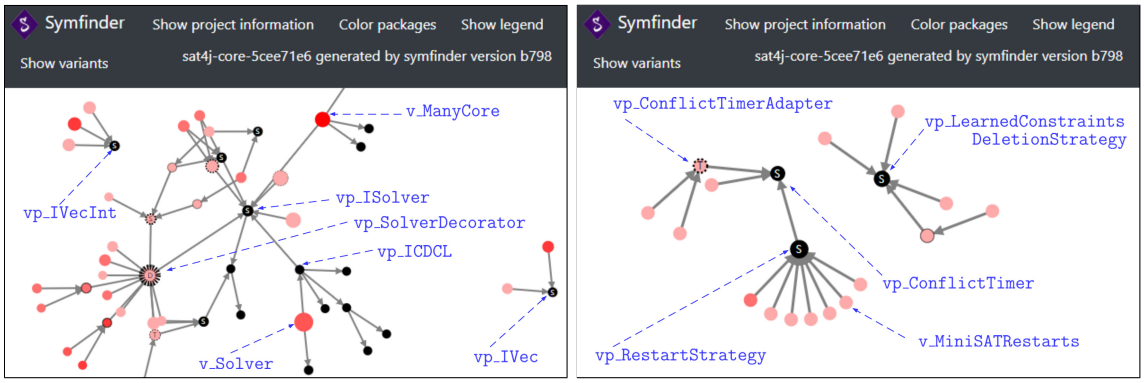
Sat4j's analysis allowed to identify improper variability management as illustrated on Figure 4.8b. `RestartStrategy` extending `ConflictTimer` does not correspond to the two levels of abstraction mentioned earlier and is clearly a bad design choice from a variability point of view. Despite being the *vp* for an identified strategy pattern, `RestartStrategy` is also a variant of `ConflictTimer`, *vp* for an identified strategy pattern but being in reality a composite pattern, that we do not identify. Having `RestartStrategy` inherit from `ConflictTimer` was therefore the simplest implementation choice from a developer perspective. On the other hand, `LearnedConstraintsDeletionStrategy` is also part of the composite pattern but using aggregation, that is not analyzed by *symfinder*. There is certainly some refactoring work to do to uniformize the design of all *vp*-s.

4.3.2.6 Requests for enhancement

Following his experience using *symfinder*, the architect formulated multiple requests to improve both the provided visualization and the identification approach:

1. In its current version, *symfinder* hides the variants that are not *vp*-s or do not have *vp*-s at method level, preventing a quick identification of a strategy according to its number of concrete classes. Since textual information is limited, seeing these variants is important for the architect.

2. Materializing Java packages might also be useful. In the case of Sat4j, strategy interfaces are most often in the same package as the solver, while their implementation is in a dedicated



(a) Identified *vp*-s with variants not related to domain features in Sat4j

(b) Identified unexpected *vp*-s with variants in Sat4j

Figure 4.8: The identified *vp*-s with variants not related to domain features and the unexpected ones

package. Being able to visually distinguish different packages would allow to quickly spot if all the variants of a strategy have been identified that way, or whether such practice is consistently used in the codebase.

3. Similarly, having different colors for each strategy interface with its related classes would enable to get a quick overview of the diversity of *vp*-s.

4. Finally, a textual enumeration of the identified design patterns would facilitate the validation of their relevance. While the graph view is useful to highlights important zones, it is not necessarily convenient for a fine-grain analysis.

Another proposal regards the identification approach. After further analysis, it resulted that an important part of false positive *vp*-s are interfaces with very small number of implementations, often one or two (*cf.* Section 4.3.2.3). Therefore, being able to customize the minimal number of variants to consider for detecting a strategy (to 3 for instance instead of 2 in the current implementation) or *vp*-s in interfaces (automatically considered *vp*-s regardless of their number of variants) would allow differentiating in this setting domain *vp*-s from implementation *vp*-s.

4.3.3 Threats to validity

The main threat to the validity of this experiment resides in the fact that Sat4j being the only analysed system in this qualitative experiment, generalization of the results to other systems is unfeasible. However, by being used in both the mapping (Section 4.2) and the qualitative analysis, the ability to cross-check the obtained results between both experiments gives more validity to the qualitative analysis.

4.3.4 Summary

It results from this experiment that *symfinder* could help Sat4j's software architect in understanding the variability implemented in his own system. The visualization provides a global picture of the implemented variability's organization and allows to spot inconsistencies in the design. Regarding

the identification method, the evaluation confirmed that a majority of the false positive *vp*-s with variants identified in the system correspond to isolated *vp*-s with a small number of variants.

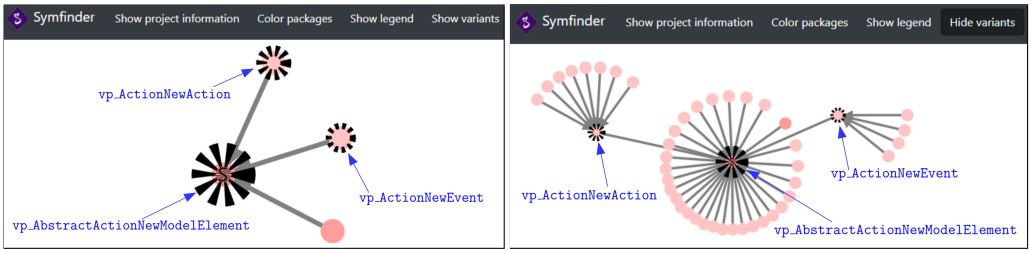
4.3.5 Addressed evolutions

As a first step towards a better visualization of the identified *vp*-s and variants, two of the previously proposed evolutions of the *symfinder* visualization in Section 4.3.2.6 have been implemented.

First, we added the option to visualize at once all class level variants, including those that are without method level *vp*-s. The importance of displaying these classes, already noticed during the measurement of *symfinder*'s precision and recall in Section 4.2 where we observed that a considerable number of feature traces were mapped to class level variants hidden in the visualization, has been confirmed by the architect when manipulating the visualization. Still, we left available the option to also hide them, as on large systems visualizing all variants considerably overloads the visualization. For example, ArgoUML's visualization exhibits 539 nodes when visualizing potential *vp*-s without variants and up to 1 233 nodes when all their variants are visualized. Taken from ArgoUML's visualization, Figure 4.9 illustrates the case when *vp*-s with class granularity are visualized without variants (Figure 4.9a) and with variants (Figure 4.9b). For this reason, we provided a toggle button on the visualization from where all class level variants can be visualized or not. By default, only potential *vp*-s with their variants that have method level *vp*-s are visualized.

Then, the visualization has been enhanced with an option allowing to color packages. A `Color packages` button on the top bar of the visualization opens a menu where the user can input a package name, namespace (in C++), or class name, whose potential *vp*-s with variants will be colored. The user can color multiple packages and/or classes, and for each one *symfinder* will automatically generate a new color. An example of visualization with colored packages is given in Figure 4.10, showing a good separation of concerns between packages in that case.

Finally, this experimentation also allowed us to see that two other design patterns were often used to implement variability, namely Decorator and Template. We therefore added the identification of *vp*-s with variants implemented by these patterns.



(a) without variants (b) with variants

Figure 4.9: Example of visualized *vp*-s.

4.4 Conclusion

In this chapter, we tackled the following questions:

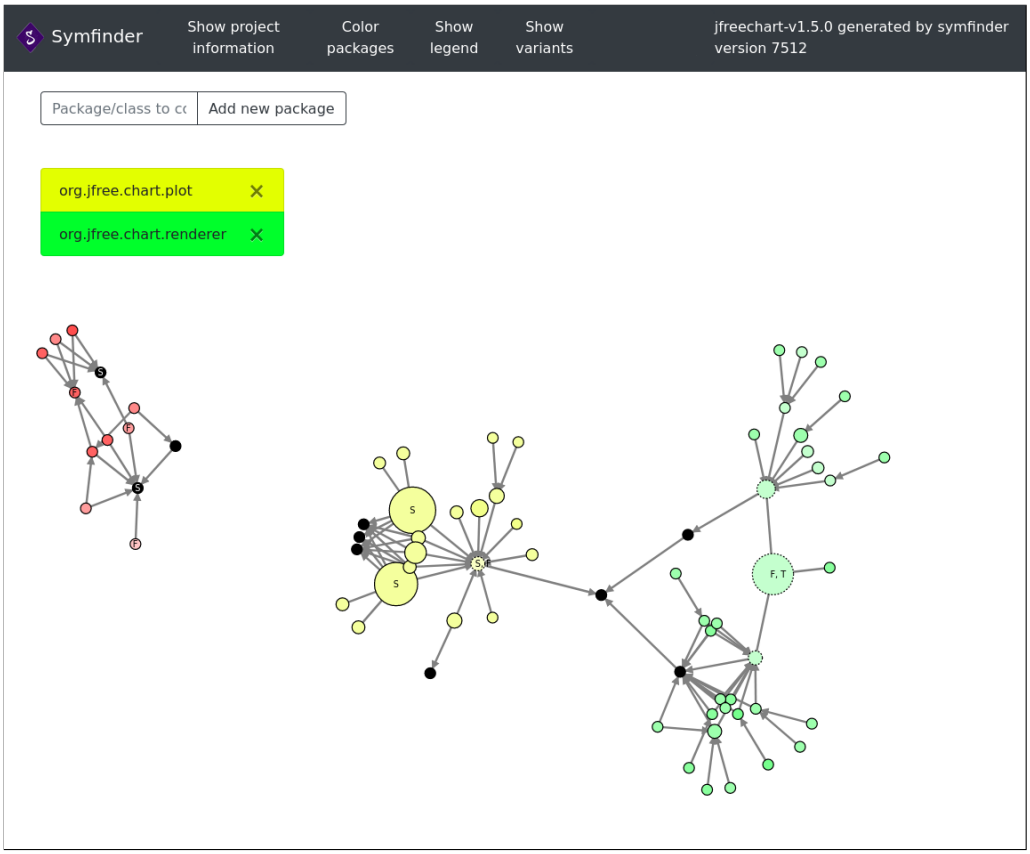


Figure 4.10: Some colored packages from JFreeChart (`org.jfree.chart.plot` in yellow and `org.jfree.chart.renderer` in green).

1. **Is the identification approach adapted to other OO languages?** We extended the identification method relying on symmetries to C++ code assets and evaluated it on a panel of five projects of various dimensions. Similarly to the analyses on Java projects conducted by Tërnava et al. [2019], some dense zones appear to represent variability implementations be relevant, while some others are use of these mechanisms for implementation purpose.

2. **Do the identified *vp*-s and variants correspond to domain features?** We measured the relevance of the identification on two systems, adapting precision and recall measures to evaluate the accuracy of the approach compared to existing features traces. It results that on these systems, although the identified *vp*-s and variants represent an important majority of the actual variability, a large number of false positives are also identified, calling for an improvement of the identification approach.

3. **To what extent does the *symfinder* approach and its visualization help comprehension of the implemented variability?** It results from the evaluation conducted with Sat4j's architect that *symfinder* helped him in understanding the implemented variability in the system thank to both the identification method and the visualization.

With this threefold evaluation, we can assess that on the studied Java and C++ systems, the symmetry-based technique identifies a majority of the variability implemented using such mecha-

nisms, providing an answer to Challenge A1. Additionally, the graph visualization allows to comprehend these variability implementations, providing an answer to Challenge A2. However, the approach still exhibits multiple limitations. While being robust, there is a strong need to improve the identification technique to increase its precision ((*i.e.*, identify less but more relevant *vp*-s and variants)). Additionally, by considering a single developer on a single system, the results of this first user evaluation cannot be generalized as it would require setting up a controlled experiment with multiple participants [Ko et al., 2015]. Nevertheless, this evaluation gave us insights on the limitations of the *symfinder* approach, both regarding the visualization (that we already started improving as detailed in Section 4.3.5) and the identification method. Therefore, prior to designing a controlled experiment with users, we extend the identification method relying on the feedback detailed in Section 4.3.2.6 by characterizing a measure of density of variability implementations to improve its precision.

CHAPTER 5

Improvement of the identification process with usage relationships

This chapter shares material with the REVE 2021 paper *"Extending the Identification of Object-Oriented Variability Implementations using Usage Relationships"* [Mortara et al., 2021c].

In the previous chapter, *symfinder* has been successfully applied to several C++ variability-rich systems, showing the applicability of the symmetry-based technique on different languages. Moreover, the application on ArgoUML-SPL has demonstrated that a large part of the identified *vp*-s and variants by *symfinder* actually implement ArgoUML's reverse engineered domain features, which can be mapped to each other. Finally, the identified *vp*-s with variants and their visualization helped Sat4j's architect to comprehend the implemented variability. However, experimenting with the *symfinder* toolchain outlined two issues in the proposed approach.

Issue 1. Detecting inheritance relationships is not enough By construction, the identified *vp*-s with variants with a heavy usage of inheritance are the most visible in the *symfinder* visualization, that is, those implemented by classes that are represented as nodes. They are simply grouped together with this relationship and the other visualized indicators, such as the size and color intensity of the nodes. While the experiments show that real *vp*-s and variants were successfully identified by applying *symfinder* on the subject systems it also appeared that their visualized density based only on the inheritance relationship is not always sufficient to comprehend the variability of a system.

To exemplify this issue, we show in Figure 5.1 an excerpt of JFreeChart class diagram. Two classes, `MeterNeedle` and `Plot`, possess subclasses and will be identified by *symfinder* as two potential *vp*-s with their variants. In the *symfinder* visualization, they are shown as two separate trees of variability, as illustrated in Figure 5.2. However, looking in their source code, it can be noticed that these classes are highly dependent on each other, as the `CompassPlot`, a variant of the *vp* `Plot`, uses every variant of the *vp* `MeterNeedle`. Naturally, we can find many of these additional dependencies at class level between potential *vp*-s and variants. For instance, in Figure 4.8b, displaying only inheritance relationships does not allow visualizing that `LearnedConstraintsDeletionStrategy` and `ConflictTimer` are part of a common design (*cf.* Section 4.3.2.5). Therefore, by not considering usage relationships between classes, the

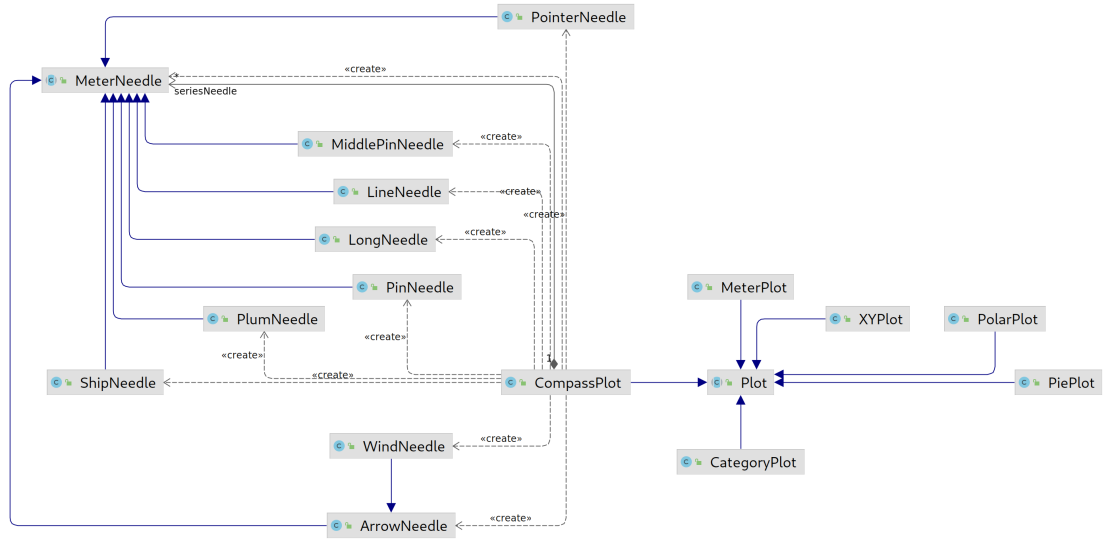


Figure 5.1: Excerpt of JFreeChart's class diagram. `CompassPlot` uses `MeterNeedle` and its variants.

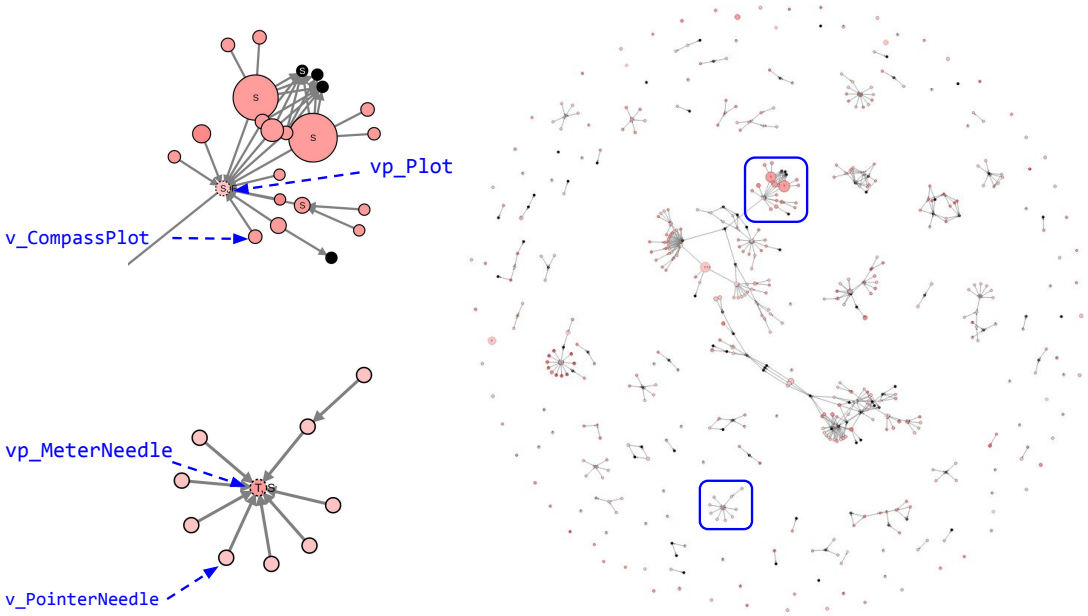


Figure 5.2: *symfinder* view of JFreeChart. `CompassPlot` and `MeterNeedle` appear in two distant trees.

identification technique identifies as isolated variability implementations that are in reality related, reopening Challenge A1.

Besides, according to the study of symmetry in software constructs by Zhao and Coplien [2003], reusability in object-oriented systems is described as being about the *instantiation* of templates, *composition* of the instances, and *substitution* of the instances. In *symfinder*, most of the seven considered traditional techniques provide an implementation of variability through the substitution of the instances, as they have a substitution symmetry [Tërnava et al., 2019]. However, composition, or more generally, usage of other instance, is also applied for reuse of code assets. For example, many software design patterns rely on inheritance and composition to characterize complex reusable designs [Freeman et al., 2008].

Therefore, it seems necessary to extend *symfinder* to consider the composition relationship of classes when visualizing the identified *vp*-s and variants. In the following, we will consider this relationship in the broad sense of the term, as an *usage relationship*, encompassing cases where a class uses another class in attributes and parameters of its methods. We expect that a visualization with inheritance and composition relationships between classes that are identified as potential *vp*-s or variants will improve their consistency and help users to better comprehend their dependencies.

Issue 2. Large systems are hardly comprehensible With its zooming, filtering, and hovering capabilities, the *symfinder* visualization naturally relies on the Shneiderman visual information seeking mantra [Shneiderman, 2003]: *overview first, zoom and filter, then details on demand*. For example, class level variants can be shown or hidden. Other options in the visualization, such as the total number of potential *vp*-s, also provide information for the overall variability of the targeted system. Nevertheless, in many systems and especially in large ones, users gave us the feedback that they were missing some clear entry points to start browsing the visualization. For instance, Figure 5.3 shows the complete *symfinder* view of all the *vp*-s and variants identified in the NetBeans IDE codebase, exhibiting about 3498 nodes and an important number of trees, making it hard to determine where to start the exploration. Consequently, the visualization needs improvement to allow comprehension on large systems, reopening Challenge A2.

After analyzing this problem, we realized that the vast majority of studied systems exposed facade classes, which were natural entry points expected by the users, or even a well-defined *application programming interface* (API) [Robillard et al., 2012] where their reusable and customizable functionalities are textually exposed. The second issue we thus identify is the need for entry points to be specified and exploited in the visualization to facilitate the variability comprehension.

In this chapter, we describe the different extensions made on the identification and visualization parts to build a new version of *symfinder* named *symfinder-2* (Section 5.1). During the identification of potential variation points with variants, we take into account their *usage relationships* so to display them in the visualization. We also refine the visualization so that entry point classes of a targeted system, such as API classes, can be used to find more easily the important zones of variability. Finally, a *parameterized density* metric allows to automatically identify *hotspots*, being classes that are part of dense zones of variability implementations, and that can also be used to filter the visualization. We apply *symfinder-2* to ten Java-based variability-rich systems and observed the impact of our changes and done improvements (Section 5.2). We notably evaluate the visualized graph, the remaining *vp*-s and variants, and the scalability of our extension. We finally discuss related work (Section 5.3).

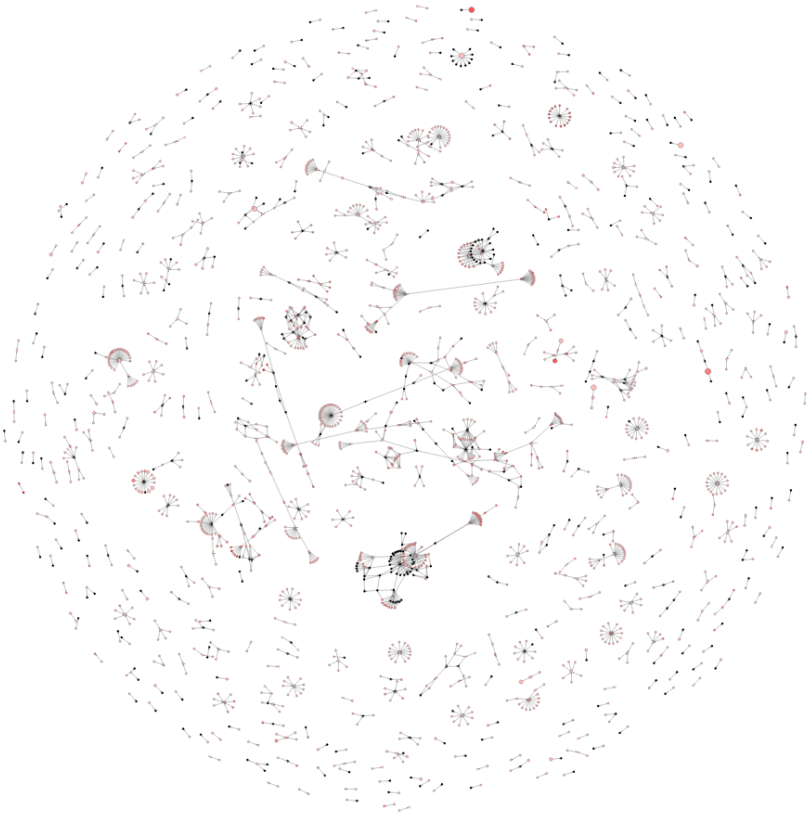


Figure 5.3: *symfinder* visualization of NetBeans 12.1 with isolated nodes filtered out

5.1 *symfinder-2*

In order to better address Challenges A1 and A2, we extended the variability identification and visualization by *symfinder*. In the following, we describe these extensions, which makes up *symfinder-2*. Its sources and conducted experiments are publicly available on *symfinder-2*'s website[1] and in a reproduction package [Mortara et al., 2021d].

5.1.1 Handling the usage relationships

To address **Issue 1**, we first improved the identification step in *symfinder* by identifying the *usage relationship*s between the identified *vp*-s with variants. In *symfinder*, the target codebase is analysed and stored in a graph representation. Classes and methods represent nodes in the graph, linked together through relationships of different types used when querying the database to identify the local symmetries, leading to potential *vp*-s with variants (details about the technical implementation of *symfinder* can be found in Section 3.1.2). In the extended identification process implemented in *symfinder-2*, a `class A` is *used* by another `class B` if the `class B` is used as a field type or method parameter type in the `class A` [Freeman et al., 2008]. Technically, at parsing time (*cf.* step 3 on Figure 4.1), each usage relationship is identified and an `USE` relationship is created in the graph database between the respective class level *vp*-s or variants that are identified. For instance, in the

[1] *symfinder-2*'s website: https://deathstar3.github.io/symfinder2-demo/

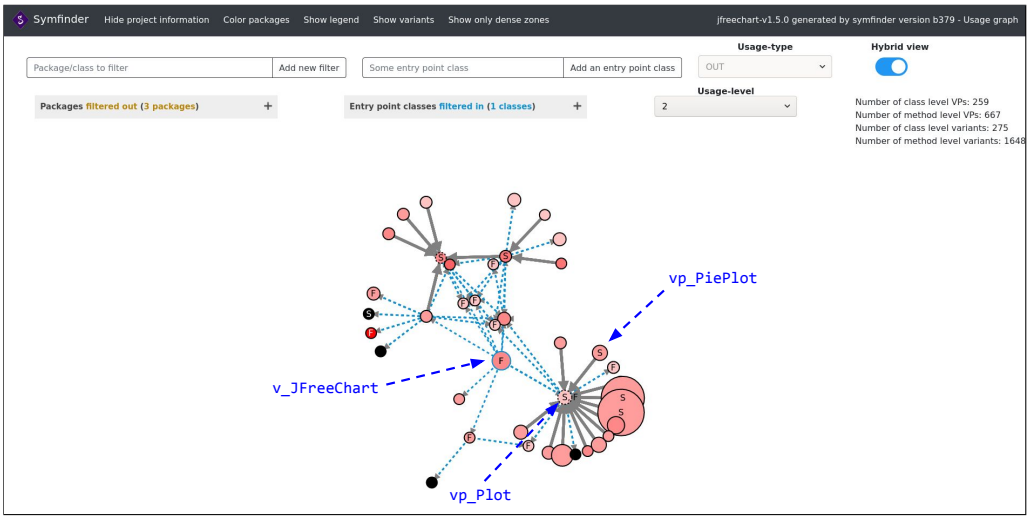


Figure 5.4: The identified *vp*-s and variants using *symfinder-2* for the excerpt of JFreeChart given in Figure 3.1

example of JFreeChart in Figure 3.1, class `Plot` is used by class `JFreeChart` as it is referenced by a field inside the `Plot` class. Therefore, each of these classes will be identified by *symfinder-2* and will be visualized with their usage relationship.

In general, all identified usage relationships between the potential *vp*-s, variants, and their additional information (such as the number of variants at class and method level, and the classes to which they are linked through usage relationships), are exported into a JSON format, which data are further used for variability visualization by *symfinder-2* (*cf.* Figure 5.5). Visually, we show in *symfinder-2* both inheritance and usage relationships, where inheritance relationships are grey arrows and usage relationships are represented as dashed blue arrows. For example, the new visualization for the JFreeChart example, given in Figure 3.1, is shown in Figure 5.4. In comparison with its prior visualization by *symfinder*, given in Figure 3.3, both relationships are represented. Therefore, for instance, the usage relationship between `Plot` and `JFreeChart` from Figure 3.1 is now explicit in Figure 5.4.

5.1.2 Handling entry points

To address **Issue 2**, we extended the *symfinder* visualization to take into account some entry points during the variability comprehension of a targeted system. In the following, we present the two kinds of entry points that we make available in the extended version of *symfinder-2*.

***vp*-s and variants under analysis** As a first kind of entry points, we add the possibility to visualize only desired *vp*-s with variants. For this reason we added a textbox with a button in the visualization, named `Add an entry point class`, as shown in Figure 5.4. In that case, starting from the classes designated as entry points, *symfinder-2* will also show the *vp*-s with variants connected to them through inheritance or usage relationships. For example, from the shown data in Figure 5.4, 926 potential *vp*-s at class and method levels are identified in JFreeChart. To show the related variability only to the `vp_Plot` and `v_JFreeChart` given in Figure 5.4, one can add their respective class paths in the dedicated textbox. In this example, shown classes are the entry

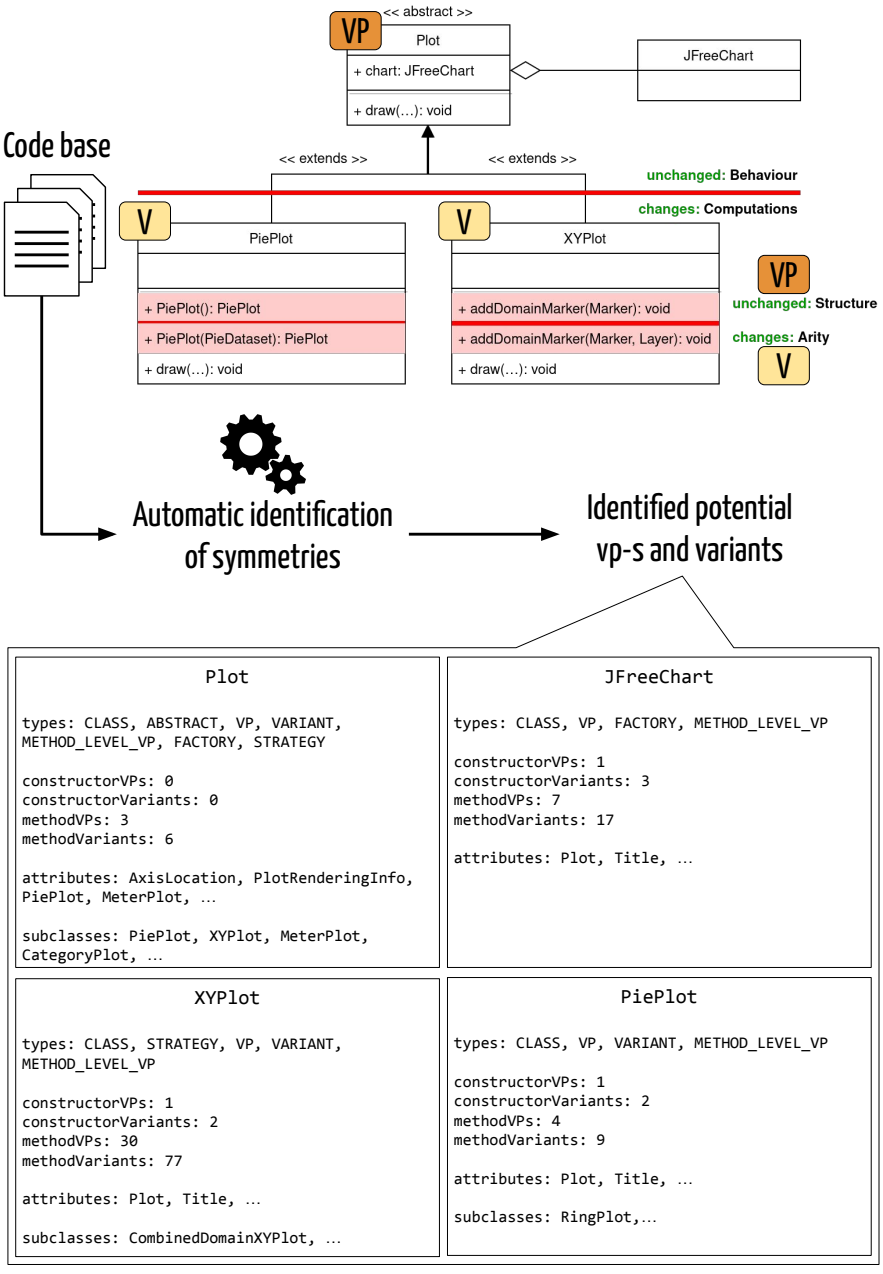


Figure 5.5: Symmetries in object-oriented code and metrics that can be extracted

points (`vp_Plot` and `v_JFreeChart`) and all the *vp*-s with variants connected to them through inheritance or usage relationships. The information about the potential *vp*-s or variants that are entry points will remain listed in the drop-down list named `Entry point classes filtered in`. This filtering capability can help users to choose interesting entry points in the visualization and to comprehend progressively the whole identified variability of a system under study.

API points Another kind of entry point that we implemented in *symfinder-2* is the usage of classes that are annotated by the API Guardian library[2] as specific API entry points. Any potential *vp* or variant that is realized by these API classes is automatically detected at parsing time and put in the *symfinder* data model. The visualization then automatically adds all detected classes as entry points so that they can be used to refine it as described in the previous paragraph.

5.1.3 Improving the readibility of the visualization

As stated in the introduction of this chapter, the *Shneiderman information seeking mantra* [Shneiderman, 2003] relies on three principles: *overview first, zoom and filter, then details on demand*. However, with both inheritance and usage relationships being visualized, *symfinder-2* displays all the information altogether, eventually leading to too many edges displayed and hampering the comprehension of variability. We thus propose three filtering techniques allowing to refine the *symfinder-2* visualization.

Filtering inheritance and usage relationships. In order to reduce the number of displayed edges of usage relationships, *symfinder-2* allows to filter the direction of usage, so that an edge is displayed only if one class being a potential *vp*-s or variant is using another one (`OUT`), or used by this other one (`IN`). A `usage-type` can thus be set to display only incoming (`IN`), outcoming (`OUT`), or all (`IN/OUT`) usage relationships between potential *vp*-s and variants. For example, the usage relationship between `v_JFreeChart` (here, the entrypoint) and `vp_Plot` (a class it uses) is shown in Figure 5.4 as the `usage-type` is set to `OUT`. From there, all other displayed *vp*-s and variants can be added to the entry points list and be used as a starting point to comprehend the identified variability of JFreeChart.

Tuning the usage level Depending on how the object-oriented targeted system is designed, there can be different layers of objects being composed to implement some variability. The visualization is thus adaptable to how the potential *vp*-s with variants could be related at a different level of usage. We added a `usage-level` drop-down list box with values corresponding up to the maximum number of usage relationships from one of the given potential *vp*-s or variants to the others in a targeted system. For example, a usage level of 4 will display all potential *vp*-s or variants that have a usage relationship to one of the potential *vp*-s or variants through at most 4 detected usages transitively. This enables one to display more or less related variability implementation classes. One can then start by a low level and expand progressively to tame the complexity of displaying too many relationships. In Figure 5.4, the `usage-level` is 2, meaning that the shown classes are *vp*-s and variants used by the entrypoint `v_JFreeChart`, or by *vp*-s and variants used by `v_JFreeChart`.

[2] https://github.com/apiguardian-team/apiguardian

Introducing a density metric In order to tackle globally both issues, we propose to filter out the less dense zones with potential *vp*-s and variants in the visualization. This idea has its roots in the center's theory [Alexander, 2002], which states that the density of local symmetries in a structure is important and can be used to easily distinguish, memorize, and describe that structure. We thus introduce in *symfinder-2* a *density* metric used with thresholds to filter nodes in the visualization. Using it, we want to show whether the density of local symmetries (*a.k.a.*, potential *vp*-s and variants) under their two types of relationships in the visualization of a system can help to comprehend its implemented variability. For instance, the visualization from JFreeChart given in Figure 3.3 shows two places with different densities of potential *vp*-s and variants. The left one seems denser than the right one with the *inheritance-only symfinder*, whereas when usage relationships are considered using *symfinder-2*, both of these places are interrelated and create a new denser zone of potential *vp*-s and variants.

To better understand its realization, we introduce several definitions allowing to formally describe the density metric. This represents any variability-related class in a system under study, *i.e.*, a class with some variability, at the class level, and/or at the method level, and/or being a variant of another *vp*.

Definition 5.1 (Class and Variability Implementation)**.** $\mathcal{C}$ represents the set of classes of the system. A class $c \in \mathcal{C}$ is defined as a tuple

$$c = \langle nbVar_{class}, nbVar_{method} \rangle$$

where $nbVar_{class}$ is the number of variants of v at class level, and $nbVar_{method}$ is the number of variants of v at method level.

We define the set of all identified variability implementations $\mathcal{V} \subseteq \mathcal{C}$ the subset of classes having at least 2 variants at class or method level.

$$\mathcal{V} = \{c \in \mathcal{C} \mid (c.nbVar_{class} \geq 2) \vee (c.nbVar_{method} \geq 2)\}$$

Definition 5.2 (Usage graph)**.** The usage graph of a system is a graph $G = (\mathcal{C}, \mathcal{R})$ where $\mathcal{C}$ is the set of classes of the system (being the vertices of the graph), and $\mathcal{R}$ the set of usage relationships between classes (being the edges of the graph).

Definition 5.3 (Individually dense class)**.** A specific variability implementation $v \in \mathcal{V}$ is individually dense if, given a threshold $minVars$, v has a minimum of $minVars$ variants at class or method level.

$$ID_{minVars}(v) = (v.nbVar_{class} \geq minVars) \vee (v.nbVar_{method} \geq minVars)$$

For example, in Figure 5.5, `XYPlot` has 77 method variants and 2 constructor variants, making it 79 method level variants (*i.e.*, $\texttt{XYPlot}.nbVar_{method} = 79$). Therefore, $ID_{minVars}(\texttt{XYPlot})$ is true for $minVars \leq 79$, regardless of its number of class level variants (*i.e.*, its number of subclasses).

Definition 5.4 (Collectively dense class)**.** Given a usage graph $G = (\mathcal{C}, \mathcal{R})$, and $d(c_1, c_2)$ the distance between two classes in the graph. A class $c \in \mathcal{C}$ is collectively dense if, given a threshold $maxDist$, there is at least a variability implementation v distant of maximum $maxDist$ from c.

$$ED_{maxDist}(c) \Leftrightarrow \exists v \mid (v \in \mathcal{V}) \wedge (d(c, v) \leq maxDist)$$

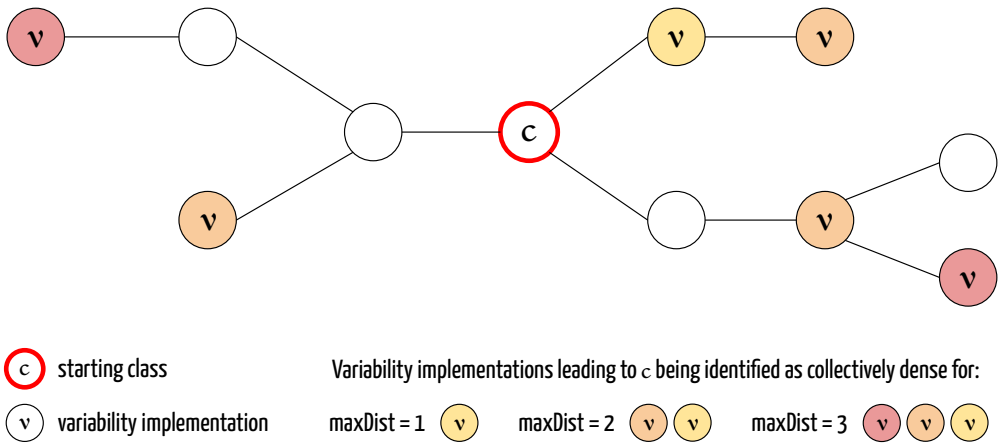


Figure 5.6: Depiction of the collectively dense class identification technique in a sample usage graph. Nodes represent classes, and edges usage relationships (*i.e.*, presence of the type as a class attribute or a method parameter).

For example, for $maxDist = 1$, a class c is collectively dense if at least one class using or being used by c is a variability implementation. For $maxDist = 2$, a class c is collectively dense if at least one class using or being used by c, or one class using or being used by these classes is a variability implementation. An illustration of this mechanism is given in Figure 5.6.

Definition 5.5 (Density)**.** We define a measure of density $\Delta : \mathcal{C} \rightarrow \{true, false\}$ which, given a class $c \in \mathcal{C}$, determines if it is a *hotspot* (*i.e.*, if it is part of a dense zone of variability implementations) by checking if c is either individually or collectively dense.

$$\Delta_{minVar,maxDist}(c) = ID_{minVars}(c) \vee ED_{maxDist}(c)$$

The set of variability implementations in dense zones for a given couple of parameters $(minVar, maxDist)$ is thus given by

$$S = \{v \in \mathcal{V} \mid \Delta_{minVar,maxDist}(v)\}$$

The density metric is directly dependent on its two input parameters. Consequently, defining a threshold over each of these two parameters enables to automatically identify *hotspots*, being zones exhibiting a density above the thresholds, hence a high concentration of variability implementation techniques. Relying on this metric, the potential *vp*-s and variants that are under the set threshold are colored in a light gray on the visualization generated by *symfinder-2*. In a further step, these potential *vp*-s and variants can also be excluded from the visualization by using the button `Show only dense zones` (*cf.* Figure 5.4). In this example, we set a threshold with the number of variants in a potential *vp* ≥ 5 and the usage relationships between the potential *vp*-s or variants ≤ 2 (in other words, the set S of variability implementations being *hotspots* is characterized by $S = \{v \in \mathcal{V} \mid \Delta_{5,2}(v)\}$). Hence, in Figure 5.4 the `v_DefaultDrawingSupplier` variant is highlighted in gray as its `vp_DrawingSupplier` *vp* has less than 5 variants.

Adapting these thresholds tailors the identification to the analyzed codebase which, depending on its size and the object coupling induced by its architecture and used frameworks, might result in too many/too few hotspot classes being identified. As a result, with information on symmetries,

metrics on their occurrences, as well as relations of inheritance and usage between classes, zones concentrating *hotspots* can be predetermined.

5.2 Evaluation

In this section, we first define the research questions to evaluate the extended approach of *symfinder-2* (Section 5.2.1). We then introduce the subject systems selected for evaluation (Section 5.2.2) before tackling the research questions successively. The tool and the used data to obtain the presented results are available online in a reproduction package [Mortara et al., 2021d].

5.2.1 Research questions

We define four research questions to evaluate our approach.

RQ_1 **: Does the identification of usage relationships improve the variability visualization of a given system by *symfinder-2*?** The visualization generated by *symfinder* represents class level *vp*-s with variants as nodes, linked together through inheritance relationships. This forms tree-like structures (*cf.* Figure 3.3). For example, two sets of *vp*-s both linked through a few inheritance relationships would appear as two small trees on the visualization. If potential *vp*-s from these two sets make use of each other, they form a more important concentration of variability implementations, which was not put together in the first version of *symfinder* visualization. Similarly, a *vp* with only method-level variability would be isolated on *symfinder*'s visualization, but might also be used by other class level *vp*-s or variants and should be linked to them on *symfinder-2*'s visualization. In *symfinder-2*, we expect that the two previous small zones would appear as a single but as a more important tree on the visualization, being more visible for the user.

RQ_2 **: What is the starting density threshold to begin with the comprehension of the visualized variability by *symfinder-2*?** In addition to the two basic user-defined variability filtering capabilities that are added in the visualization and can be activated interactively, the third filtering capability is based on the density of potential *vp*-s and variants. As explained in Section 5.1.2, our filtering by density has for objective to reduce the number of potential *vp*-s and variants (*a.k.a.*, nodes) visible on the visualization through identifying those *vp*-s that have a minimum number of variants or those that are linked to another *vp* through a maximum number of usage relationships. This threshold is set before their identification, and setting it is not trivial as the user needs to know which is the right threshold to start with.

RQ_3 **: Is the API information of a given system useful to simplify its identified variability by *symfinder-2*?** When studying large systems, the number of identified potential *vp*-s with variants becomes extensive. Consequently, analysing and comprehending the implemented variability from the provided visualization can be really difficult. With this research question, we aim to evaluate whether classes annotated as APIs in the system are good candidates for filtering the visualization and improve its comprehension. But, as a testing framework, Cucumber has an API that exposes classes for defining the dependency steps and an object factory for customizing the dependency injections.

Table 5.1: The ten variability-rich subject systems.

Subject system	Version	LoC	# *vp*-s	# *variants*	API	Type
Java AWT	jb8u202-b1532	69,974	795	1,706	Documented	Library
Apache CXF	3.2.7	48,655	3,403	7,625	Documented	Framework
JUnit	r4.12	7,717	109	245	Documented	Framework
Maven	3.6.0	105,342	612	1,147	Documented	Application
JFreeChart	v1.5.0	94,384	926	1,923	Documented	Library
ArgoUML	bcae373	134,367	776	1,959	Documented	Application
Cucumber	v6.8.0	42,662	238	282	Annotated	Framework
Logbook	2.2.1	16,210	96	162	Annotated	Library
Riptide	2.11.0	12,626	102	218	Annotated	Library
NetBeans	12.1	5,058,448	3,621	6,736	Documented	Application

RQ_4 **: Does the identification of usage relationships impact the scalability of *symfinder-2*?** This question aims at determining whether the additional functionalities in *symfinder-2* harm its scalability. The impact could be located in the identification phase, as the usage relationship is identified in the source code representation, as well as in the visualization phase, where new elements are computed to filter all displayed elements.

5.2.2 Subject systems

To evaluate *symfinder-2*, we chose ten popular variability-rich subject systems, being Java applications, frameworks, or libraries (*cf.* Table 5.1). For the time frame of up to twelve last years, they have received between 150 and 8,000 stars in GitHub, but we particularly considered them because of the following criteria. The first six ones were already used to evaluate the first version of *symfinder* in [Tërnava et al., 2019], namely **Java AWT**, **Apache CXF**, **JUnit**, **Apache Maven**, **JFreeChart**, and **ArgoUML**. Then, we chose the other three as they use in their codebase a form of API annotations, namely the API Guardian library, to annotate each code unit that constitutes their API. These new chosen systems are **Cucumber** – a framework for BDD testing, **Logbook** – a library to enable logging for different client- and server-side technologies, and **Riptide** – a library based on Spring to implement client-side response routing. Finally, we selected the **NetBeans IDE** because of its size with about 5M lines of code (LoC), which helps in evaluating the scalability issues of both the approach and the prototyped toolchain.

5.2.3 RQ_1: Improved visualization

To answer RQ_1, we applied *symfinder* and *symfinder-2* to all ten subject systems and compared their respective visualizations. Results are given in Table 5.2.

On all studied systems, we notice a smaller number of disconnected graphs and isolated nodes by *symfinder-2* for the same number of nodes displayed, meaning that the zones in the visualization that seemed previously uncorrelated are now linked through usage relationships and appear as such on the new visualization. Such an example is given on Figure 5.4, where the disconnected graph having `vp_MeterNeedle` as *vp* and the disconnected graph of `vp_CompassPlot` presented on Figure 5.2 are now grouped as a single one. We observe that the difference between the number of disconnected graphs is not proportional to the size of the studied system. For instance,

Table 5.2: Comparison of the number of disconnected graphs and isolated nodes with *symfinder* and *symfinder-2*

Subject	Nodes	*symfinder*		*symfinder-2*	
		# graphs	# isolated nodes	# graphs	# isolated nodes
Java AWT	431	55	142	2	20
Apache CXF	3085	473	1149	105	500
JUnit	118	23	36	6	18
Maven	616	177	172	21	79
JFreeChart	578	54	167	5	51
ArgoUML	1270	123	460	38	183
Cucumber	331	45	122	14	50
Logbook	117	19	40	4	16
Riptide	89	20	37	8	19
NetBeans	3498	504	1666	195	836

the number of NetBeans' graphs are reduced by 61% whereas JFreeChart's graphs are reduced by 90%. However, their number could be related to the architecture of the project. A project of an important size may have an architecture in layers, limiting the number of interactions between classes, and therefore exhibit fewer usage relationships. Besides, we notice that some isolated nodes still appear on *symfinder-2*'s visualization. This may suggest that other usage mechanisms are present in the studied systems [Lau and Rana, 2010]. Taking into account these specific types of usage relationships is part of our future work. Further, it is important to emphasize that although the visualizations by *symfinder* and *symfinder-2* have different numbers of disconnected graphs, their overall number and kinds of identified *vp*-s with variants remain unchanged. This indicates that *symfinder-2* is an extension of *symfinder* with an intact variability identification. To conclude, the reduced number of disconnected graphs by *symfinder-2* shows an improved and denser visualization of the identified *vp*-s and variants for a given system.

5.2.4 RQ_2: Starting density threshold

To answer RQ_2, we run *symfinder-2* on each subject system with three different density settings (*cf.* Definition 5.5):

$\Delta_{5,3}$ ($\geq$ 5 variants and $\leq$ 3 usage hops), $\Delta_{10,3}$ ($\geq$ 10 variants and $\leq$ 3 usage hops), and $\Delta_{30,2}$ ($\geq$ 30 variants and $\leq$ 2 usage hops). We have carefully chosen these parameters values, based on a previous empirical evaluation with ArgoUML, JFreeChart, and Java AWT, and for which we have the best knowledge to manually evaluate the impact of the density threshold. By increasing the threshold on the number of variants, we aim to consider only highly-dense *vp*-s, whereas by decreasing the threshold on the usage hops between such *vp*-s, we aim to consider only highly-dense *vp*-s which are close in terms of usage relationships. The obtained results are given in Table 5.3.

It can be observed that, in all subject systems, fewer nodes are displayed when using any of the three density settings. For instance, JFreeChart has in total 578 identified *vp*-s with variants at class level. After applying the three density settings, 34, 15, and 3 *vp*-s and variants (*i.e.*, entry points) remain, respectively. Moreover, the number of remaining *vp*-s decreases as we increase

Table 5.3: Number of nodes identified as being part of dense zones compared to the total number of nodes in all subject systems

Project	***symfinder***	***symfinder-2***		
		$\Delta_{5,3}$	$\Delta_{10,3}$	$\Delta_{30,2}$
Java AWT	431	28	22	3
Apache CXF	3085	98	32	4
JUnit	118	5	0	0
Maven	616	8	1	0
JFreeChart	578	34	15	3
ArgoUML	1270	40	15	3
Cucumber	331	4	0	0
Logbook	117	0	0	0
Riptide	89	0	0	0
NetBeans	3498	58	22	2

the minimum number of variants for a *vp* and decrease the number of usage relationships between both of them. However, these values are not adapted to such projects like Logbook or Riptide, for which no *vp*-s remain with these three default thresholds.

These results suggest that determining a set of appropriate values for the parameters when setting the density threshold is highly dependent on the studied project's characteristics. That is, even some large projects in terms of lines of code, such as Maven, can have a considerable number of potential *vp*-s but with few variants. For such reason, setting a high density threshold may filter out most or all potential *vp*-s with variants (*i.e.*, there will be no entry points). Based on our experiments with these ten subjects, we conclude that the first density setting (*i.e.*, $\Delta_{5,3}$) can be used as a good versatile starting point to begin the exploration and comprehension of the identified variability by *symfinder-2*.

5.2.5 RQ_3: Usefulness of API-based filtering

To answer RQ_3, we ran *symfinder-2* on three subject systems, namely Cucumber, Logbook, and Riptide, while taking into account the code units annotated by developers using the API Guardian library. We then compared the number of nodes that are displayed in their visualizations before and after using their respective API to filter in the related classes. Results are given in Figure 5.7.

It can be observed that in all three subjects the visualization with the applied API has notably fewer nodes than the original visualization by *symfinder-2*. We manually checked that, while reducing the number of potential *vp*-s with variants, it always shows those that are considered as the most relevant due to their nomenclatures. They can help to comprehend each system's variability, that is, to give us an insight on the implemented domain variability. In the three cases, these *vp*-s with variants can be used as entry points for users in order to begin with the variability comprehension of the systems. We interpret this filtering by an API as facilitation in the overview and zooming parts of the Shneiderman mantra [Shneiderman, 2003]: *overview first, zoom and filter, then details on demand*. In the longer term, we believe that this should be extended by the integration of the *symfinder-2* toolchain with other sources that contain variability information for

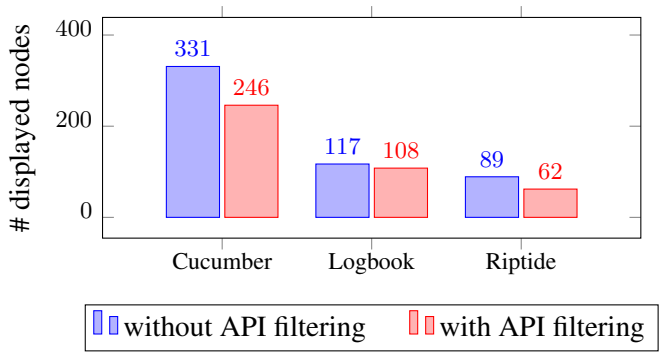


Figure 5.7: Number of *vp*-s and variants (nodes) displayed on the visualization before and after refinement by API

a given system, or simply to show how different variability information sources could be blended together.

5.2.6 RQ_4: Scalability

To answer RQ_4, we measured the time taken to identify and visualize the variability in all ten subject systems. We conducted our experiments on a Linux environment running Arch Linux 5.11.12-arch1-1 x64 with Intel i7-9850H (12 cores) @ 4.6GHz and 32Go memory. The visualizations are tested using Mozilla Firefox 87.0 and Google Chrome 89.0.4389.114.

We noticed that the computation needed to render and display the visualization is not very time consuming, with 700 ms on Chrome and 850 ms on Firefox for the NetBeans system. The identification step is clearly the most time-consuming activity. We hence measured and compared this time on the ten subject systems with *symfinder* and *symfinder-2*. Figure 5.8 summarizes the obtained execution times for both versions with a density threshold of $\Delta_{5,3}$. Although the execution time with *symfinder-2* is higher than with *symfinder* for every system, we observe that the difference increases with the size of a system and number of identified *vp*-s and variants, for instance, 20% of difference for Riptide (23 sec $\rightarrow$ 28 sec) and 85% of difference for NetBeans (01:02:10 $\rightarrow$ 01:55:04). This can be explained by the fact that a higher number of relationships between classes needs to be parsed and treated by *symfinder-2* in its database. While there seems to have an exponential evolution *w.r.t.* LoC with systems of the size of NetBeans (5M LoC), we believe the analysis step is still adapted to large systems, as *symfinder* was also successfully applied to Firefox and its 25M LoC (*cf.* Section 4.1.3). Then, waiting for around 1 or 2 hours to run *symfinder-2* only on the new releases of a project, for example, every 6 months, is affordable.

5.2.7 Discussion and threats to validity

Summary of $RQ_1 - RQ_4$ On our set of studied systems, *symfinder-2* provides a more focused identification and visualization of relationships among the potential *vp*-s with variants than *symfinder*. Depending on the system's size, *symfinder-2* can take between 30 seconds to 2 hours to identify between 250 and 11K potential *vp*-s with variants. Besides, it supports users with up to four ways to begin the variability comprehension of a given system from its visualization. In particular, our experiments suggest using a density threshold of $\Delta_{5,3}$ (*i.e.*, ≥ 5 variants and ≤ 3 usage hops) or, if available, the API-based filtering.

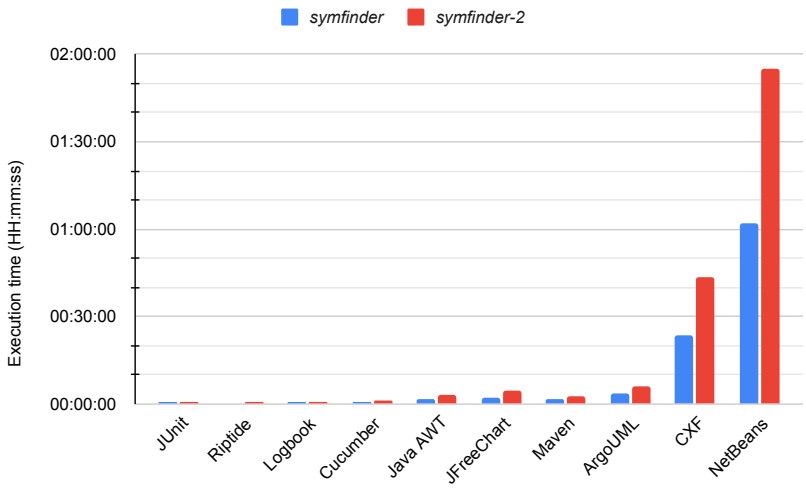


Figure 5.8: Execution time in *symfinder* and *symfinder-2*

Internal threats to validity. A first internal threat concerns the distinction of real *vp*-s and variants from the potential ones proposed by the different versions of *symfinder*. For the considered subject systems (except NetBeans that was too large), we manually determined whether the remained *vp*-s and variants after applying the thresholds represent some variability implementations. We thus did a sample verification by examining identified classes, checking for their documentation on the project website, and devising the kind of variability that was implemented. Although we could be partially wrong in our interpretation, this manual verification allowed to obtain relevant results. Then, determining whether an identified *vp* or variant actually implements some domain variability was hard to conclude as none of the subject systems, except ArgoUML, had a ground truth.

External threats to validity. To address the research questions, we used up to 10 subject systems, which vary across domains, size, type, and developers. While the dataset is still small, we have good confidence that the obtained results also apply to other Java-based variability-rich systems of mid-size. Besides, our experiments over NetBeans show that while the toolchain is likely to scale on very large systems, the proposed improvements in *symfinder-2* are not sufficient to comprehend all the implemented variabilities. Entry points and the usage links enable to provide a better overview and better filtering over the system, but it is still difficult to browse effectively towards a comprehension of the system variability.

5.3 Related Work

Outside the software variability domain, some other works rely on the identification of inheritance and composition to define *hotspots*, being zones of an object-oriented design that are exposed to client software and hence have to be comprehended to ensure reuse [Schauer et al., 1999; Flores et al., 2005]. While this identification relies on design patterns detection used to implement reusable interfaces, it was not related to variability implementations.

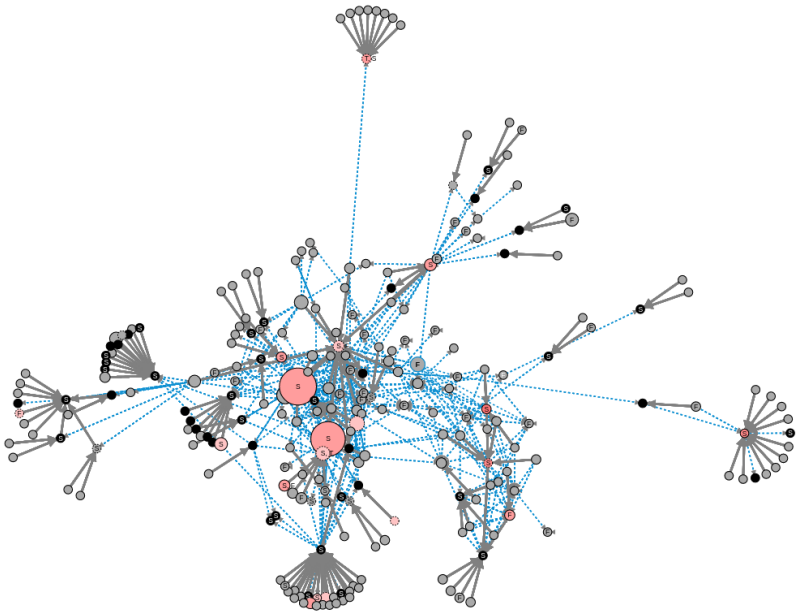


Figure 5.9: *symfinder-2* view of JFreeChart, having `JFreeChart` as an entrypoint, `OUT` as usage orientation and a usage level of 4.

Previous studies on software API comprehension focus on the extraction of usage patterns relying on unit tests analysis [Zhu et al., 2014], client code analysis [Zhong et al., 2009], or approaches combining client and library code analysis [Saied and Sahraoui, 2016], as well as their evolution in time [Huppe et al., 2017; Saied et al., 2020] to help developers in library reuse. In the variability domain, APIs have been studied for the variability of their evolution [Alrubaye et al., 2019], but not to facilitate the comprehension of variability implementations through a visualization approach as in our work.

5.4 Conclusion

The identification method provided by *symfinder* is incomplete by not taking into account usage relationships between classes and thus cannot provide appropriate means to help users start the comprehension activity. In *symfinder-2*, we extended this method with usage relationships and provided filtering capabilities based on both automatically detected API-related entry points, and a user-defined density metrics based on two thresholds. Application to ten systems has shown that *symfinder-2* provides a more focused identification and visualization of object-oriented variability implementations. Although determining adequate parameters for the density is project-dependent and thus cannot be done automatically, adapting the density thresholds allows to show a smaller but still relevant set of classes. Therefore, the improvements brought by *symfinder-2* compared to *symfinder* allow to better answer Challenges A1 and A2 (*"Identifying variability implemented in OO software systems"* and *"Making the identified variability implementations comprehensible"* respectively).

Although this improved visualization helps to understand the implemented variability at fine grain, it does not solve the scalability issue of the view that we identified in Chapter 4. As an example, Figure 5.9 shows an excerpt of the generated visualization for JFreeChart. The view exhibits only outgoing usage relationships on 4 usage levels out of a maximum of 8 and is already hard to interpret. There is therefore a need to change the shape of the visualization for another metaphor that would better represent the variability-related metrics of each class and their relationships between each other to reveal dense zones of variability implementations.

CHAPTER 6

Comprehending the organization of the implemented variability

This chapter shares material with the VISSOFT 2021 paper *"Visualization of Object-Oriented Variability Implementations as Cities"* [Mortara et al., 2021a].

As detailed in Section 3.1.1, the implemented variability in OO code assets is neither explicit nor documented, which hinders its management, but more globally, hampers the simple comprehension of it. As features to be understood are not known in advance, and the code is not cloned and modified per product, none of the feature location techniques [Dit et al., 2013; Michelon et al., 2021a; Linsbauer et al., 2022] can be applied in this context. We thus advocate that our context naturally calls for visualization-based solutions [Koschke, 2003; Storey et al., 2005; Teyseyre and Campo, 2008]. While visual representations have already been proposed in the variability management field, they most entirely focus on the domain variability and feature models [Lopez-Herrejon et al., 2018; Kästner et al., 2008; Bergel et al., 2021; Andam et al., 2017; Greevy et al., 2005] and are therefore not applicable to our context.

As inheritance relationships between classes (one of the mechanisms used to implement variability in OO systems) are often represented and manipulated as a graph [Snyder, 1986; Booch et al., 2008] we used and adapted this representation to visualize the identified variability implementations in *symfinder* and *symfinder-2*. However, it results that due to the complex nature and the diversity of OO variability implementation mechanisms, representing all the information required to understand them in the form of a graph leads to a visualization that becomes quickly overloaded and thus does not scale to large codebases. There is therefore a need for a more adapted visualization capable of bringing an answer to the A2 challenge, that is, to represent these variability implementations and make them understandable by developers and architects. Such visualizations often rely on metaphors as they help to represent concepts in a comprehensible manner [Knight and Munro, 2000]. For example, the city metaphor [Wettel and Lanza, 2007] has been shown to scale on large projects for visualizing metrics related to software quality and received multiple adaptations to various contexts (*cf.* Section 3.2).

In this chapter, we address the following questions:

1. **What are the requirements for a visualization approach to comprehend OO variability implementations?** As program comprehension is seen as a process of both information

seeking [Sillito et al., 2008] and feature location [Dit et al., 2013], it is obvious that even if our problem is not related to *domain features* in a classic SPL terminology, identifying *vp*-s with variants is indeed a comprehension problem. In Section 6.1, we identify the requirements that such a visualization should meet and propose two validation scenarios.

2. **Is the city metaphor adapted to visualize OO variability implementations?** Based on the previously obtained requirements, we adapt the city metaphor to our identification problem and propose *VariCity*, a 3D visualization using the city metaphor to exhibit zones of interest, being zones of high density of variability implementations (Section 6.2).

3. **To what extent does the *VariCity* approach help the comprehension of the implemented variability?** To evaluate to what extent *VariCity* fulfills the identified requirements, we provide two evaluations. First, we unfold the two validation scenarios presented in Section 6.1 to validate whether *VariCity*'s capabilities help in understanding the implemented variability (Section 6.3). Then, we report on a controlled experiment with 49 students aiming to evaluate whether *VariCity* performs better than state-of-the-practice approaches to discover a codebase (Section 6.4).

6.1 Requirements

As the essence of software visualization consists of creating an image of software by means of visual objects that represent structure and/or behavior [Knight and Munro, 2000], we believe it is well suited to enable perception of variability implementations with a closer fit to the user mental model. Furthermore, the difficulty of discovering a codebase increases with its complexity, we believe that such visualization should be able to meet the constraints of an onboarding process. Onboarding is a case of program comprehension in which a new developer joins a project or a company [Berlin, 1993; Sim and Holt, 1998]. Contrary to the usual information seeking in program comprehension (*i.e.*, information pull), onboarding is more based on information push [Yates et al., 2020] and is harder when little is known about the system [Steinmacher et al., 2014]. In onboarding, it has also been shown that newcomers look for major patterns [Yates et al., 2020], such as the ones used in variability implementations. Finally, to avoid frustration by newcomers being onboarded [Begel and Simon, 2008], the capability to configure and make up adapted visualization for an expert is also crucial.

In this context, we structure our requirement analysis around software comprehension scenarios for visualization within an onboarding process. We are then supposed to target two types of users:

- **newcomers** in the project, skilled but with no real knowledge about the code (this role can be generalized to anyone attempting to comprehend some software with little or no prior knowledge);

- **experts** in the project, with knowledge of the code and its architecture, but with no explicit vision of the variability implementations. With experts, once they have gained knowledge on the variability, they are likely to be more interested in its evolution [Wettel and Lanza, 2008b; Pfahler et al., 2020]. We consider that all evolution scenarios are out of the scope of this work, as we first need to provide a visualization for a single snapshot of a project.

Consequently, we focus on scenarios that engage the expert to comprehend the implemented variability while building a preconfigured visualization for newcomers.

We then propose two scenarios:

- *Scenario 1*: **The expert wants to facilitate the exploration of the codebase by giving a pre-configured visualization to the newcomer.** Through this scenario, the newcomer onboards on a large codebase of which he needs to have a global comprehension of the implemented variability (*e.g.*, understand a library or API that is going to be reused).
- *Scenario 2*: **The expert wants the newcomer to comprehend a subpart of the codebase for the newcomer to be able to reuse it.** Through this scenario, the newcomer onboards on a codebase in which they will be asked to add a new feature. They, therefore, have to understand in more detail the interactions between the classes implemented variability in this subpart.

Finally, Yates et al. [2020] analyze the different types of information transmitted from an expert to a newcomer during onboarding sessions. It results that newcomers find helpful when experts give coarse-grained information about complex zones (ranging from a group of classes to design patterns) of the codebase to them, so they can dig into them by themselves. According to these findings, we can say that a visualization for a newcomer should: (*i*) display the main elements allowing them to understand the codebase (design patterns, zones with complex variability implementations), (*ii*) be configurable by the expert to tailor it for newcomers, (*iii*) provide navigation and interaction capabilities to be adapted by a newcomer (filtering, zooming), (*iv*) scale on large codebases.

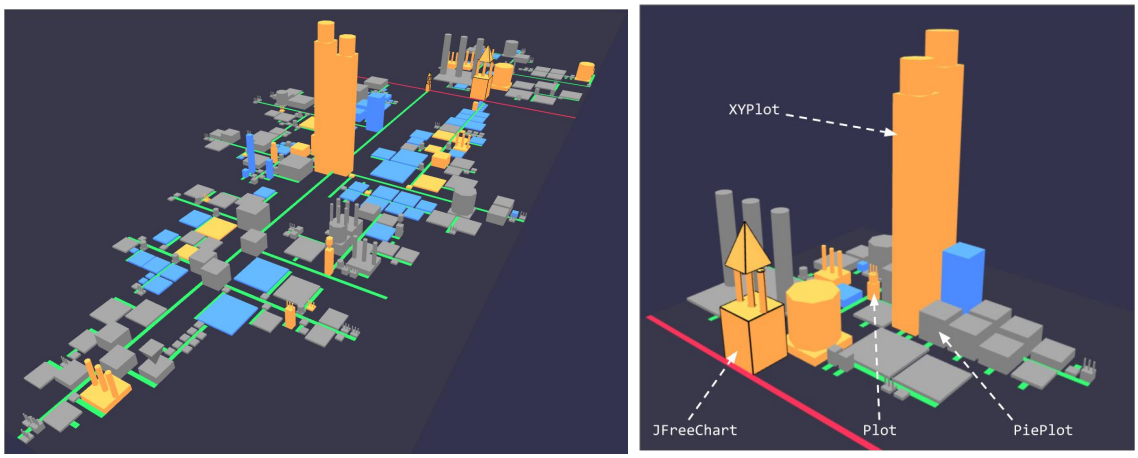
6.2 *VariCity*

As shown in Section 3.2, the city metaphor is a recognized way to visualize different properties of software systems. We hence adapt this visualization to the data we want to visualize and the defined scenarios.

6.2.1 Main principles

Buildings In CodeCity, classes are buildings and their size evolves according to metrics related to code quality which are inherent to the represented class, such as the cyclomatic complexity or the number of lines of code (LoC). For example, an important number of methods will lead to a tall building, catching the attention of the user on it. In *VariCity*, we aim to focus the user on classes making heavy use of variability implementations. Therefore, the dimensions of every building represent the class-based metrics related to variability, *i.e.*, the number of variants at method level – a tall building shows an important number of method variants, whereas a wide building shows an important number of constructor variants. For example, on Figure 6.1b, `XYPlot` appears as very tall as it has 77 method variants, whereas `Plot` is small as it has 6 method variants (*cf.* Figure 5.5). Moreover, buildings in color on the visualization (by default yellow for *vp*-s and blue for non *vp*-s) represent classes defined as *hotspots* as being part of dense zones of variability (*cf.* Section 5.1.3). The shape of the building is altered according to the design pattern(s) exhibited by the class[1] (*cf.*

[1] A design pattern often involves multiple classes, however only the *vp* of the design pattern has a special crown on it, not to overload the visualization.



(a) Visualization of JFreeChart 1.5.0 (b) Example from Figure 5.5 in *VariCity*

Figure 6.1: Sample views of *VariCity*

Table 6.1). Displaying differently classes being *hotspots* and/or exhibiting design patterns brings to the user insights on highly variable zones of the project, which they can then explore in more detail by using the different interactions provided by the visualization (spanning, zooming).

Streets Analogously, since the representation proposed by CodeCity groups classes belonging to the same package in a district to exhibit the packages containing the most complex classes, our goal is to group in the same neighborhood classes concentrating a high density of variability implementations.

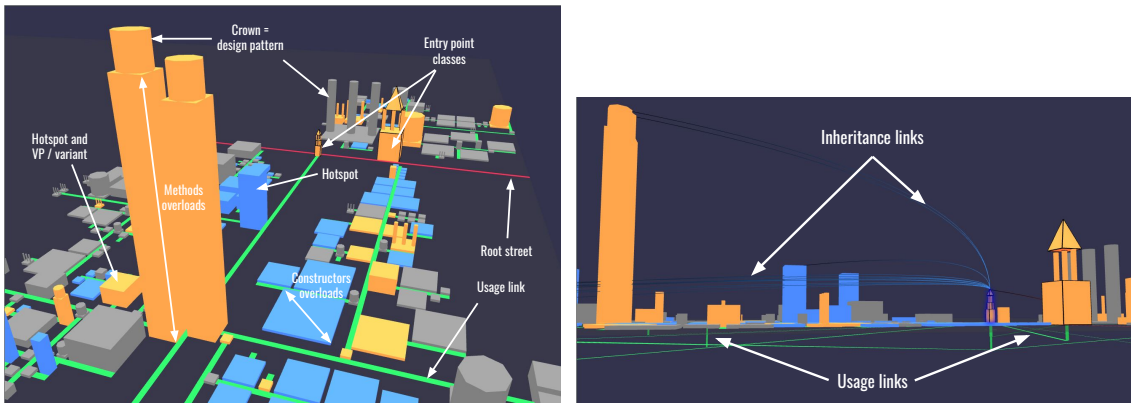
However, although the nested districts allow to efficiently represent the decomposition hierarchy of classes belonging to nested packages, it is not adapted to our notion of density of variability implementations which derives from usage relationships between classes (as a class can use and/or be used by multiple other classes). We thus rely on the visualization proposed by Evo-Streets [Steinbrückner and Lewerentz, 2013], which uses streets to decompose a hierarchy instead of boxes. In the original Evo-Streets layout, streets represent subsystems, with orthogonal branching streets representing their subsystems. The buildings on a street represent the modules belonging to this system. We adapt the visualization with buildings on streets being classes, and streets departing from a building (instead of another street) to represent a usage relationship between this class and every other class whose building is on the street. As we consider inheritance links as less important for variability, they are represented as aerial links between buildings, being only displayed when hovering over a building. This enables the user to see the inheritance information if needed, while the hotspot coloring and streets for usage bring the most important information first.

A summary of the visual properties is presented in Table 6.1 and illustrated in Figure 6.2. The view reveals the three different types of density detailed in Chapter 5. Classes with an important individual density, resulting from an important number of methods or constructors overloads (Definition 5.3), are more noticeable due to the important height or width of their buildings. Groups of classes exhibiting collective density, being close in terms of distance in usage relationships (Definition 5.4), have their buildings close to each other on the visualization, as if they were neighbors. Finally, hotspot classes exhibiting both density types (Definition 5.5) are colored, emphasizing

Table 6.1: Visual properties and their default color

Representation in *VariCity*	**Signification**
Buildings	
Yellow color	Variation point that is part of a hotspot
Blue color	Non-*vp* class that is part of a hotspot
Gray color	Class that is not part of a hotspot
Pyramide crown	Entry point class
Dome crown	Strategy pattern
Chimneys crown	Factory pattern
Inverted Pyramide crown	Template pattern
Sphere crown	Decorator pattern
Streets	
Plan (red)	Street aggregating entry point classes
Plan / Underground (green)	Usage relationship
Aerial (blue)	Inheritance relationship

classes that are parts of dense zones but initially less visible on the visualization by being less individually dense or linked through underground streets.



(a) Elements displayed by default

(b) Inheritance links and underground usage links appear when hovering a building

Figure 6.2: Visual properties of *VariCity*

Adaptable cities While the view should allow to quickly spot dense zones of variability implementations, a lot of information of different nature needs to be displayed: classes, links between them, design patterns. However, on a large project, providing a first view with all classes (and their usage / inheritance relationships) displayed would bring too much information. There is thus a need to focus the visualization around known points of interest of the system. The idea is there-

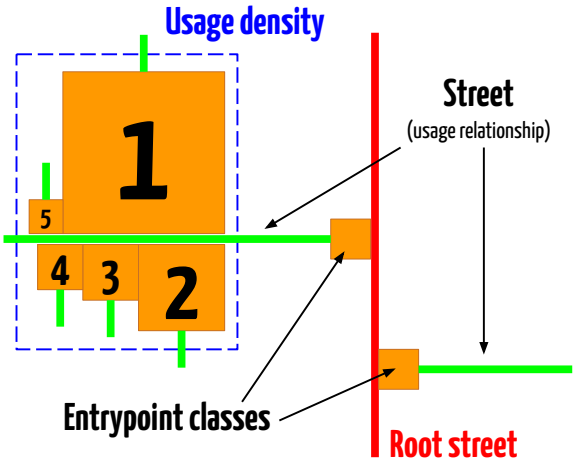


Figure 6.3: Depiction of the layout algorithm

fore to allow the expert to create a city in line with the most important elements for them and to give a first simplified vision of the city which does not show all the relationships between classes. The visualization algorithm thus relies on a certain number of inputs that focus the view (*cf.* Section 6.2.2). From this first visualization, it will also be possible to gradually adapt the city, among other things, by adding or removing relationships and classes (*cf.* Section 6.2.3).

6.2.2 From buildings and streets to a city

The goal of *VariCity* is to display the main elements allowing one to understand the variability implementations related to a given important class in the system (*cf.* Section 6.1). To do so, analogously to the previously proposed *symfinder-2*, *VariCity* relies on three mandatory inputs (Sections 5.1.2 and 5.1.3). The first one defines *entry point* classes, which represent important classes for the comprehension of the system (*e.g.*, endpoint of an API that could be automatically inferred, or complex classes of the system given by the expert). The second input is the *usage orientation*, which can be `IN` and/or `OUT`. An orientation `IN` means that the classes displayed will be the classes *using* the defined entry points (*i.e.*, having it as an attribute or method parameter). On the opposite, an orientation `OUT` implies that the classes displayed will be the classes *being used* by the defined entry points (*i.e.*, being an attribute or method parameter of the entry point). Depending on the objectives of the onboarding scenario envisaged by the expert, they might show either how the entry point uses or is used by other classes. More detailed examples are given in Section 6.2.3. Finally, setting the orientation to `IN/OUT` displays classes using or being used by the entry points. The third input is the *usage level*, which is an integer value. With a usage level of n, all classes distant from an entry point by n usage relationships will be displayed. For example, a visualization set up with an entry point, usage orientation `OUT` and usage level of 2 will display the entry point, the classes being used by the entry point, and the classes used by these classes. Being able to adapt this value is important as depending on the complexity or the layered architecture of a system, a given level of usage might be adapted to it but shows too many classes on another one.

Figure 6.3 depicts how the city is built. The root (first) street, in red, aggregates all the entry points. Then, starting from them, classes using (or being used by) them up to the usage level set are displayed. A street, in green, is initiated from an entry point, and for each class related to it, a building is placed on the border of the street. In order to exhibit density between classes, we need to place as close as possible buildings linked by a usage relationship to the same class. Following this principle, we place the buildings by descending order of width on both sides of the street, minimizing the total length of the street to keep the buildings as close as possible.

Our placing algorithm can lead to long straight streets if a class uses many others. Work presenting techniques to prevent this behavior and keep cities compact exist. A widely used technique is the use of Treemaps [Shneiderman, 1992; Bruls et al., 2000; Scheibel et al., 2020; Faccin Vernier et al., 2018; Tua et al., 2021; Kratt et al., 2011]. For example, Evo-Streets [Steinbrückner and Lewerentz, 2013] uses a packing-based layout to group neighboring buildings in rectangular districts along the street. However, such strategies are inapplicable in the case of *VariCity* since streets issue from buildings, therefore every building needs to be on its street to keep the shape of the city. Another approach consists in folding streets by adding turns [Kratt et al., 2011]. Still, having long streets is valuable in the case of *VariCity* as it allows to quickly visualize classes concentrating many usage relationships.

It is also likely to happen that a class is linked through a usage relationship to multiple visualized classes. In that case, these additional usage relationships are represented as green underground streets and appear only when hovering the class, as well as the inheritance relationships not to overload the visualization[2]. An example of visualization after generation is presented in Figure 6.1a.

6.2.3 Configuring the view to adapt the city

The configuration of *VariCity* is done in two steps. The first step concerns the adaptation of the mandatory inputs required to build the visualization (*i.e.*, entry point classes, usage level, and usage orientation), which are preconfigured by the expert. Based on their knowledge, the expert determines which classes are relevant enough to be entry point classes. The orientation will be set depending on what they expect the newcomer to understand from the system: if they want the newcomer to reuse a part of the implementation, they will likely choose the `IN` orientation as it will show which classes already use the entry point so that the newcomer can see how the class is already used. On the opposite, if they want the newcomer to add a new feature, they will more likely choose `OUT` so that the newcomer sees which classes are used by the entry points to know which classes they may need to reuse. Finally, choosing `IN/OUT` gives an overview of both aspects. Determining the usage level can only be done empirically. A level too low might hide important information for the comprehension of the variability, and a level too high might display too much information. Such characteristics are dependent on every codebase. For example, the visualization of JFreeChart presented in Figure 6.1a has `JFreeChart` and `Plot` as entry points, a usage level of 4, and a usage orientation `OUT`. The expert can also choose not to display classes that they consider irrelevant by putting them in a *blacklist*.

The second step represents options allowing to adapt the visualization, such as visual settings (colors of the visual elements, padding between the buildings) that may improve the readability of the visualization. Metrics for the height and width of the buildings can also be adapted. This

[2] In both the standalone and integrated versions of *VariCity*, when hovering over, class names are also displayed in a sidebar for the same reason.

parameter may be useful for the expert that has a particularly deep understanding of the system. For example, if the method level variability of classes is due to constructor overloads, it might be useful to use this metric for the height instead of the width of the buildings.

Although all these parameters for both steps have default values set by the expert, they can also be adapted by the newcomer while exploring the visualization in a sidebar to maximize their autonomy. We will illustrate in Section 6.3 how different values for the inputs in the first step impact the structure of the visualization by detailing the two scenarios presented in Section 6.1.

6.2.4 Implementation

VariCity is deployed as a standalone web-based visualization, and developed in TypeScript with the Babylon.js[3] 3D library. It relies on *symfinder-2* for the automatic identification of the symmetries and, relying on them, of the potential *vp*-s and variants as depicted in Figure 5.5. Information is structured by class and used by *VariCity* to build the visualization (*cf.* Section 6.2.2), relying on the settings provided in a configuration file. The whole application is deployed with Webpack and requires only a web browser to be viewed. As for *symfinder-2*, *VariCity* is deployed using Docker to ease the reuse and reproducibility of the visualizations presented in this chapter. The source code of *VariCity* is available online [Mortara et al., 2021b].

6.3 Scenario-based evaluation

In Chapter 5, the *symfinder-2* toolchain, which detects potential *vp*-s with variants, was applied on ten popular open-source and variability-rich Java systems, being applications, framework, or libraries, with different characteristics (size, variation points, explicit API provided). We chose to select the same systems to test the results of *VariCity*. In Table 6.2 are listed the systems and their *VariCity* configuration to facilitate the exploration or deepening of a particular area, as shown by our scenarios. The entry points have been determined by exploring the codebases and documentations, and selecting important classes accordingly. The values for usage level and orientation were determined empirically to provide a visualization showing interesting zones. By tailoring the inputs for these systems, we show that our approach is applicable to systems of various sizes and structures. The generated cities for all systems are available in the reproduction package [Mortara et al., 2021b]. Entry point classes being preconfigured, the user just needs to adapt the values for the usage level and orientation.

In this section, we evaluate whether *VariCity* answers to the needs expressed in Section 6.1, relying on the scenarios presented in the same section. We chose the Apache NetBeans IDE with its 5 MLoC[4] for Scenario 1 to illustrate the exploration of a large codebase. We chose the JFreeChart charting library for Scenario 2 to illustrate comprehension for reuse, as this scenario requires a finer-grained knowledge of the codebase, and we already detailed parts of its variability implementations in [Tërnava et al., 2019]. A video walkthrough of the scenarios is available on *VariCity*'s website at https://deathstar3.github.io/varicity-demo/.

[3] https://www.babylonjs.com/
[4] https://netbeans.apache.org/

Table 6.2: Subject systems

System	Entry point(s)	Usage level	Usage orientation
Java AWT	`java.awt.Shape`	3	IN/OUT
Apache CXF	`org.apache.cxf.endpoint.Endpoint`	6	OUT
JUnit	`org.junit.Assert` `org.junit.rules.TestRule`	3	IN/OUT
Maven	`org.apache.maven.Maven` `org.apache.maven.execution.MavenSession`	7	OUT
JFreeChart	`org.jfree.chart.JFreeChart` `org.jfree.chart.plot.Plot`	2	OUT

ArgoUML	`org.argouml.cognitive.Designer`
	`org.argouml.uml.ui.UMLModelElementListModel2`
	`org.argouml.uml.diagram.ui.FigNodeModelElement`
		2	IN/OUT

Cucumber	`io.cucumber.plugin.event.Event` `io.cucumber.java.StepDefinitionAnnotation`	11	IN/OUT
Logbook	`org.zalando.logbook.Logbook` `org.zalando.logbook.Sink`	4	OUT
Riptide	`org.zalando.riptide.Http`	6	IN/OUT
NetBeans	`org.netbeans.api.java.platform.JavaPlatform`	5	IN/OUT

6.3.1 Scenario 1: exploration of the codebase

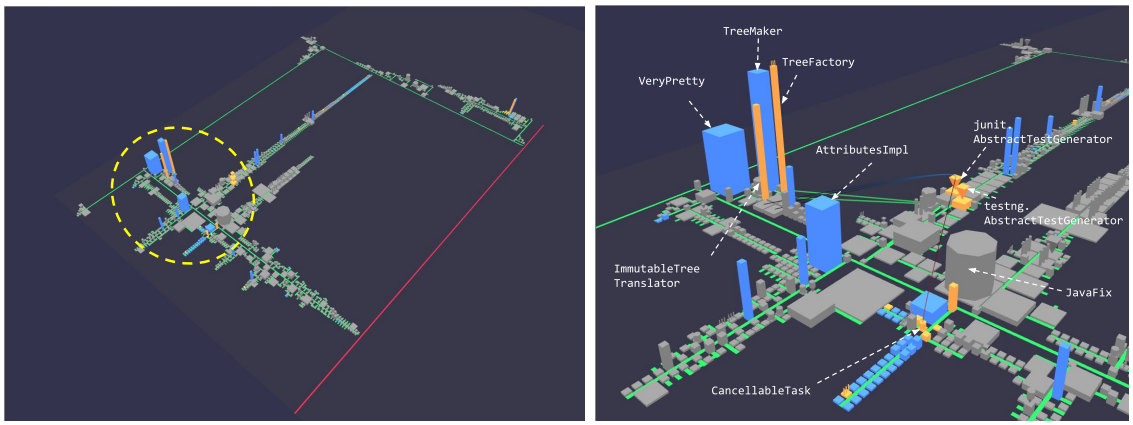
6.3.1.1 Objectives

With this scenario, we want to evaluate how *VariCity* and its configuration capabilities can help to distinguish zones of high density of variability in a codebase, which are manifested by buildings of particular height or width (*i.e.*, important number of method level variability implementations), in color (*i.e.*, part of dense zones of variability implementations), or with a crown (*i.e.*, presence of a design pattern).

6.3.1.2 Unfolding the scenario

The newcomer onboards on the NetBeans IDE codebase and needs to use the JavaPlatform API, which configures the version and location of Java to be used when building and running a project[5]. To better understand the operation of the API, the newcomer thus needs to have a global vision of the structure of the usage and inheritance relationships between the classes. To this effect, the expert configures the visualization to use the endpoint of the API, namely `JavaPlatform`[6], as the entry point of the visualization. Both classes using and being used by `JavaPlatform` on 5 levels (usage level 5, orientation `IN` and `OUT`) are shown to have a first overview of the classes being closely related to the endpoint of the API. The obtained visualization is shown in Figure 6.4a. A neighborhood of tall and colored buildings (circled in yellow) detaches from the other buildings in the city, showing to the user zones with classes heavily using variability implementation techniques. By zooming and spanning the visualization, the user

[5] https://bits.netbeans.org/12.2/javadoc/org-netbeans-modules-java-platform/overview-summary.html
[6] `org.netbeans.api.java.platform.JavaPlatform`



(a) NetBeans, usage level 5, orientation IN/OUT, `JavaPlatform` as entry point.

(b) Zoom on a hotspot zone

Figure 6.4: Visualization of the package `java` of NetBeans 12.2

can focus on this precise part of the city (Figure 6.4b)[7]. The different implemented design patterns are distinguishable due to the special shape of their buildings (*e.g.*, `JavaFix` is a Strategy, `ngtest.AbstractTestGenerator` and `junit.AbstractTestGenerator` are Templates). The two last classes are not only design patterns but also hotspots, giving a strong intuition about the relevance of the potential identified *vp*. In fact, these classes allow generating test code for two different unit test libraries, JUnit[8] and TestNG[9] and are variants of the `CancellableTask` interface[10].

6.3.2 Scenario 2: comprehension of a subpart of the codebase for reuse

6.3.2.1 Objectives

With this scenario, we want to evaluate how the customization of the view by the newcomer can allow them to tailor the visualization to obtain fine-grained details about the codebase.

6.3.2.2 Unfolding the scenario

The newcomer onboards on JFreeChart, a Java library allowing to draw different types of charts, and is asked to implement a new type of chart in the library. Contrary to the first scenario, the newcomer aims at adding a new feature to the codebase, thus they need a more fine-grained understanding of it, as, for example, the classes used by the other charts that they might also need to use. The expert thus configures the visualization to use as entry points `JFreeChart`[11], being the endpoint of the library used by the users to create plots and `Plot`[12], the superclass of all

[7] The names and arrows have been manually added on the figure. The name of the class corresponding to a building appears in a sidebar of the visualization when hovering over the building. Packages, when unnecessary, have been omitted for readability.

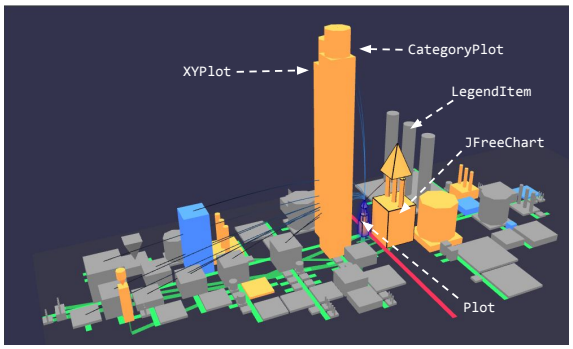
[8] https://junit.org/junit5/

[9] https://testng.org/doc/

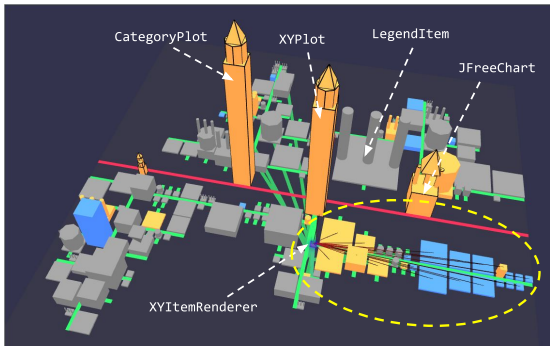
[10] See here and here.

[11] `org.jfree.chart.JFreeChart`

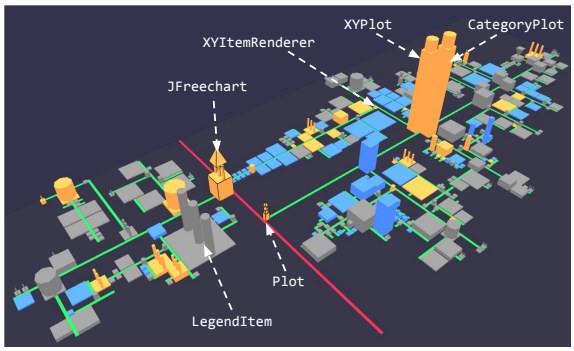
[12] `org.jfree.chart.plot.Plot`



(a) JFreeChart, usage level 2, orientation OUT, `JFreeChart` and `Plot` as entry points. Displaying links of `Plot` reveals that `XYPlot` and `CategoryPlot` are subclasses.



(b) Figure 6.5a after adding `XYPlot` and `CategoryPlot` as entry points.



(c) Figure 6.5a after increasing the usage level to 4.

Figure 6.5: Scenario 2

classes implementing a different type of chart. As the goal is to display which classes are used by these two entry points, the usage orientation is set to `OUT` and the usage level to 2. The obtained visualization is shown in Figure 6.5a.

We notice that the colored buildings, which represent classes being part of dense zones of variability implementations, do not align with the most variable classes. For example, `LegendItem` is a factory and, due to its large base, exhibits an important number of constructor variants. However, although this class is internally dense in variability, it is a utility object which not related to any other *vp*, and for this reason, is not characterized as a hotspot.

By hovering over `Plot`, the newcomer can see the different displayed subclasses of the class (*i.e.*, the variants of the *vp* `Plot`). To add another type of chart in the library, they will need to implement a new variant of this *vp* and needs thus to have an overview of the classes used by these subclasses. To do so, the user adds the two most variable ones (`XYPlot`[13] and `CategoryPlot`[14]) as entry points (Figure 6.5b). The shape of the city changes to display the usages related to each entry point in separated neighborhoods, allowing to better visualize if *(i)* a particular entry point is the starting point of a dense zone of variability implementations, and *(ii)* a class is related (to a certain degree) to two entry points with underground streets. On Figure 6.5b, an important number of classes making heavy use of variability implementations is visible, and

[13] `org.jfree.chart.plot.XYPlot`
[14] `org.jfree.chart.plot.CategoryPlot`

are directly used by `XYItemRenderer`[15], itself related to both `XYPlot` and classes related to `CategoryPlot`. Given these characteristics, the newcomer may need to reuse it to implement his feature and thus can add it as another entry point to visualize its usage if needed.

To visualize the classes used by `XYPlot` and `CategoryPlot`, the newcomer could also have chosen to increase the usage level on the visualization given by the expert, as shown on Figure 6.5c. However, an important number of classes and relationships not related to the newcomer's interest would appear, hampering the comprehension. The newcomer could also have chosen other class variants of `Plot` to add as entry points. However, most likely the classes that will be added to the visualization are not dense in variability, thus less interesting for the scenario.

6.3.3 Summary

Through these two scenarios, *VariCity* enables the newcomer to see variability implementations in an unknown codebase from a high-level perspective, and also to dig into them to have a more precise understanding.

6.4 Controlled experiment

While the two unfolded scenarios demonstrated *VariCity*'s capabilities, the relevance of the approach is conditioned by its capacity to perform better than state-of-the-practice approaches to comprehend the implemented variability. In collaboration with Anne-Marie Dery-Pinna, we thus design a controlled experiment with real users to observe how using *VariCity* impacts the time needed to complete variability comprehension tasks and its difficulty.

6.4.1 Experimental design

Wettel et al. [2011, 2010] designed an empirical evaluation of CodeCity aiming to evaluate whether the view helped the identification of quality-critical zones in an OO codebase and extracted from the literature a wish list of requirements for their experiment. As we conduct a similar evaluation, we therefore rely on this list to design our experiment and detail its design in the remainder of this section. Table 6.5 summarizes, for each of these requirements, how our design fulfills them or not.

6.4.1.1 Research Questions

With this experiment, we aim to answer the following questions:

RQ_1**:** Does the use of *VariCity* increase the correctness of the solutions to variability identification tasks, compared to state-of-the-practice tools?

RQ_2**:** Does the use of *VariCity* reduce the time needed to solve variability identification tasks, compared to state-of-the-practice tools?

RQ_3**:** Is *VariCity* regarded as easy to use to solve variability identification tasks compared to state-of-the-practice tools?

[15] `org.jfree.chart.renderer.xy.XYItemRenderer`

6.4.1.2 Subjects

This experiment was realized as part of a reverse-engineering graduate course at the Polytech Nice Sophia engineering school. The population is made of 49 students in the last year of Master's in Computer Science, specialized in Software Architecture. While it is known that having students as subjects for controlled experiments does not always give reliable results as they might not be representative of the target population [Feitelson, 2015], we think it is appropriate in our case for two reasons. First, being in the last year of Master's in Computer Science, an important majority of them will integrate an industrial company in the next few months and need to onboard on an unknown codebase, thus exactly the usage scenario of *VariCity*. Second, they are advanced developers in Java, thus mastering object-oriented programming concepts. They also followed multiple courses prior to the experiment related to the comprehension of complex code architectures, thus preventing a bias on their knowledge of these aspects.

6.4.1.3 Purpose and variables

Through the three defined research questions, the goal of this experiment has been set towards evaluating whether *VariCity* allows subjects to better identify patterns involved in complex zones of variability implementations, *i.e.*, the *effectiveness* of the approach. Additionally, we aim to assess whether *VariCity* reduces the time needed for subjects to answer the tasks and their perceived difficulty compared to state-of-the-practice tools, *i.e.*, the *efficiency* of the approach. Such goals being identical in the empirical evaluation of CodeCity by Wettel et al. [2010], we therefore share identical dependent and independent variables. We detail them hereafter.

Dependent variables Our first dependent variable concerns the tool used to solve the task. In order to mitigate the effect of this variable, we must compare our approach with a state-of-the-practice approach used to achieve an identical goal, that is, understanding the variability implemented in OO software systems. While comparing *VariCity* to *symfinder-2* would allow evaluating the potential gain brought by the city metaphor, we cannot consider it a state-of-the-art approach as it is not used regularly by the subjects. Therefore, the comparison would be irrelevant as between two approaches that subjects do not master. Since, to the extent of our knowledge, no similar and commonly used approach exists, we build a baseline ourselves relying on tools that developers would actually use to navigate and understand the code artifacts. IDEs are widely used tools for program comprehension [Minelli et al., 2015]. While a majority of our subject students use the IntelliJ IDEA IDE, we did not impose any particular IDE as (*i*) the given tasks (listed in Section 6.4.1.6) can be answered using only basic features supported by a large majority of IDEs (as finding usages, code navigation), thus we do not expect them to use advanced features that would be specific to a specific IDE (*e.g.*, tracing, dynamic analysis) and (*ii*) we limit the bias regarding the mastering of the IDE as every subject can use the one they master the most.

VariCity however uses data and metrics that are previously computed by the identification backend (*cf.* Figure 5.5). Since our goal is to compare the gain of *VariCity* compared to the use of an IDE, we should provide the subjects with all information given by *VariCity* that cannot be determined using the IDE's features. The inheritance and usage relationships between classes being standard navigation features, it is thus possible to infer the variants at class level, and to determine hotspot classes and design patterns, whose definitions are given to the subjects. It results that the only missing information is the number of overloads of methods and constructors. We collected

Table 6.3: Structure of the given CSV containing data on the classes

Class name	**Method variants**	**Constructor variants**
`org.jfree.chart.ChartPanel`	6	5
`org.jfree.chart.ChartRenderingInfo`	0	2
`org.jfree.chart.JFreeChart`	17	3
`org.jfree.chart.LegendItem`	0	10
...	...	...

Table 6.4: Statistics on the object system used for the experiment, JFreeChart

# LoCs	# classes	# *vp*-s			# variants		
		class level	method level	total	class level	method level	total
94,384	990	259	667	926	275	1,648	1,923

this information in a CSV file to complete the baseline. The structure of the file is given in Table 6.3. As for the IDE, no restriction has been imposed on a particular spreadsheet to manipulate the CSV file for similar reasons.

Our second dependent variable regards the studied object system and its architecture. While a large system or with multiple layers of abstraction would require too much time to be understood in such an experiment, a too small system would on the opposite not require an approach as *VariCity* to help its understanding and therefore not allow evaluating its potential gain. For these reasons, we selected JFreeChart 1.5.0 as an object system, whose characteristics are presented in Table 6.4. Not only does its 95k LoC make it a system of medium size, being a charting library that we studied to evaluate *symfinder* and *symfinder-2*, we know that both the domain and the implementation are accessible to the subject students. For similar reasons, we selected ArgoUML as a test project on which the subjects can familiarize themselves before the actual experiment on JFreeChart (*cf.* Section 6.4.1.7). Given the size of the population, we decided not to experiment on a second object system as the groups for each treatment would have been too small (around 12 subjects) to draw any conclusion (*cf.* Section 6.4.1.5).

Independent variables Our independent variables regard the *correctness* of the solution given for a task and the *time* to complete the task, which respectively allow measuring the *effectiveness* and the *efficiency* of our approach.

6.4.1.4 Controlled variables

In their experiment, Wettel et al. [2010] benefited from a large panel of subjects from academia (ranging from bachelor students to professors) and industry. Therefore, subjects in this panel exhibited large differences in terms of background of experience, potentially having an influence on their capacity to complete the tasks. In our case, all our subjects are all students having studied similar topics. While 37 out of the 49 students are apprentices and thus have between 6 and 24 months of professional experience, we consider the impact that this difference may have on the student's capacity to solve the tasks as negligible. We therefore do not consider these variables for our experiment.

6.4.1.5 Treatments

We split the overall population randomly into two groups:

***VariCity* (24 subjects)** The first group is given a link to a GitHub repository containing:

- the result of JFreeChart's and ArgoUML's analysis by *symfinder-2*, used as input by *VariCity*;
- a *VariCity* configuration file to display the views.

The *VariCity* image is distributed as a Docker image hosted on the Docker Hub, thus requiring no installation on the students' computers.

IDE + CSV (25 subjects) The second group is given a link to a ZIP file containing:

- the source code of JFreeChart and ArgoUML;
- the CSV file containing the metrics for the classes[16].

6.4.1.6 Tasks

The tasks derive from the onboarding scenarios presented above (Section 6.1). The subjects have 1h10 to complete all the tasks.

Part 1 (estimated duration: 35 mins)

1. **Task.** Identify 2 variants at class level for each of the following variation points:
   - `org.jfree.chart.plot.Plot`
   - `org.jfree.chart.title.TextTitle`

   Goal. Inheritance is a heavily-used mechanism to implement OO variability (Section 2.2.2), therefore their identification and exploration is crucial to identify variability in this context. With this task, we want to evaluate whether the used tools allow the exploration of such mechanisms.

2. **Task.** How many classes are linked with a usage relationship to the each of the following classes? Give 3 examples.
   - `org.jfree.chart.plot.CategoryPlot`
   - `org.jfree.chart.title.CompositeTitle`

   Goal. The collective density of variability implementations is characterized by a cluster of *vp*-s linked through usage relationships (Definition 5.4). With this task, we want to evaluate whether the used tools allow an overview of these mechanisms by distinguishing the classes linked by usage relationships to a given one.

[16] Although configuring the view might add or remove classes on the visualization, the given tasks do not require this action. Therefore, the given data is strictly identical between both groups.

3. **Task.** Complete the following sentences:

   - Classes (1) and (2) have an important number (≥ 5) of subclasses (*i.e.*, are variation points with an important number of variants at class level).
   - Classes (3) and (4) have an important number (≥ 10) of overloaded methods and constructors (*i.e.*, are variation points with an important number of variants at method level).

   Goal. While usage relationships induce collective density, the individual density of variability implementations is characterized by the presence of a *vp* with an important number of variants at class or method level (Definition 5.3). With this task, we want to evaluate whether the used tools allow an overview of where such mechanisms are concentrated in the codebase.

4. **Task.** Identify the 3 classes with highest individual density higher to the threshold $v = 20$.
   Goal. For a given density threshold, the number of classes characterized as dense can remain important depending on the dimensions and architecture of the studied system. It is therefore important to be able to focus on the most dense classes. With this task, we want to evaluate whether the used tools allow this.

Part 2 (estimated duration: 35 mins)

5. **Task.** Give 2 examples of each of the following design patterns:

   - Strategy pattern;
   - Factory pattern.

   Goal. Those two design patterns being used to implement OO variability (Section 2.2.2), we want to evaluate with this task whether the used tools allow their identification.

6. **Task.** What is the distance between the `org.jfree.chart.JFreeChart` and `org.jfree.chart.title.DateTitle` classes?
   Goal. As the density of OO variability implementations relies on usage relationships between classes, we aim to evaluate whether state-of-the-practice tools allow a user to compute the distance between two given classes.

7. **Task.** Identify 3 hotspots for an individual density threshold of $v = 20$ and a collective density threshold of $d = 5$.
   Goal. The most dense zones concentrating variability implementations are characterized as *vp*-s being simultaneously individually and collectively dense (*cf.* Definition 5.5) and it is therefore important to identify them. With this task, we want to evaluate whether the used tools allow identifying them.

8. **Task.** Identify the classes that according to you implement each of the following features, and specify if they are hotspots for $v = 20$ and $d = 5$:

   - “draw a chart” feature;
   - “title of the chart” feature.

Goal. Variability identification activities being often conducted to help the comprehension of this variability [Krüger et al., 2019b], we aim with this task to evaluate whether the exploration of the system with the given tools allowed the subjects to determine the classes involved in the implementation of a feature.

For each task, the subjects are also asked:

- to **input the start and end time** of the task. With this information, we aim to evaluate whether the time spent completing a task differs when using *VariCity* or the IDE.
- to **rate the difficulty of the task** on a scale from 1 to 4. With this information, we aim to evaluate whether the perceived difficulty for a task differs when using *VariCity* or the IDE.
- to **list the actions they accomplished** (*e.g.*, navigating the inheritance in the IDE or zooming on the visualization). We plan to use this data to better understand how the tools were used to solve the task and better understand the causes of the results obtained with the two previous pieces of information.

6.4.1.7 Operation protocol

Before the experiment A lecture of one hour and a half introducing the main concepts related to variability, *symfinder-2* and *VariCity* was given to the students on January 12, 2022.

The day of the experiment (February 23, 2022) On the day of the experiment, a short lecture was given to all the students, presenting definitions and examples of the various terminologies used in the tasks (about 45 mins). Then, after splitting, each group benefited from a short session to fill a preliminary questionnaire aiming to gather personal information, setup their environment and familiarize with the tools they will manipulate (about 40 mins):

- subjects in the *VariCity* group were introduced to the visualization's features, had to open the visualization for the test project (ArgoUML), and had to answer a few questions ensuring that they can interpret the visualization accurately;
- subjects in the IDE + CSV group were introduced to the data they are given, had to open the test project's codebase in their IDE, and had to answer a few questions ensuring that they can interpret the data accurately.

After this session, the subjects had 1h10 to answer the tasks, having for only help a cheat sheet of the definitions detailed in the lecture. Finally, the students filled out a questionnaire aiming to gather their feedback on the experiment.

6.4.2 Results

We detail hereafter the results obtained from our experiment. Due to the limited time allocated for the experiment, not all subjects could finish all the tasks. 17/24 subjects in the *VariCity* group and 14/25 subjects in the IDE group gave at least a partial answer (*i.e.*, filled at least one of the answer fields of the task) to all the tasks. Figure 6.6 presents, for each task, the percentage of subjects having given at least a partial answer. We observe a drop in the percentage of answers at tasks 6 and 7 for the IDE and *VariCity* groups respectively for two reasons. First, we ordered

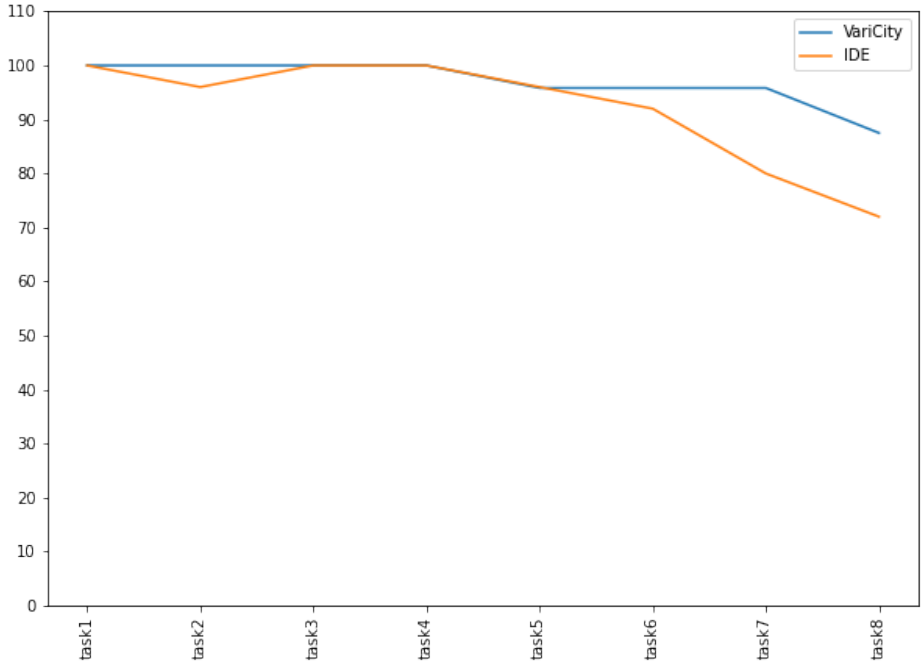


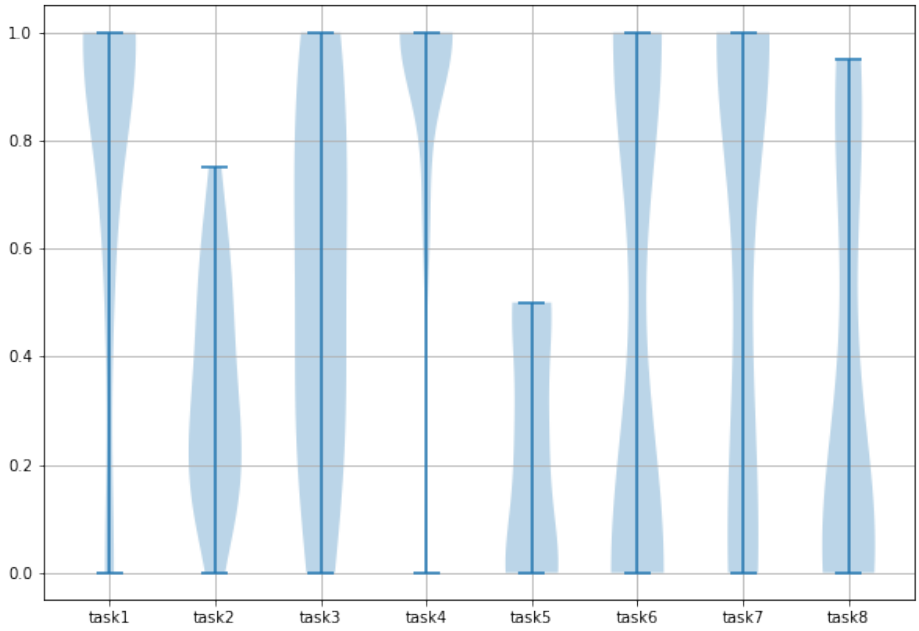
Figure 6.6: Percentage of subjects having given at least a partial answer for each task

the tasks in increasing order of difficulty as we perceived it. Then, we made the choice not to impose a time limit for each task as we were unsure about the average time that the subjects would take depending on the tool(s) they use and feared a too low answers rate. Therefore some subjects took more time than they should have on the first tasks and happened to be late at the end of the experiment. In the following results, we considered answers that are at least partial.

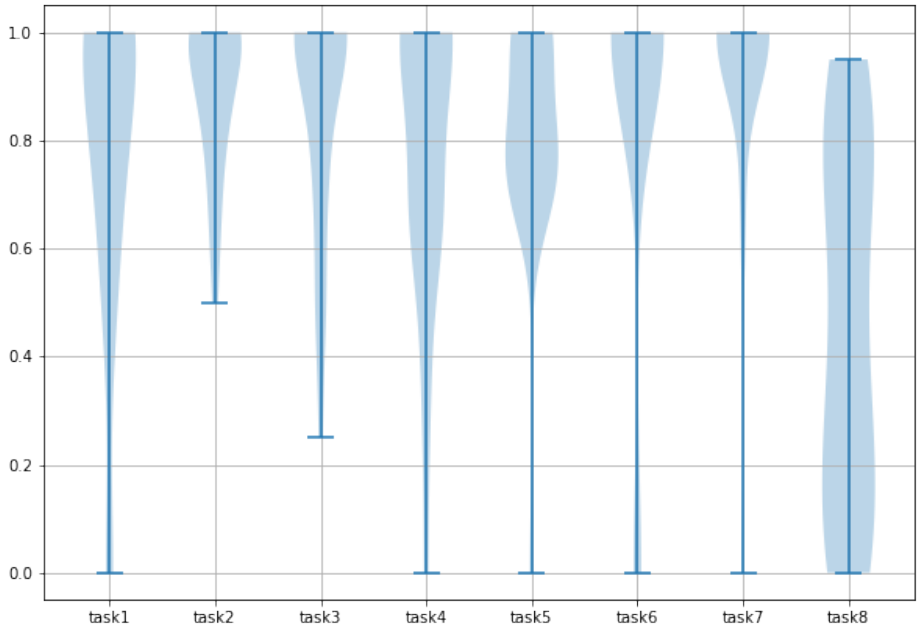
6.4.2.1 Answer to RQ_1

Figure 6.7 presents the correctness of the answers given by the two groups on each task. We calculate the correctness of an answer to a task by calculating the percentage of correct answer elements. For example, when 2 names of classes are expected, the correctness is 0% if no class is in the expected set of classes, 50% for 1 class, and 100% for both classes in the set.

We notice that subjects using *VariCity* globally gave more accurate answers to the tasks than subjects using the IDE + CSV combination. For a number of tasks, we believe this is due to the layout of the visualization and the choice of visual axes. For example, using the dimensions of the buildings to represent their methods and constructors overloads, and aerial links for subclasses exhibits individual density helped subjects to complete task 3 (finding *vp*-s with an important number of class or method level variants). Similarly, choosing to use streets to represent usage relationships helps their understanding, exhibiting collective density, and helped subjects to complete tasks 2 (finding classes linked through usage relationships to a given one) and 6 (finding the distance between two given classes). Answering tasks 5 (finding design patterns) and 7 (identi-



(a) IDE + CSV group



(b) *VariCity* group

Figure 6.7: Correctness of the answers given by the two groups.

fying hotspots) was facilitated by the automatic computation of dense zones and design patterns provided by the *symfinder-2* approach used by *VariCity* and their visualization using crowns on buildings and colors respectively.

We notice however that using *VariCity* does not improve the correctness of the answers to all the tasks compared to the IDE + CSV combination. Although the ranges of obtained correctness values are similar between both groups, subjects with *VariCity* gave more correct answers on average on task 8. This task was divided in two parts, (*i*) the identification of classes implementing a given feature, and (*ii*) indicating whether they were hotspots or not. To answer the first part, subjects from both groups mainly relied on the names of the classes that are given by both *VariCity* and the IDE, leading to similarly accurate answers. However, as it has been previously detailed for task 7, identifying hotspots has to be done manually when using the IDE and the CSV files, leading to less accurate answers for this part of the question in this subgroup.

Comparable correctness results are obtained on task 1 where the goal was to give two variants at class level (*i.e.*, two subclasses) for two *vp*-s. Both *VariCity* and the IDE allow searching a class by its name and easy access to the subclasses of a given class (by hovering its building in *VariCity*, and a button in the sidebar of the IDE to navigate the inheritance hierarchy). Analyzing the actions achieved by the subjects in both groups confirms that they heavily relied on these features, and a majority of the subjects in the IDE + CSV group did not use the CSV file.

Finally, subjects from the IDE + CSV group perform better on task 4, which consisted of the three most individually dense classes. For the IDE + CSV group, obtaining the answer to this task consisted in finding the classes maximizing the sum of their method and constructor overloads, and was achieved by most students. On the opposite, *VariCity* displays those two pieces of information using the height and width of the buildings, making it less intuitive to identify the buildings maximizing both aspects. As a result, some subjects from this group not only indicated tall but also wide buildings that were maximizing their constructors' overloads but not the total method variants.

It results that subjects using *VariCity* globally answer the tasks more correctly than subjects using the IDE + CSV combination, due to the organization of the information provided by the city metaphor as well as its computation of other information such as the presence of a design pattern or a hotspot. Given these encouraging results and the little rate of wrong answers given, we choose not to exclude them from the analyses answering RQ_2 and RQ_3.

6.4.2.2 Answer to $\mathbf{RQ_2}$

Figure 6.8 presents the average time spent on each task for both groups. It results that no tool performs better on all the tasks.

VariCity performs better on tasks 2, 4, 5, 6 and 7 as it directly exhibits on the visualization the presence of hotspots and design patterns, while IDE users need to identify them manually. The structure of the city based on usage relationships also helped the subjects to identify the distance and usage relationships between two classes while IDE users needed to explore the code.

Concerning task 1, subjects with the IDE performed better. Completing this task is equivalent to finding the subclasses of a given class, an action that the subjects are used to accomplish regularly with their IDE. This is confirmed by the fact that 23/25 subjects in the IDE group used the IDE only to complete the task. Therefore, subjects in this group were on average faster than

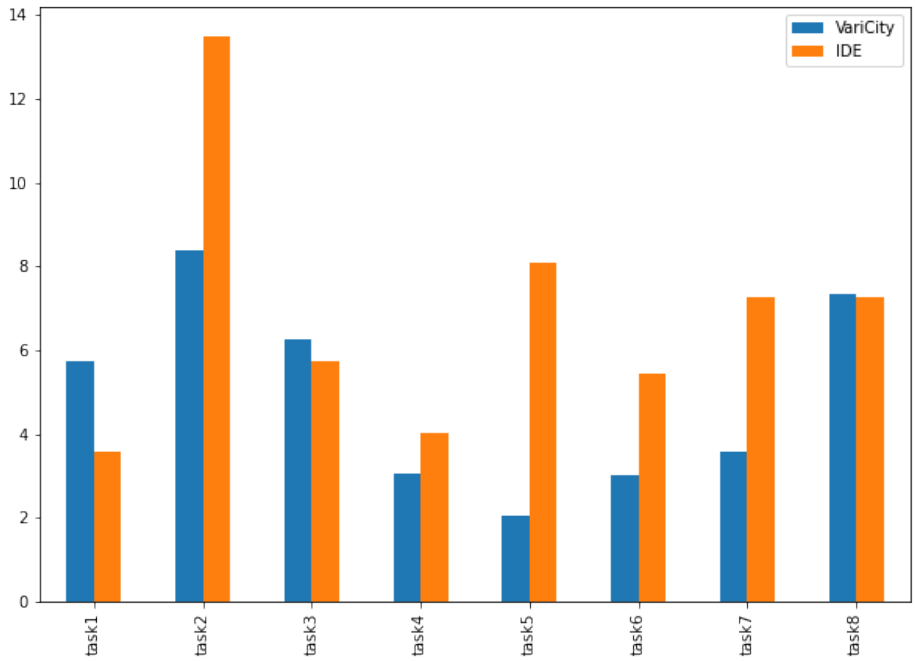


Figure 6.8: Average completion time (in minutes) for each task when using *VariCity* or the IDE

subjects in the *VariCity* group that had to find the information in a visualization they were less familiar with.

Finally, on tasks 3 and 8, the performances of both approaches appear equivalent. To complete task 3, finding the classes with an important number of variants at method level consisted in looking at the corresponding column in the CSV for the IDE group, and in looking at high and/or wide buildings in the visualization, suggesting that all subjects had a clear idea of where to find this information. Regarding the number of variants at class level, while subjects in the IDE group heavily relied on class diagrams reverse-engineered with the IDE, subjects in the *VariCity* group mainly hovered over random classes until finding the information, suggesting that finding class level variants in *VariCity* is not intuitive. We would expect task 8 to take less time using *VariCity*. In practice, it results that the average time spent is equivalent as multiple subjects from the IDE group did not give the hotspot information. However, correlating this result with the fact that 50% of subjects indicated the maximum perceived difficulty (*cf.* Figure 6.9) is a strong indication that this part of the task was too difficult for the IDE group.

It results that the subjects in the IDE group performed better on single tasks they are familiar with (*e.g.*, finding a class, navigating inheritance). On the opposite, identifying complex zones concentrating variability implementations is more efficient in *VariCity* (*e.g.*, design patterns, hotspots). While determining the individual density of variability appears to be equally time consuming thank to the CSV file, determining the collective density of variability and hotspots is more efficient by the organization of the *VariCity* view.

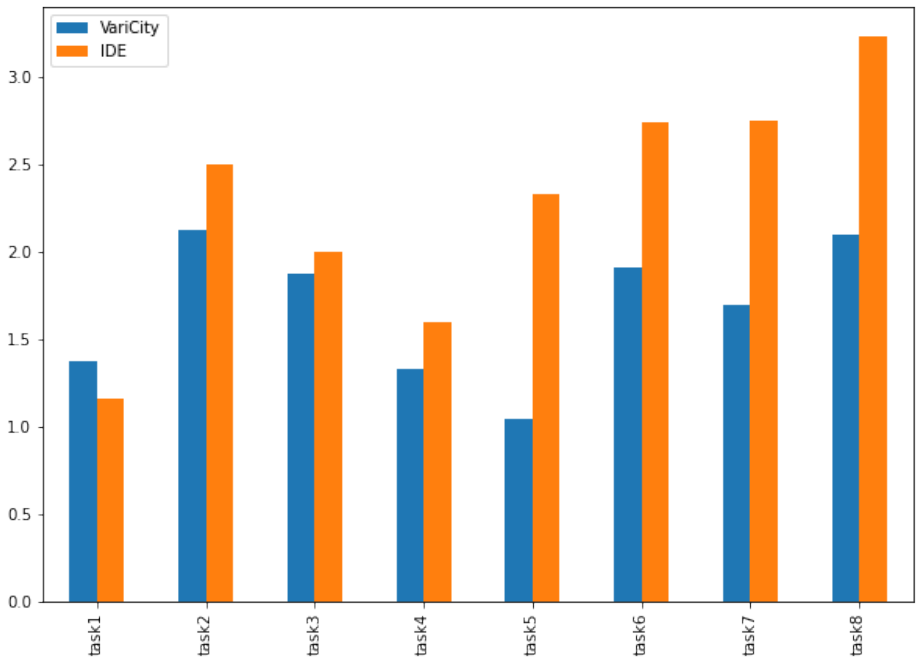


Figure 6.9: Average difficulty (on a scale from 0 to 4) for each task when using *VariCity* or the IDE

6.4.2.3 Answer to RQ_3

Figure 6.9 presents the average difficulty as perceived by the subjects on each task for both groups. While a correlation can be seen for both groups between the perceived difficulty and the time spent for each task (Figure 6.8), we can observe that on average, subjects in the *VariCity* group find it easier (or at least as difficult) to complete all tasks except for the second. This finding coincides with the observation concerning the time spent on this task in the two groups in the previous section and can be explained by the fact that subjects are used to navigating inheritance relationships in the IDE. We notice that the difference in perceived difficulty between both groups is more considerable on tasks 5 to 8 which are focused on identifying design patterns, hotspots, and distance between classes. This is also coherent with the results obtained regarding the time spent completing these tasks. Tasks 5, 6 and 7 took considerably more time for subjects using the IDE compared to subjects using *VariCity*. This is not the case for task 8 as, as explained in the previous section, an important number of subjects did not give information about the presence of a hotspot and only partially completed the task.

It results that subjects in the *VariCity* group find it easier to complete the tasks compared to subjects in the IDE group, especially for tasks related to the identification of zones concentrating variability implementations using complex structures, such as design patterns or hotspots that implement variability at both class and method levels.

6.4.2.4 Summary

By providing a visual representation of the system's classes and exhibiting metrics on their variability, *VariCity* helped in globally reducing both the completion time and the perceived difficulty of the tasks. A deeper analysis of the actions performed by subjects from both groups suggests some improvements for *VariCity*. Some information is more easily accessible with the visualization, such as the distance in usage between classes or the density of variability implementations. Other actions were facilitated with the IDE, such as obtaining a reverse-engineered class diagram to navigate the inheritance relationships. Feedback from some subjects in the *VariCity* group reveals that although the view allowed to quickly spot important zones of the system (*"The visualization allows us to easily notice the features present in the system. Classes that are less important can be ignored to focus on the ones that have more variability."*), their comprehension was limited by not having the actual source code (*"We stay really abstract by visualizing the code with VariCity, a Java IDE would allow us for example to have access to comments that can help comprehension."*, *"It is hard to understand how the system works only with the buildings."*). This feedback suggests that although *VariCity* helps in guiding the exploration of a system, having access to the source code remains of prime importance to have a deep understanding of it.

6.5 Threats to validity and Limitations

Without an empirical assessment, the main threat of our scenario-based evaluation concerns the scenarios that we designed by ourselves. Nevertheless, we relied on both empirical work on onboarding with real experts and newcomers [Yates et al., 2020] and challenges related to the comprehension of variability concepts [Acher et al., 2017], giving us good confidence in the relevance of the scenarios.

Another threat related to this evaluation arises by the fact that both authors and developers of *VariCity* determined empirically the inputs (entry points, usage level and orientation) for each scenario, based on their knowledge of the systems and of *VariCity*'s capabilities. Still, even by having a coarse-grain understanding compared to a real expert, the obtained visualizations already exhibit satisfying results. We expect real experts to be able to determine appropriate inputs in real settings, as it has been the case with Daniel Le Berre during the controlled experiment on *symfinder* applied to Sat4j (Chapter 4).

Concerning the structure of the visualization, the placement of the buildings on a street only relies on the width of the buildings to compact them in the street. This implies that the variability represented by the height of the buildings is not taken into account. Even if this dimension is largely visible on the visualization, this calls for an adaptation of the placement algorithm to take into account both dimensions while placing the buildings.

Finally, due to organizational constraints, our controlled experiment could only be conducted on a single object system and with a panel of subjects of similar experiences, preventing mitigation of biases related to the specificities of the object system and the background and experience of the subjects. Nevertheless, our panel of subjects correspond to the target population of *VariCity*, and we estimate that the dimensions and characteristics of the chosen library (JFreeChart) were reasonable to allow an evaluation in a reasonable time.

6.6 Related Work

Works on the city metaphor in software visualizations were studied in Section 6.2.1. In this section, we discuss work related to visualization for variability management and to assist onboarding activities.

6.6.1 Metaphors to visualize software properties

Numerous visualizations for software rely on diverse metaphors [Chotisarn et al., 2020], in particular to display static properties about software [Caserta and Zendra, 2010], from 2D visualizations relying on treemaps [Balzer et al., 2005] to 3D representations of object-oriented elements. For example, Software Landscapes [Balzer et al., 2004] uses geometric shapes in a 3D space to represent classes, methods, and attributes, as well as the relationships between them (inheritance, method calls, attribute used by methods). Package nesting is represented with spheres, classes are circular discs, and methods and attributes are cuboids on these discs. These elements are connected between each other to represent a relationship. A recent approach by [Hoff et al., 2022] represents a software system as a solar system, picturing the different granularities of layers in the system's architecture with concepts ranging from planets to floors of buildings in cities grouped by continents.

6.6.2 Visualization in the Software Product Line field

A recent mapping study has shown that visualizations in the SPL domain mainly target feature models, using tree or graph representations [Lopez-Herrejon et al., 2018]. These visualizations are mainly used to facilitate the configuration process over features. To visualize variability at the code level, some approaches use colors [Kästner et al., 2008] or bar diagrams [Duszynski and Becker, 2012], while some others focus on feature traces [Andam et al., 2017] or feature interactions between features and code [Greevy et al., 2005; Bergel et al., 2021]. None of them focus on object-oriented techniques as variability implementations.

In *VariCity*, we reused the symmetry-based detection part of *symfinder* [Tërnava et al., 2019; Mortara et al., 2021c], but this tool also provides a graph-based visualization in which each class level *vp* and variant is represented as a circle node that points out the used implementation technique, with size and shades of nodes indicating some occurrences of symmetries. These nodes are linked with both inheritance and usage relationships being different kinds of edges, forming a set of disconnected graphs. While this visualization allows showing some dense zones of variability and has filtering capabilities, it has only been used for the validation of the capabilities of *symfinder* in identifying potential *vp*-s and variant. It is not adapted for comprehending variability as in our considered scenarios, especially in large-scale systems in which the resulting visualization is not usable (approx. 4k nodes for NetBeans).

6.6.3 Visual tools to assist onboarding

Some visualizations have been especially proposed for onboarding activities. Isopleth [Hibschman et al., 2019] represents call relationships in front-end JavaScript implementations in the form of a call graph, which is interactive and can be edited to see the impact in real-time on the page. Other tools integrate information from the organization to help information seeking during development

activities, such as Tesseract [Sarma et al., 2009] which visualizes the relationships between technical information from a codebase and related social data (*e.g.*, developers, communication, code, and bugs). Finally, recent studies on onboarding in SPLs [Azanza et al., 2021] explore concept maps [Novak and Cañas, 2006] to structure information about the SPL. However, this approach, as many others evoked in Section 6.6.2 relies on a feature model and documentation, which does not apply in our case.

6.6.4 Controlled experiments

Controlled experiments in software engineering context have been conducted to assess the gain brought by tooled approaches [Ko et al., 2015]. Naturally, being especially designed for software comprehension, visualization approaches are often evaluated this way [Müller et al., 2014; Fittkau et al., 2015]. Additionally to the empirical evaluation conducted on CodeCity [Wettel et al., 2010, 2011], other city-based approaches mentioned in this chapter benefited from such an evaluation, such as ExplorViz [Fittkau et al., 2015] or DynaCity [Dashuber and Philippsen, 2022b]. Controlled experiments are also conducted to evaluate the relevance of evolutions of these metaphors, for example by adapting them to more immersive environments such as virtual reality [Rüdel et al., 2018] for which variants of Evo-Streets [Steinbeck et al., 2019] and CodeCity [Moreno-Lumbreras et al., 2022] have been designed.

In the context of variable systems, multiple approaches assisting the maintenance of feature models [Bagheri and Gasevic, 2011] and the understanding of the implemented variability have been evaluated with controlled experiments. For example, VarXplorer [Soares et al., 2018] identifies feature interactions and displays them as a graph, and has been compared to the state-of-the-art closest approach, the Varviz Eclipse plugin [Meinicke et al., 2018]. Focusing on recovering information on the implemented variability, Pérez et al. [2020] conduct a controlled experiment comparing manual and automated feature location approaches on their performance, productivity and satisfaction.

6.7 Conclusion

In this chapter, we tackled the following questions:

1. **What are the requirements for a visualization approach to comprehend OO variability implementations?** Relying on onboarding scenarios where a newcomer has to comprehend the implemented variability, we identified that such a visualization should (*i*) display the main elements allowing them to understand the codebase (design patterns, zones with complex variability implementations), (*ii*) be configurable by the expert to tailor it for newcomers, (*iii*) provide navigation and interaction capabilities to be adapted by a newcomer (filtering, zooming), and (*iv*) scale on large codebases.

2. **Is the city metaphor adapted to visualize OO variability implementations?** We could implement these requirements by adapting the city metaphor in the *VariCity* approach, a 3D visualization proposing adapted and configurable views that exhibit zones of high density of variability implementations. The density relies on previous work on automated detection of symmetries in the variability implementation mechanisms. Metrics on their occurrences together with information on inheritance and usage relationships are exploited to build a city with, notably, classes as buildings and streets as usage relationships.

3. **To what extent does the *VariCity* approach help the comprehension of the implemented variability?** We detailed two onboarding scenarios showing how the different capabilities of the visualization can help to spot critical variability-related zones of a codebase and obtain fine-grained information about them. A controlled experiment with real users showed that *VariCity* participated in reducing the completion time of the given variability comprehension tasks and eased their completion.

By providing a 3D view and adapting the city metaphor, the interactions provided by *VariCity* show considerable improvements to comprehend the implemented variability compared to the graph view proposed by *symfinder-2*, consequently answering Challenge A2 (*"Making the identified variability implementations comprehensible"*). However, the conducted evaluations exhibited limitations that restrain the practical use of the approach. First, multiple subjects that were using *VariCity* during the controlled experiment pointed out that not having the source code was limiting their comprehension of the variability, and that they could have performed better with if they had the source code side by side with the visualization. Therefore, it results that in order to bring a real gain, *VariCity* should be used together with the IDE, and not as an alternative. Second, understanding the implemented variability is often a step to reach another goal. For example, it is known that variability implementations add complexity to the codebase [Galster et al., 2017] and consequently hamper the quality of the system [Wolfart et al., 2021]. Ensuring the stability of a variable system thus does not only require identifying the implemented variability, but more particularly the quality-critical implementations. As a consequence, we extend *VariCity* to incorporate quality metrics and allow a user to observe simultaneously dense zones of variability implementations and metrics on their quality.

Table 6.5: Elements from the experimental design responding to requirements extracted from Wettel et al. [2010]'s wish list.

Requirement	**Experimental design element**
Fulfilled requirements	
Avoid comparing using a technique against not using it. Provide the same data to all participants.	As we validate the visualization approach, we give a CSV document opened with a spreadsheet containing structured information on the classes of the system visualized with the given settings using *VariCity* by the other group (Sections 6.4.1.3 and 6.4.1.5).
Provide a not-so-short tutorial of the experimental tool to the participants. Use the tutorial to cover both the research behind the approach and the implementation.	To complement the 1h30 lecture given two weeks prior to the experiment, a short tutorial introducing definitions and demonstrating the tool has been given before the experiment (Section 6.4.1.7).
Find a set of relevant tasks. Include tasks on which the expected result is not always to the advantage of the tool being evaluated.	The tasks are inspired by the onboarding scenarios in the first evaluation of *VariCity*, and some of them are expected to be more easily completed using the IDE and the CSV document (Section 6.4.1.6)
Choose real object systems that are relevant for the tasks.	JFreeChart has been chosen for its medium size and because the domain and the implementation are accessible to the subject students (Section 6.4.1.3)
Provide all the details needed to make the experiment replicable.	The questionnaires, slides and answers given by the students are available online at https://deathstar3.github.io/varicity-demo/
Report results on individual tasks.	Additionally to the answers given, subjects were asked for each task to provide the start and end time, an estimation of their perceived difficulty and the list of the actions they accomplished to solve the task (Section 6.4.1.6).
Non-fulfilled requirements	
Include more than one subject system in the experimental design.	Having a second subject system would have led to four treatments of 12 people each and would have prevented drawing any relevant conclusion.
Involve participants from industry. Take into account the possible wide range of experience level of the participants.	This requirement could not fulfilled due to organizational constraints. Although apprentice students have more professional experience, the difference of professional experience with the other students is not important enough to conclude on whether the performance of subjects differs *w.r.t.* this parameter (Section 6.4.1.2).
Avoid, whenever possible, to give the tutorial right before the test.	Although we gave a long lecture introducing *VariCity* and the associated concepts about one month before the experiment, we could not give the more detailed tutorial before the day of the experiment for organizational constraints (Section 6.4.1.7)
Limit the amount of time allowed for solving each task.	While the overall set of tasks should be completed in 1h10, we did not limit the time for each task to prevent fast but less qualitative answers (Section 6.4.1.6).

CHAPTER 7

Comprehending the quality of the implemented variability

This chapter shares material with the SPLC 2022 papers *"Customizable Visualization of Quality Metrics for Object-Oriented Variability Implementations"* [Mortara et al., 2022a] and *"IDE-assisted visualization of indebted OO variability implementations"* [Mortara et al., 2022b].

In the previous chapter, the *VariCity* approach has been proposed to visualize dense zones of variability implementations. It completes the *symfinder-2* identification method and allows better understanding of which zones of the codebase concentrate variability implementations and using which mechanisms, thus providing an answer to Challenge A2.

As discussed in Section 3.3, variability implementations add complexity in a system [Galster et al., 2017] and hamper its capacity to evolve [Favre, 1996; Kröher et al., 2022], eventually leading to technical debt [Avgeriou et al., 2016; Li et al., 2015]. By relying on the traditional OO mechanisms, OO variability implementations are intertwined in the implementation and hard to comprehend (Section 2.2.2), thus leading potentially to technical debt. Monitoring quality being crucial for the maintenance and evolution of such systems [Martini and Bosch, 2015], there is a need for a solution allowing to better identify and understand it (Challenge A3).

In this chapter, we tackle Challenge A3 with the following questions:

1. **How to identify indebted zones of variability implementations?** Identifying indebted zones of variability implementations implies first establishing relevant quality metrics for its identification, then determining how to reveal such zones to a user.

   (a) **How to measure technical debt related to OO variability implementations?** Relying on previous work on variability debt by Wolfart et al. [2021], we defined variability debt in the context of OO codebases in Section 3.3. As quality metrics have been recognized as useful for determining technical debt at the code level, we thus determine appropriate ones to measure this particular type of technical debt (Section 7.1).

   (b) **How to reveal technical debt related to OO variability implementations?** Approaches to identify either OO variability implementations or technical debt already exist and rely on visualization. However, simultaneous use of both techniques is cumbersome (Section 7.2). We therefore propose *VariMetrics*, an extension of *VariCity*

to support software quality metrics and reveal critical zones concentrating variability implementations (Section 7.3).

2. **Does *VariMetrics* allow visualizing indebted zones of variability implementations?** To validate the visualization approach, we proceed to a quantitative evaluation on 7 subject systems and report on views we designed for each system (Section 7.4.1).

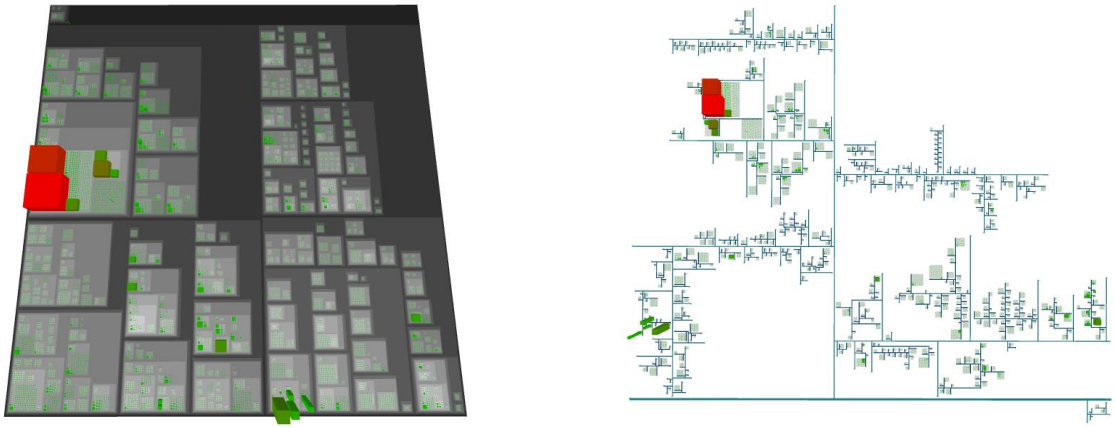
3. **Are the revealed indebted zones of variability implementations relevant?** To validate whether the visible quality-critical zones of variability implementations are relevant, we proceed to a qualitative evaluation on one of the studied systems, JFreeChart, and apply maintenance actions aiming to correct the identified debt in identified classes, and report on the evolution of the visualization and the quality metrics (Section 7.4.2).

7.1 Determining relevant quality metrics for OO variability debt

In Section 3.3, we studied the work of Wolfart et al. [2021] and the catalog of ten types of variability debt they introduced. After deep analysis of the contribution, it resulted that three types of variability debt are applicable to the systems we target, namely *Code duplication*, *Lack of tests*, and *System-level structure quality issues* in the implementation. There is need now to determine a technique to measure them. While technical debt covers diverse aspects of the software and its development ecosystem [Kruchten et al., 2012], its identification at the implementation level is mainly done through code analysis (*e.g.*, by computing metrics [Li et al., 2015; Rasool and Arshad, 2015]). More particularly, in OO systems, this technical debt is often measured using OO software metrics [Kafura and Henry, 1981; Rosenberg and Hyatt, 1997; Fowler, 2018; McCabe, 1976; Campbell, 2018; Misra et al., 2018]. We therefore advocate that such OO quality metrics are suited to identify OO variability debt, and detail hereafter which ones can be used to identify these types of variability debt in our context.

A common metric to identify a lack of tests is the code coverage, which can be measured at different granularities (line, condition, ...). For our evaluation, we opted for a coverage metric that aggregates measures for different granularities. Similarly, code duplications are commonly identified at two levels of granularity: line or block. We advocate that blocks are more likely to represent duplicated code related to variability than a single line of code. Finally, structure quality issues in the codebase impact maintainability and evolution of the system. Even though code duplication and lack of tests impact maintainability and evolution of the system, the understanding of the implementation by the maintainers of the project is also an important aspect, and cognitive complexity [Campbell, 2018] appears to be relevant for this purpose [Peitek et al., 2021]. We thus choose as relevant metrics for our evaluations duplicated blocks, test coverage, and cognitive complexity.

Most often, standard tools for measuring software quality metrics also determine technical debt measures giving an estimation of the effort, as a duration, to fix the identified code smells [Avgeriou et al., 2020]. We did not use such measures in our evaluation for multiple reasons. First, by providing an aggregated duration, this measure is more helpful in estimating effort at the management level, but it does not describe the real causes of the debt. Then, some first empirical results seem to indicate a possible inaccuracy in the given values [Baldassarre et al., 2020], and exploiting such metrics may therefore require some knowledge of the system and its implementation, which we do not have for our subject systems.



(a) CodeCity view (b) Evo-Streets view

Figure 7.1: Views of GeoTools, using cyclomatic complexity as footprint, # LoC as height, and complexity as color. The two most visible classes are `gml311.DocumentRootImpl` and `gml311.Gml311PackageImpl`.

Now that we identified measures for the three types of variability debt that can be identified in our targeted OO systems, we need to find an approach allowing us to identify this variability debt in actual systems.

7.2 Possible approaches

7.2.1 Tools for OO technical debt identification

On one side, tooled approaches have been proposed to measure the quality of OO systems and automatically compute values for quality metrics, including the ones we identified as relevant to measure OO variability debt [Lenarduzzi et al., 2018]. Understanding these metrics is then often enabled through the use of visualization. For example, one of the most popular tools used to monitor the quality of software projects, SonarQube[1], proposes the SoftVis3D[2] plugin embeding the CodeCity [Wettel and Lanza, 2008a] and Evo-Streets [Steinbrückner and Lewerentz, 2010] visualizations, both relying on the city metaphor. Figure 7.1 illustrates the two visualizations on the GeoTools project[3], an open-source Java library for geospatial data management. Classes are represented as buildings and their width, height, and color are used to display the quality metrics, making discernible classes maximizing these metrics. Districts in CodeCity (Figure 7.1a) and streets in Evo-Streets (Figure 7.1b) represent the decomposition in packages. However, none of them allows displaying information on the system's variability.

7.2.2 The *VariCity* approach

On the other side, the *VariCity* approach proposed in Chapter 6 has been designed to reveal dense zones of variability implementations (*cf.* Section 6.2). An example of a generated *VariCity* visu-

[1] https://www.sonarqube.org/
[2] https://softvis3d.com/
[3] https://www.geotools.org

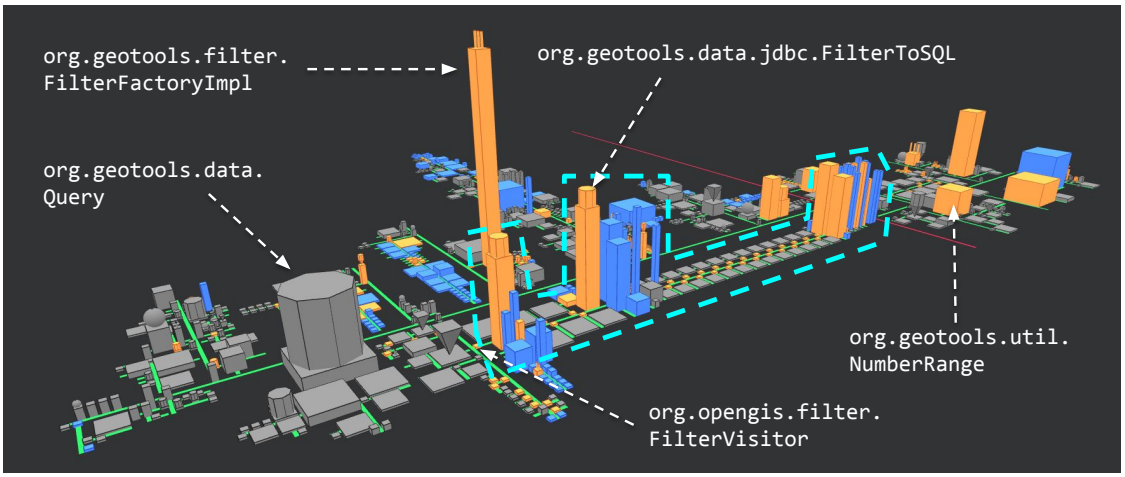


Figure 7.2: *VariCity* visualization of GeoTools.

alization is shown in Figure 7.2 and presents a visualization generated from GeoTools. Compared to the visualizations exhibited on Figure 7.1, the noticeable classes are different as discernible classes on a *VariCity* view are the ones concentrating variability implementations. For example, `FilterFactoryImpl` is shaped as a skyscraper due to an important number of method overloads (141). Its goal is to create filters allowing to select zones from a map[4]. The large strategy is `Query` (10 constructors), which uses filters to query information from a data source. On the opposite, `FilterVisitor` is not very variable in itself but uses all the implemented filters, in the blue dotted box, noticeable by being a long street. Coloring the hotspot classes not only emphasizes the filters having more variants, but also exhibits some isolated classes, for example `NumberRange`, which implements a numerical range of values. On the opposite, the two red classes exhibited in Figure 7.1 because of their too high cyclomatic complexity (`gml311.DocumentRootImpl` and `gml311.Gml311PackageImpl`) are not visible in Figure 7.2 as they are not part of zones concentrating variability implementations. *VariCity*, however, does not display information related to the software quality of the displayed classes.

While solutions exist to visualize either OO metrics (CodeCity, Evo-Streets) or OO implementations (*VariCity*), we could think about using both approaches simultaneously. However, navigating between *VariCity* and a metric-specific tool would be cumbersome as it would require manually finding and mapping information having heterogeneous representations, even between two views reusing the city metaphor (*e.g.*, *VariCity* and Evo-Streets) as they are shaped differently. Consequently, to the extent of our knowledge, no solution exists to visualize at the same time, for an OO system, its variability implementations, and quality metrics over them. This thus calls for a unified but customizable visualization and we propose to extend *VariCity* to incorporate quality metrics over a variability-centric visualization.

[4] https://docs.geotools.org/latest/userguide/library/main/filter.html

7.3 *VariMetrics*: exploring the quality of variability implementations

As shown in Section 7.2, although state-of-the-art approaches allow visualizing either the density of variability implementations (*e.g.*, with *VariCity*) or quality metrics (*e.g.*, with CodeCity [Wettel and Lanza, 2008a] or Evo-Streets [Steinbrückner and Lewerentz, 2013]), no existing approach allows the simultaneous representation of both aspects of OO software systems. Therefore we adapt *VariCity* to display information about quality in the city.

7.3.1 Choice of metrics

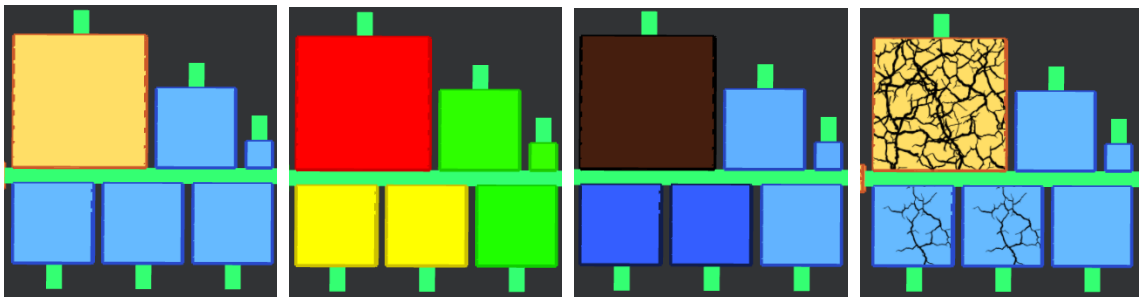
State-of-the-art proposes a plethora of quality metrics to measure several properties of a software system [Colakoglu et al., 2021], ranging from the architecture [Nuñez-Varela et al., 2017] to the source code level [McCabe, 1976; Stevanetic and Zdun, 2015]. Since no metric is relevant for all software systems due to the elusive definition of quality [Kitchenham and Pfleeger, 1996], software practitioners need to pick and combine different metrics to obtain a quality measure relevant for their use case. *VariMetrics* extends the configuration of *VariCity* so that experts can choose the quality metrics they want to display, and how to combine them, to tailor the visualization according to their needs. Consequently, although we chose for our evaluation three quality metrics being the number of duplicated blocks, the test coverage, and the cognitive complexity (*cf.* Section 7.1), *VariMetrics*'s configuration capabilities allow visualizing other metrics if the expert finds them relevant.

7.3.2 Coloring strategies

By default, *VariCity* displays in yellow *vp*-s being hotspots, in blue variants being hotspots, and in grey classes not being hotspots (Figure 7.3a). On their side CodeCity and Evo-Streets color the buildings to expose properties inherent to the classes [Wettel and Lanza, 2008c; Steinbrückner and Lewerentz, 2010]. We thus propose two coloring strategies for quality metrics: a coloration following a red-to-green sequence (Figure 7.3b), and a saturation keeping the original colors of the buildings and lightening or darkening them (Figure 7.3c). While *VariMetrics* should enable some combination of metrics, combining both coloring strategies leads to bivariate chromatic maps, which are known to be difficult to read [Wainer and Francolini, 1980]. On the opposite, applying textures on colors has shown to be an efficient way to display multiple software quality metrics [Holten et al., 2005]. We hence provide a crackled texture (Figure 7.3d) variably covering the building, thus enabling views simultaneously exhibiting two quality metrics.

7.3.3 Configuration

These three visual properties are configurable to be adapted to the metric they represent, as some quality metrics are symptoms of lower quality if they have a high value (*e.g.*, complexity) but other metrics with such values may instead indicate good quality (*e.g.*, test coverage). Analogously, not all projects have similar ranges of values for the same metric, and proposing a fixed range of values may not allow revealing a difference of quality in some projects, thus *VariMetrics* allows to specify these ranges. Figure 7.4 shows the *VariCity* view of Figure 7.2 in *VariMetrics* showing the cognitive complexity using the red-to-green color scale. Where the classes concentrating variability implementations revealed by *VariCity* (cf Section 7.2.2) remain visible independently of their quality (*e.g.*, `FilterFactoryImpl` or `NumberRange`), *VariMetrics*



(a) *VariCity* (b) Red-green (c) Saturation (d) Cracks

Figure 7.3: Visual properties used to display quality metrics compared to the original *VariCity* visualization.

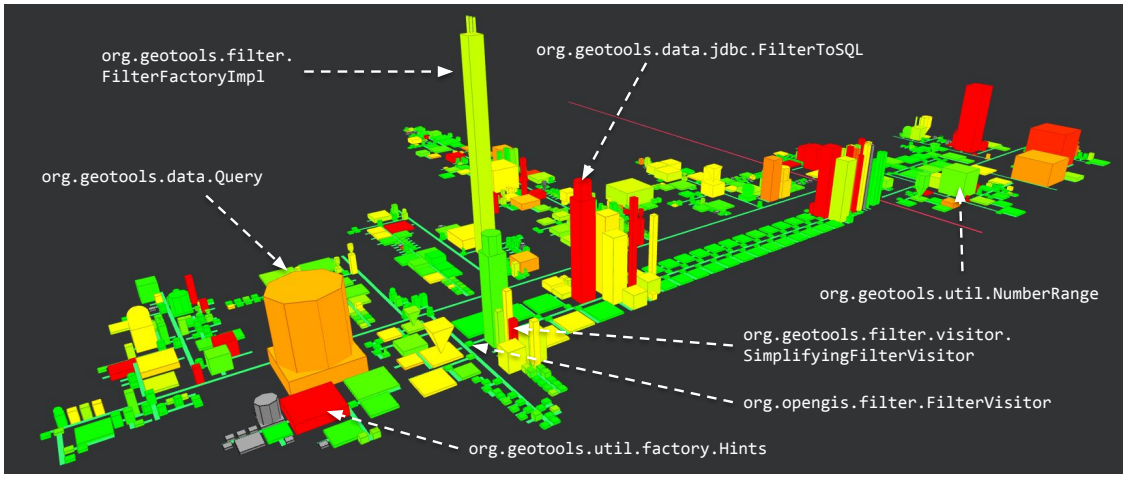


Figure 7.4: Figure 7.2 in *VariMetrics*. The view is configured to display the cognitive complexity using the red-to-green color scale.

also exposes quality-critical classes, being variable (*e.g.*, `Query` or `FilterToSQL`) or not (*e.g.*, `Hints` or `SimplifyingFilterVisitor`).

7.3.4 Implementation

The *symfinder* toolchain, used by *VariCity* to identify the variability implementations and compute the related variability-related metrics, has been extended to support fetching of the quality metrics and their mapping with the identified variability information. If a SonarCloud account exists for the system, metrics are fetched by using the SonarCloud Web API[5]. Otherwise, a SonarQube server is executed locally to extract the metrics while running the *symfinder* analysis. The *symfinder* configuration has been extended to specify wherever running a SonarQube instance is needed or not.

[5] https://sonarcloud.io/web_api

Table 7.1: Subject systems and their available metrics.

Project	Version	Java LoCs	# *vp*-s / variants	Available metrics		
				DB	COMP	COV
Azureus	5.7.6.0	633,248	10,105	A	S	✗
GeoTools	23.5	1,312,727	22,534	A	S	✗
JDK	17-10	2,434,983	71,489	S	S	✗
JFreeChart	1.5.0	94,203	2,849	S	S	S
JKube	1.7.0	40,952	795	A	S	S
OpenAPI Generator	5.4.0	88,172	768	S	S	S
Spring framework	5.2.13	662,579	12,622	A	S	✗

DB – duplicated blocks, COMP – cognitive complexity, COV – coverage
✗ – unavailable metric, A – available metric, S – significant metric (available and showing differences between classes)

7.4 Evaluation

The evaluation of *VariCity* presented in Section 6.3 validates its capacity to exhibit zones in the code concentrating mechanisms used in OO variability implementations. *VariMetrics* should therefore be able to reveal the subset of these classes having quality issues. To evaluate whether *VariMetrics* identifies variability implementations for which quality metrics are problematic, we apply our approach to multiple open-source systems. We select views with metrics combinations revealing the variability implementations that are shown by *VariCity* while being the most quality-critical (Section 7.4.1). We then validate the relevance of such classes by applying maintenance actions on these classes within one project, JFreeChart (Section 7.4.2), and show the impact on the view of the project.

7.4.1 Quantitative evaluation

Subject systems We used for this evaluation 7 variability-rich open-source Java systems of various sizes, depicted in Table 7.1. Five of them were chosen as their documentation clearly states they implement variability: **Azureus** (Vuze) is a BitTorrent client which supports multiple network communication protocols, **GeoTools** a library for geospatial data management providing multiple tools and filtering capabilities to manipulate maps, **JKube**, a Maven plugin to generate different types of container images, **OpenAPI Generator**, a library to create APIs for a plethora of programming languages, and the **Spring framework**, providing a Java-based support for components and services with many different plugins, on persistence management, validation, security, etc. We also picked the **Java Development Kit (JDK)** for its large size of ~2.5M LoC to evaluate the scalability of our approach. Finally, we also used **JFreeChart**, a charting library used as a subject system in the evaluation of *VariCity* (Section 6.3), as its size enables us to master the implemented variability at a fine granularity. Five projects are forks from their original repositories in the Corpus-2021 GitHub organization[6], designed by Irrazábal et al. [2021] to serve as a catalog of software projects to analyze their metrics. They provide a SonarCloud instance for these projects[7],

[6] https://github.com/Corpus-2021
[7] https://sonarcloud.io/organizations/corpus-2021/projects

Table 7.2: Number of noticeable classes due to their variability concentration, criticality, and both aspects for the given views on all subject systems.

View configuration				Noticeable classes *w.r.t.*		
Entry point classes	**Usage orientation**	**Usage level**	**Metrics (visual property)**	**variability**	**criticality**	**both**
Azureus						
`com.aelitis.azureus.core.AzureusCoreComponent`	`OUT`	4	COMP (red-green)	74	32	12
GeoTools						
`org.geotools.data.simple.SimpleFeatureSource` `org.geotools.map.MapContent`	`OUT`	4	COMP (red-green)	104	27	18
JDK						
`java.net.URI` `java.net.URL`	`IN`	1	COMP (red-green) DB (cracks)	84	17	13
JFreeChart						
`org.jfree.chart.JFreeChart` `org.jfree.chart.plot.Plot`	`OUT`	4	COV (red-green) DB (cracks)	35	31	10
JKube*						

`api.support.BaseGenerator`
`javaexec.JavaExecGenerator`
`api.Generator`
	`IN/OUT`	7	COV (red-green)
			COMP (cracks)
				28	115	14

OpenAPI Generator						
`org.openapitools.codegen.languages.OpenAPIGenerator`	`IN/OUT`	6	COV (red-green) COMP (cracks)	77	51	21
Spring framework**						
`parsing.BeanComponentDefinition` `support.AbstractBeanFactory`	`IN`	8	COMP (red-green)	57	13	6

* classes base package: `org.eclipse.jkube.generator`
** classes base package: `org.springframework.beans.factory`

allowing us to reuse these metrics for our study. Two others have also a SonarCloud instance and JFreeChart is the only one for which we had to use our prototyped setup with Sonarqube to obtain the quality metrics. Besides, the JFreeChart's build configuration was also adapted to be analyzed by a local SonarQube instance [Mortara et al., 2022].

Evaluation process We first generated for each project a visualization with *VariCity* following the same stages as in the *VariCity*'s evaluation (Section 6.3). After determining entry points by selecting important classes after exploring codebases and documentations, we experimented empirically with different combinations of usage level and usage orientation to obtain a visualization we consider relevant (*i.e.*, exhibiting classes detaching from others because they concentrate variability implementations). We finally identified manually on each view the classes that are the most visible for us (by being a hotspot or a design pattern, or due to their dimensions) to obtain a set of "noticeable classes *w.r.t.* variability". For example, for GeoTools (Figure 7.2), classes such as `FilterFactoryImpl`, `FilterToSQL`, `Query`, and `NumberRange` draw attention due to their size and/or the fact that they are hotspots, as opposed to `FilterVisitor`.

To determine a relevant *VariMetrics* view, we systematically applied all available metrics on each project and selected the ones being relevant to identifying OO variability debt (*cf.* Section 7.1). During this step, it happened that no building stood out for a metric (*i.e.*, no class exhibits variability debt), suggesting that the overall quality is decent *w.r.t.* this metric. On the opposite, if all classes appear as quality-critical, it may indicate that this metric has been neglected in quality

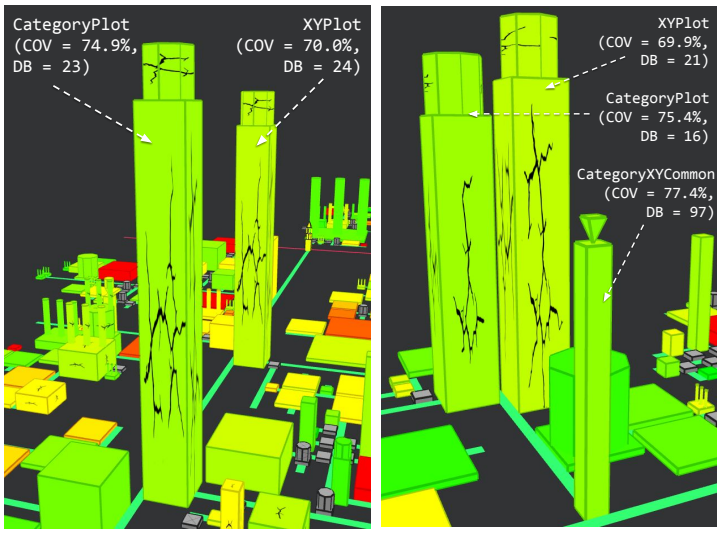
requirements for the project as a whole. We thus restrained in this evaluation the set of *significant* metrics relevant to identify OO variability debt to those showing some differences in quality between classes. Table 7.1 summarizes for each system the relevant metrics being available and significant. We then manually identified on the views the classes appearing to be quality-critical, regardless of their variability, by enumerating the classes that appeared to be the most cracked and/or red to obtain a set of "noticeable classes *w.r.t.* criticality". For example, for GeoTools (Figure 7.4), `Hints`, `Query`, `SimplifyingFilterVisitor`, and `FilterToSQL` are easily discernible. The quality-critical and variability intense classes of the project thus correspond to the intersection between the two sets of classes (*i.e.*, in this example, `FilterToSQL` and `Query`).

Observations In all observed systems, it appears that although fewer classes are noticeable *w.r.t.* criticality than *w.r.t.* variability, there is no direct relation between variability and quality, as it can already be seen in Figure 7.4. Whereas some *vp*-shave an important number of variants, they can be reliable, such as `FilterFactoryImpl` in GeoTools, and thus do not need particular attention. On the opposite, some critical classes may not concentrate variability implementations, such as `Hints` in GeoTools, and they are therefore less important for maintaining the functional code. This shows that, in the studied systems, visualizing both variability and quality is useful to determine quality-critical variability implementations. To evaluate to which extent, we calculated for each project the number of noticeable classes *w.r.t.* variability, *w.r.t.* criticality, and *w.r.t.* both aspects. The results with the configuration for each view are reported in Table 7.2. This shows that representing on a single view variability and quality information allows reducing the number of classes appearing as relevant on the visualization between 50% (JKube) and 91% (Spring framework) compared to the *VariCity* visualization. We believe the mildly encouraging results obtained on JKube come from its size, so that less variability intense zones have been identified by *VariCity* compared to larger projects. An important number of classes are also noticeable in this project as it has globally a low code coverage. Besides, by adapting the thresholds on which the hotspot detection relies, we could obtain fewer zones and better results, but we consider these experiments as out of the scope of this chapter. The definition of a hotspot is elusive (Section 5.1.3) and determining whether a class is a hotspot or not depends on user-defined thresholds, a limitation already evoked in the work on *VariCity* (Chapter 6). Nevertheless, we consider these results as satisfying, because without *VariMetrics*, finding OO variability debt would have needed to manually map relevant classes on the *VariCity* view to their metrics, which, already on the smallest project being JKube, represents 28 classes.

Summary By representing OO variability implementations and quality metrics in a unified representation, *VariMetrics* not only allows to visualize both classes concentrating variability implementations and critical classes, but also to focus on specific zones of OO variability debt.

7.4.2 Qualitative evaluation

Identifying technical debt helps to understand where to apply maintenance actions aiming to improve software quality. Therefore, if zones of variability debt identified by *VariMetrics* are relevant, correcting identified weaknesses should improve the project quality, and the effects should be visible in the visualization. To validate the relevance of these zones, we conduct an experiment in which we apply modifications to the identified classes in one project, JFreeChart.



(a) Before refactor (b) After refactor

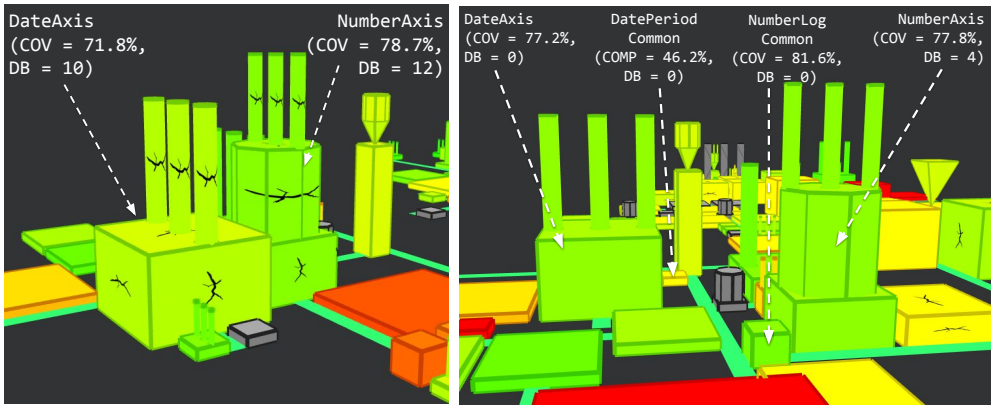
Figure 7.5: View of `XYPlot` and `CategoryPlot` before and after the refactor. Block duplications are displayed on the red-green scale (range: 0 → 50 blocks) and test coverage using the crackled texture (range: 0% → 100%).

Subject system We chose JFreeChart as a subject system not only for its intermediate size allowing an easy discovery of the codebase, but also because this system has been extensively studied in previous work [Tërnava et al., 2019; Tërnava et al., 2022; Mortara et al., 2021a], thus for which we have details on the implemented variability.

Evaluation process We first selected in the set of 10 critical variability intense classes determined in the quantitative evaluation (*cf.* Table 7.2) the ones maximizing their number of duplicated blocks or minimizing their test coverage. Six classes remained, of which four suffer from code duplication (`CategoryPlot`, `XYPlot`, `DateAxis`, and `NumberAxis`, visible due to their extensively cracked texture on Figure 7.5a for the first two and on Figure 7.6a for the last two) and two others from a lack of tests (`ChartPanel` and `ChartEntity`, visible due to their orange and yellow colors on Figure 7.7a).

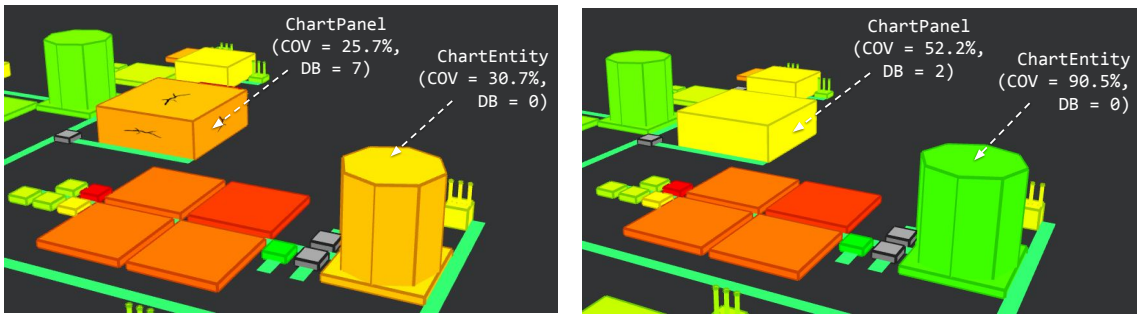
We then defined and applied maintenance actions for these classes. Regarding classes suffering from code duplication, duplicated blocks were factorized in new methods. It happened that block duplications were present in different classes (*e.g.*, behavior from `CategoryPlot` is duplicated in `XYPlot`). In this case, the factorization was placed in another class, created for that purpose (here, `CategoryXYCommon`). Regarding classes lacking tests, new test cases for several methods that were little to not tested have been added to the existing test classes. To ensure as much as possible that our modifications did not hamper the system stability, we did not change the logic of existing tests and made sure that the project could build with all tests passing.

Observations A first observation we made concerns the nature of the duplicated blocks. Whereas some duplications are pure technical debt in classes concentrating variability implementations, others clearly correspond to improperly managed variability implementations. For



(a) Before refactor (b) After refactor

Figure 7.6: View of `DateAxis` and `NumberAxis` before and after the refactor. Visualization settings are identical to Figure 7.5.



(a) Before refactor (b) After refactor

Figure 7.7: Classes lacking tests before and after the refactor. Visualization settings are identical to Figure 7.5.

example, in `DateAxis`, multiple lines of the `refreshTicksHorizontal`[8] method are duplicated in `refreshTicksVertical`[9]. They correspond to the common part creating the time tick, whereas the variable part concerns the orientation of the text on the plot. Therefore, such zones exhibited by *VariMetrics* actually spot improper variability management. We reapplied *VariMetrics* on the new codebase [Mortara et al., 2022] and observed the differences shown in Figures 7.5b, 7.6b and 7.7b. We also computed the test coverage, cognitive complexity, and number of duplicated blocks for all the classes impacted by our maintenance actions before and after their modification, and summarized these results in Table 7.3.

Regarding the classes suffering from code duplications, evolutions can be observed in Figures 7.5 and 7.6. The disappearance of the cracks on `NumberAxis` and `DateAxis` suggests that very little to no duplication remains, while the reduced amount of cracks on `CategoryPlot` denotes a decrease of duplications while some are still present. Finally, `XYPlot` appears equally cracked, propounding that duplications are still present. These observations are confirmed by the

[8] https://github.com/jfree/jfreechart/tree/v1.5.0/src/main/java/org/jfree/chart/axis/DateAxis.java#L1558-L1609
[9] https://github.com/jfree/jfreechart/tree/v1.5.0/src/main/java/org/jfree/chart/axis/DateAxis.java#L1676-L1726

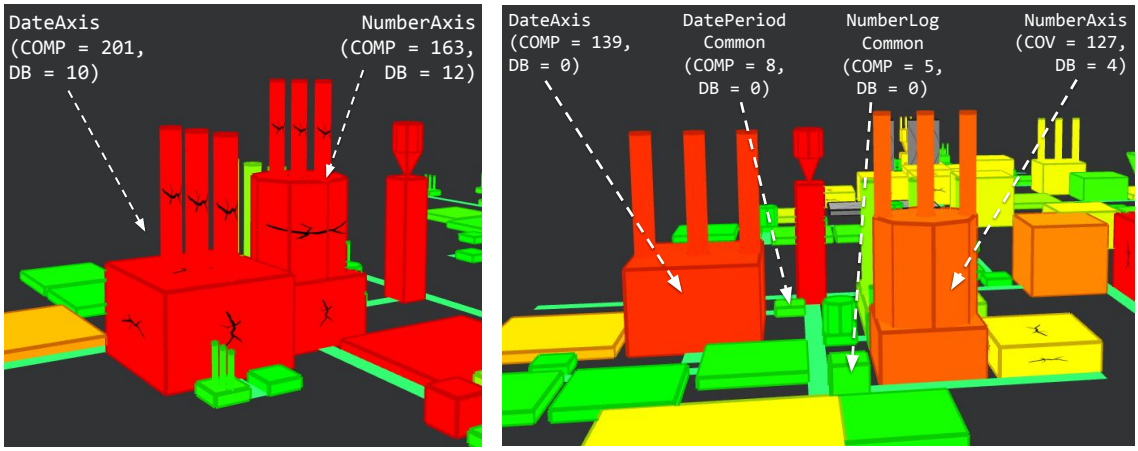
Table 7.3: Measures of the refactored and added classes, before and after the refactor.

Class name (in the `org.jfree.chart` package)		**Coverage**	**# duplicated blocks**	**Cognitive complexity**
Identified relevant classes				
`plot.CategoryPlot`	*before*	74.9%	23	503
	after	75.4%	16	392
`plot.XYPlot`	*before*	70.0%	24	666
	after	69.9%	21	603
`axis.DateAxis`	*before*	71.8%	10	201
	after	77.2%	0	139
`axis.NumberAxis`	*before*	78.7%	12	163
	after	77.8%	4	127
`entity.ChartEntity`	*before*	30.7%	0	26
	after	90.5%	0	26
`ChartPanel`	*before*	25.7%	7	322
	after	52.2%	2	295
Other already present classes				
`axis.LogarithmicAxis`	*before*	16.0%	4	315
	after	17.1%	0	281
`axis.LogAxis`	*before*	45.3%	10	92
	after	47.0%	7	87
`axis.PeriodAxis`	*before*	29.3%	2	112
	after	30.6%	1	104
Added classes				
`plot.CategoryXYCommon`	*before*	–	–	–
	after	77.4%	6	97
`axis.DatePeriodCommon`	*before*	–	–	–
	after	46.2%	0	8
`axis.NumberLogCommon`	*before*	–	–	–
	after	81.6%	0	5
Overall project	*before*	54.5%	604	15,858
	after	55.3%	565	15,622

values from Table 7.3: duplications in `NumberAxis` and `DateAxis` have been reduced by 75% and 100%, leaving respectively 4 and 0 duplications. Although the number of duplications in `CategoryPlot` diminished by 29%, 16 duplicated blocks remain, representing a non-negligible amount. Finally, 3 duplications have been removed in `XYPlot`, representing 13% of reduction, that is not significative enough to be shown on the visualization.

Similarly, improvements can also be seen in the classes that were lacking tests (Figure 7.7b). The transition from 31% to 91% of coverage for `ChartEntity` is translated on the visualization by a bright green color for its building, where the more contained improvement on `ChartPanel`'s coverage leads to its building color changing from orange to yellow.

Another effect induced by these maintenance actions can be seen in the visualization. The crack on `ChartPanel`'s building visible in Figure 7.7a disappeared in Figure 7.7b, although removing duplications was not a maintenance action for this class. This is because testing some methods required splitting them, leading to smaller blocks that could be reorganized. In this case, three duplicated blocks were extracted in a single testable method.



(a) Before refactor

(b) After refactor

Figure 7.8: Figure 7.6 displaying cognitive complexity on the red-green scale instead of the coverage (range: 0 → 150).

Finally, it appears that the maintenance actions on these classes improved their quality *w.r.t.* the considered metrics (*i.e.*, coverage and duplicated blocks). These changes however did not only impact the six considered classes, but also three other existing classes having duplications and led to the creation of three new classes to host some duplications. It is therefore important to consider these classes and ensure that they do not express the variability debt that has been treated. Modifications applied to the already existing classes solely concern the removal of duplications, therefore their quality has also been improved. Regarding the newly created classes, they are now visible (*cf.* Figures 7.5b and 7.6b). `DatePeriodCommon`'s yellow color presents a relatively low test coverage of 46%, which can be explained by the low initial test coverage of `PeriodAxis` of 29.3%. Adding tests would help to solve the issue. The other two classes have high coverages above 70%, and none of the three classes has a cracked texture, showing that no variability debt related to these metrics has been created.

By presenting the coverage and the number of duplicated blocks, the visualizations exhibited in Figures 7.5 to 7.7 can only demonstrate variability debt related to those two metrics. However, as explained in Section 7.1, cognitive complexity is also a factor of variability debt. As this metric is significant for JFreeChart (*cf.* Table 7.1), it is thus important to evaluate its evolution. It appears in Table 7.3 that the cognitive complexity globally decreased for all relevant classes and the other already present ones. This can be explained by the fact that removing code duplications and adding tests often implies splitting methods into smaller ones, thus reducing cognitive complexity. This decrease can also be observed using *VariMetrics*, as its configuration capabilities easily allow adapting the view to display this metric (*cf.* Figure 7.8 with an intensity decrease on `DateAxis` and `NumberAxis`). Concerning the newly created classes, `CategoryXYCommon`'s important cognitive complexity of 97 is because `CategoryPlot` and `XYPlot` have major cognitive complexities of 503 and 666 respectively. Therefore, the factorized blocks are themselves complex, and would need further refactoring (*e.g.*, splitting into separate methods) to reduce this complexity and remove its 6 duplicated blocks.

Summary By implementing maintenance actions on the identified quality-critical variability intense classes, we improved their quality regarding the considered metrics without introducing new debt factors, leading to a positive impact at the project level. These changes are also clearly observable in the visualization. Moreover, part of the identified variability debt directly concerning roughly managed variability that could be refactored.

7.5 Improving *VariMetrics*' usability

7.5.1 Requirements

In practice, developers use Integrated Development Environments (IDEs) as tools support to assist development and program comprehension activities [Minelli et al., 2015]. Xia et al. [2017] measured not only that program comprehension activities represent on average more than half of the developers' time (58%), but also that, on average, around 30% of the time spent in program comprehension activities consisted in switching between the IDE (to comprehend the source code) and the web browser (to obtain additional information and help the comprehension, such as *"bug fixing solutions, feature implementation suggestions, or tool installation guides"*). The IDE, therefore, represents an important tool for program comprehension, especially for developers with less experience.

We could observe this behaviour during the controlled experiment conducted on *VariCity* (Section 6.4). Subject students were split in two groups and had to complete similar variability comprehension tasks using either the *VariCity* visualization or the source code open in an IDE. It resulted that not only subjects having access to the IDE heavily relied on it to explore the codebase, but also that subjects using *VariCity* felt that not having access to the source code was limiting their comprehension of the system and its implemented variability, suggesting that both approaches should be used simultaneously.

While it would be possible to use *VariMetrics* and an IDE together, this would imply important context switching as a user would need to manually input information from their IDE in *VariCity* and then go back to it to explore the classes exhibited by the visualization, implying important context switching that is known to hamper the concentration of developers [Xia et al., 2017]. According to these limitations, we advocate that *VariMetrics*' usability would be improved by being embedded into an IDE. We therefore provide an integration for the popular IDE JetBrains IntelliJ IDEA[10].

7.5.2 IDE integration

As the goal of this integration is to minimize the interactions out of the development environment, the integration embeds all the interactions needed to configure and use *VariMetrics*. As shown on Figure 7.9, the visualization is embedded in a panel in the IDE window and provides controls over *symfinder* and *VariMetrics* (see violet boxes in Figure 7.9), allowing the execution of the whole toolchain from the editor's window. A dedicated entry in the IDE's settings allows to configure the view and the execution of the toolchain (*cf.* Figure 7.10).

Additionally, bidirectional navigation between the visualization and the code is provided to ease transitioning between the code and its visual representation. On one side, it is possible to select multiple buildings in the visualization and to open their sources as tabs in the IDE. On the

[10] https://www.jetbrains.com/idea/

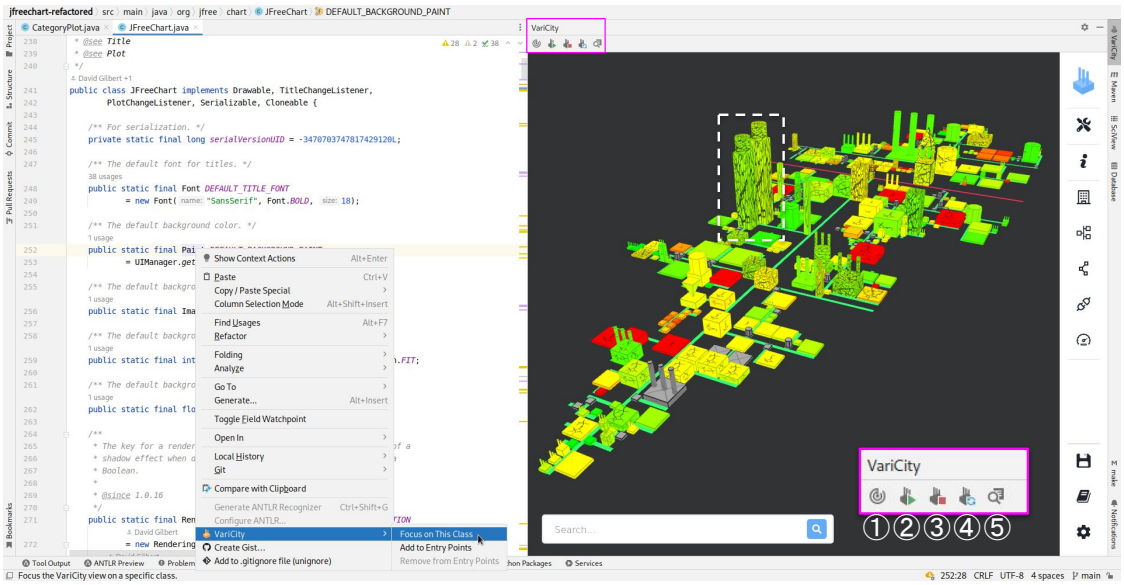


Figure 7.9: *VariMetrics* visualization of JFreeChart embedded in the IDE. The white and violet boxes have been manually added on the figure. In the visualization panel, buttons allow to ① run *symfinder* on the codebase, ② start the *VariCity* visualization server, ③ stop it, ④ reload it, and ⑤ open the classes corresponding to selected buildings in the IDE's editor.

other side, IntelliJ's context menu has been enriched, and right-clicking on a class in the `Project` sidebar or on the name of the class in the editor panel proposes a `Focus on This Class` button, zooming the visualization on the desired class, and another entry to add or remove a class from the entry points list. Finally, the plugin settings window allows to configure the usage level and usage orientation, and to browse the classes of the project to add them as entry points. It also embeds all the configuration capabilities of *VariMetrics*. The IDE integration is developed using the IntelliJ Platform SDK[11] and is installed as any other plugin for the IDE.

7.6 Threats to validity and Limitations

7.6.1 Threats to validity

As we did not conduct an empirical evaluation on the evolution of the view and its integration in the IDE, the major threat of our work is related to the design and realization of the evaluations done by ourselves, including the configuration of the views and choice of the metrics. Nevertheless, the scenarios demonstrating *VariCity* and the results of the empirical evaluation conducted on this approach gave us insights into the criteria to design views exhibiting relevant variability implementations. The metrics choice was driven by recent work on the factors causing variability debt [Wolfart et al., 2021], giving us confidence in their relevance in our context. Moreover, as the views we obtained allowed us to obtain positive results, we expect real experts to obtain good outcomes on their systems by applying their settings. The interactions provided by the IDE

[11] https://plugins.jetbrains.com/docs/intellij/welcome.html

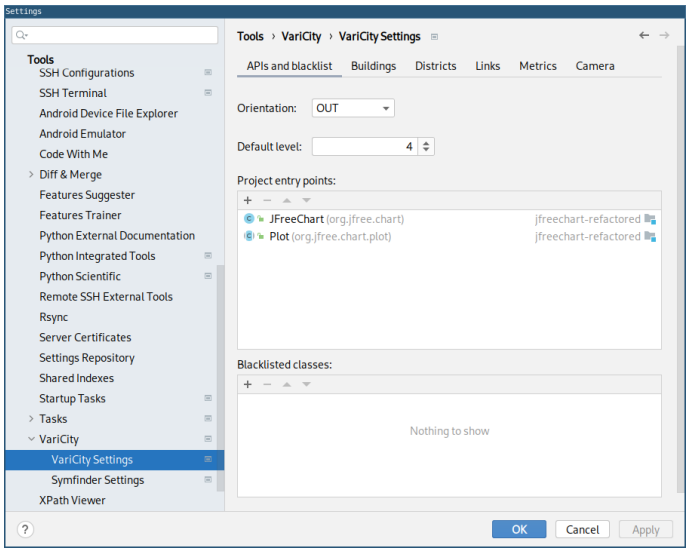


Figure 7.10: The view can be configured from a dedicated menu in the IDE settings.

integration have been designed following the IntelliJ Platform UI Guidelines[12] to ensure visual homogeneity in the editor, thus giving us confidence in its usability. In our own experience with the IDE integration, all exploration tasks conducted in the validation of the metrics part described in Section 7.4 were heavily facilitated.

We evaluated our approach on 7 systems. Although this dataset is small, the studied systems have various sizes (40k → 2.5M LoC) and architectures (API, standalone library…), and represent different domains (charting, programming language, geospatial data management…). We are thus confident in the applicability of our results to other Java-based systems.

7.6.2 Limitations

Regarding the visualization, we chose to offer as many configuration capabilities as possible to the expert so that they can tailor it freely and reach the view that helps them most. Combining multiple metrics on different axes can yet induce cognitive load and hamper the view's understanding. While measuring this load is of prime importance when designing visualizations [Huang et al., 2009] including city-based ones [Caserta et al., 2011; Dashuber et al., 2021] to ensure readability and usability, it would require in our case to empirically validate our approach with real experts to exchange on their needs[13].

As for scalability, the analysis part is directly related to *symfinder* capabilities, which can handle projects with several millions of LoC but takes hours to do so (more than 120h for the JDK as shown in Table 7.4). As the analysis can be synchronized with main releases, this is still reasonable for such very large projects. On the rendering side, this extension of *VariCity* only configures coloring and adds some textures, which are negligible for the rendering time. We are thus dependent on the main bottleneck of *VariCity* rendering, which lies in the computation of the city shape and streets (more than 5 five minutes for the JDK). For very large projects, this is

[12] https://jetbrains.github.io/ui/

[13] Such a validation would also exhibit potential accessibility issues (for example, regarding the choice of colors) that can be tackled by extending the existing configuration capabilities.

Table 7.4: Subject systems and their execution times.

System	*symfinder* execution	*VariMetrics* city rendering
Azureus	1 h 25 min	4 s
GeoTools	24 h	1 min 10 s
JDK	123 h 22 min	5 min 40 s
JFreeChart	5 min 13s	1 s
JKube	2 min 8 s	< 1 s
OpenAPI Generator	3 min 45 s	< 1 s
Spring framework	1 h 5 min	6 s

System for *symfinder*: Ubuntu 18.04.2 LTS with Intel Xeon CPU E5-2637v2 @ 3.5GHz and 128Go memory.
System for *VariMetrics*: Google Chrome 99.0.4844.84 on Arch Linux 5.16.16-arch1-1 with Intel i7-9850H (12 cores) @ 4.6GHz and 32Go memory.

hampering the configuration of the view as recomputation must be done when usage orientation and levels are changed. Nevertheless, from our analysis of the algorithm used in *VariCity*, we believe that some significant improvements could be made to make (re-)rendering practicable.

7.7 Accessibility of the artifacts

A reproducible artifact is available online as an archive [Mortara et al., 2022] containing the source code of *VariMetrics*, the Excel file used to obtain the data presented in Table 7.2, additional views for all projects presented in Table 7.2, the codebases of JFreeChart before and after the refactor presented in Section 7.4.2 (with the corresponding diff file, excerpts of the SonarQube analysis of both codebases showing the information presented in Table 7.3. These information can also be found on a companion webpage[14].

7.8 Conclusion

In this chapter, we tackled the following questions:

1. **How to identify indebted zones of variability implementations?**
   (a) **How to measure technical debt related to OO variability implementations?** Out of the 10 types of variability debt identified by Wolfart et al. [2021], three types are applicable to our studied codebases: *Code duplication*, *System-level structure quality issues* in the implementation and *Lack of tests*. They can be identified using standard OO quality metrics, namely the number of duplicated blocks, the cognitive complexity and the test coverage, respectively.
   (b) **How to reveal technical debt related to OO variability implementations?** We propose *VariMetrics*, an extension of the city-based *VariCity* visualization to support the organized display of OO quality metrics as additional visual properties in a city in

[14] https://deathstar3.github.io/varimetrics-demo/

which dense zones of highly visible buildings already show zones of potential variability implementations.

2. **Does *VariMetrics* allow visualizing indebted zones of variability implementations?** Representing OO variability implementations and quality metrics in a unified representation allowed on the studied subject systems to visualize both classes concentrating variability implementations and critical classes, enabling to focus on specific zones of OO variability debt.

3. **Are the revealed indebted zones of variability implementations relevant?** Implementing maintenance action on JFreeChart's classes being identified as both variability intense and quality-critical allowed to improve their quality regarding the considered metrics without introducing new debt factors, therefore leading to a positive impact at the project level clearly observable in the visualization. Additionally, it could be notice that part of the identified variability debt directly concerned roughly managed variability, and could be refactored.

By superposing code quality metrics on the original *VariCity* view, the *VariMetrics* view allows a user to focus on quality-critical zones concentrating variability implementations, answering Challenge A3. Moreover, with the IDE integration, the configuration and execution of both the identification of variability implementations and the view are available in the development environment. The integration also enables bidirectional navigation between the classes in the code editor and their building representation in the city, further helping the comprehension of the variability implementations and improving the answer to Challenge A2 given by *VariCity*. Further assessing the completion of these challenges would require a user evaluation of both *VariMetrics* and its integration in the IDE.

PART II

Comprehending the variability managed by build systems

CHAPTER 8

A unified representation for anomalies in the Linux build system

This chapter shares material with the SPLC 2021 paper *"Capturing the diversity of analyses on the Linux kernel variability"* [Mortara and Collet, 2021a] and its companion technical report [Mortara and Collet, 2021b].

As detailed in Sections 3.4 and 3.5, existing work on the derivation mechanism of build systems and their anomalies is focused on the Linux kernel. While these contributions cover anomalies in and between all spaces of constraints, their expressions are not aligned, exhibiting incoherences. Additionally, their differences in granularity prevent a fine grain understanding of the interactions between assets in different spaces. There is therefore a need to bring together these definitions under a unified formalism detailing with precision how an asset selection impacts the selection in another one throughout the build system.

In this chapter, we tackle both Challenges B1 and B2 (*"Making explicit the derivation mechanism of build systems"* and *"Characterizing and identifying anomalies in build systems"* respectively) by bringing together existing formalizations in a single formalism that captures all relevant elements of the Linux kernel variability. Instead of extracting a partial representation to reason about it, our formalism first considers selectable entities of the entire build workflow, *i.e.*, features, files and code blocks, to express properties. These properties are determined over two concepts: a configurator, which can represent the KCONFIG, and a derivator, which can be instantiated differently to represent the KBUILD (to select files) and CPP (to select code blocks) (Section 8.1). We then show the instantiated models and express the already identified defects from the main previous contributions, establishing their coverage on the defined properties (Section 8.2).

8.1 Proposed Models

The Linux kernel build system is made of three steps (Section 2.3.1). The KCONFIG configures the system by defining constraints on features. Then, the KBUILD defines conditions to select source files, and CPP defines conditions to select code blocks. In order to cover all three stages of the Linux kernel build system and to represent its variability mechanisms independently of

their implementation, we capture the mechanisms implemented by all three steps of the kernel and abstract them in a single representation. The stages are represented as follows:

- a *configurator* defines presence conditions on features (*i.e.*, the condition allowing an individual feature to be selected). Presence conditions on features are propositional formulas on other features;
- a *derivator* defines presence conditions on assets, (*i.e.*, the condition allowing an individual asset to be selected). Presence conditions on assets are propositional formulas on both features and assets, which can be either of the same type or of another type.

Consequently, the Linux kernel build system has one configurator, KCONFIG, and two derivators. The first one, KBUILD, defines presence conditions on source files. The second one, CPP, defines presence conditions on code blocks.

Some properties will also be defined on the internal and external consistency of the elements as to cover the different anomalies devised in the previous section. We could also have built our models on a more expressive theoretical background, such as the refinement theory [Borba et al., 2012], to potentially obtain for free some properties, but we decided to rely on a more simple but very explicit basis to clarify first all concepts and inconsistencies.

In the following, we will also use these utility definitions:

$terms(\phi)$ a helper function which, given a propositional formula, returns the terms in it (*e.g.*, $terms\left((A \wedge B) \vee C\right) = \{A, B, C\}$).

$expand(\phi)$ a helper function which, given a propositional formula ϕ, replaces every asset a in ϕ by its presence condition, noted $\mathcal{PC}_{Int}(a)$ and defined in Definition 8.2 (*e.g.*, $expand\left(b_1 \wedge \neg b_2\right) = \mathcal{PC}_{Int}(b_1) \wedge \neg\mathcal{PC}_{Int}(b_2)$).

$slice(C, T)$ an operator which, given a set of boolean conditions on terms C and a set of terms T, returns the conjunction of all propositional formulas from C containing terms from T. The operator is recursively applied to the terms that appear in these formulas[1] (*e.g.*, $slice\left(\{A \wedge B, A \wedge C, C \vee D, H \wedge I\}, \{A\}\right) = \{A \wedge B, A \wedge C, C \vee D\}$).

8.1.1 Derivator Model

In this section, we introduce the concepts to form the derivator model and illustrate them with its application to CPP.

Definition 8.1 (Asset)**.** An asset $a = \langle \phi_{select}, \phi_{preds}, \phi_{depInt}, \phi_{depExt} \rangle$ from a set of assets $\mathcal{A}_X$ is defined as follows:

- ϕ_{select} is a propositional formula for the asset's selection ;
- ϕ_{preds} is a propositional formula on other assets that are evaluated before a. We call these assets *predecessors* ;
- ϕ_{depInt} is a propositional formula on assets on which a is dependent ;
- ϕ_{depExt} is a propositional formula on assets from another context on which a is dependent.

[1] The principle of slicing has already been applied to feature models [Acher et al., 2011; Krieter et al., 2016] and its goal is to extract a subset of formulas equivalent to the whole space by keeping only formulas relevant to terms from T.

Application to CPP An asset b is a code block, with:

- ϕ_{select} the condition of the `#if` surrounding the block ;
- $\phi_{preds} = \neg(\bigvee_i b_i)$ if b is an `#elsif` or `#else` block, b_i represents the corresponding `#if` block and the potential `#elsif` blocks before b ;
- $\phi_{depInt} = p$ with p the parent block of b if b is a nested block.
- $\phi_{depExt} = file$ the file containing b.

Example. In Figure 2.2, the `lib/dir/foo.c` file and the `B1`, `B2`, `B3`, and `B4` blocks it contains are represented by the following assets:

- $file = \langle true, true, true, true \rangle$
- $b_1 = \langle FOO \vee BAR \vee BAZ, true, true, file \rangle$
- $b_2 = \langle FOO, true, b_1, file \rangle$
- $b_3 = \langle BAR, \neg b_2, b_1, file \rangle$
- $b_4 = \langle true, \neg(b_2 \vee b_3), b_1, file \rangle$

Application to KBUILD At the KBUILD level, an asset $s = \langle \phi_{select}, \phi_{preds}, \phi_{depInt}, \phi_{depExt} \rangle$ can represent a C object file. We then express presence conditions and related anomalies with our model. As seen in Section 2.3.1, an object is selected for compilation by being added to defined lists, with possible constraints on one or more features in case of multiple definitions. Before, objects can also be added to composite variables.

- $\phi_{select} = \bigvee f_i$ with f_i being features which at least one needs to be set for the source file to be selected. If the asset is always selected, $\phi_{select} = true$. If the asset is defined but never added to a list, $\phi_{select} = false$;
- $\phi_{preds} = comp$ with $comp$ the name of the composite variable if s is part of a composite definition. $comp$ must be selected ;
- $\phi_{depInt} = dir$ with dir the directory containing the source file represented by s which also needs to be selected ;
- $\phi_{depExt} = true$ as the selection of a source file only relies on its feature.

Example. In Figure 2.2, the `dir` directory and the `foo.c`, `file_a.c`, `file_b.c`, and `file_c.c` files it contains are represented by the following assets:

- `dir` $= \langle BAR, true, true, true \rangle$
- `foo.c` $= \langle FOO, true, \texttt{dir}, true \rangle$
- `file_a.c` $= \langle true, \texttt{foo.c}, \texttt{dir}, true \rangle$
- `file_b.c` $= \langle true, \texttt{foo.c}, \texttt{dir}, true \rangle$

- `file_c.c` $= \langle true, true, \texttt{dir}, true \rangle$

Definition 8.2 (Internal presence condition)**.** The internal presence condition of an asset is the boolean formula that needs to be satisfiable for the asset to be selectable. It is defined as

$$\mathcal{PC}_{Int}(a) = \phi_{select_a} \wedge expand\left(\phi_{preds_a}\right) \wedge expand\left(\phi_{depInt_a}\right)$$

Note. An asset is selected if and only if its presence condition is satisfied: $\mathcal{PC}_{Int}(a) \Leftrightarrow a$.

Application to CPP Let us take again the previous example.

$$\begin{aligned}
\mathcal{PC}_{Int}(b_1) &= \phi_{select_{b_1}} \wedge expand\left(\phi_{preds_{b_1}}\right) \wedge expand\left(\phi_{depInt_{b_1}}\right) \\
&= (FOO \vee BAR \vee BAZ) \wedge expand(true) \wedge expand(true) \\
&= (FOO \vee BAR \vee BAZ) \\
\mathcal{PC}_{Int}(b_2) &= \phi_{select_{b_2}} \wedge expand\left(\phi_{preds_{b_2}}\right) \wedge expand\left(\phi_{depInt_{b_2}}\right) \\
&= (FOO) \wedge true \wedge expand\left(b_1\right) \\
&= (FOO) \wedge (FOO \vee BAR \vee BAZ) \\
\mathcal{PC}_{Int}(b_3) &= \phi_{select_{b_3}} \wedge expand\left(\phi_{preds_{b_3}}\right) \wedge expand\left(\phi_{depInt_{b_3}}\right) \\
&= (BAR) \wedge (\neg \mathcal{PC}_{Int}(b_2)) \wedge \mathcal{PC}_{Int}(b_1) \\
&= (BAR) \wedge (\neg (FOO \wedge (FOO \vee BAR \vee BAZ))) \wedge (FOO \vee BAR \vee BAZ) \\
&= (BAR) \wedge \neg (FOO) \wedge (FOO \vee BAR \vee BAZ) \\
\mathcal{PC}_{Int}(b_4) &= \phi_{select_{b_4}} \wedge expand\left(\phi_{preds_{b_4}}\right) \wedge expand\left(\phi_{depInt_{b_4}}\right) \\
&= \neg (\mathcal{PC}_{Int}(b_2) \vee \mathcal{PC}_{Int}(b_3)) \wedge \mathcal{PC}_{Int}(b_1) \\
&= \neg (FOO \vee BAR) \wedge (FOO \vee BAR \vee BAZ)
\end{aligned}$$

Note. Extracted presence conditions can be complex and may contain redundant terms (*e.g.*, $\mathcal{PC}_{Int}(b_2)$ is equivalent to FOO). Approaches to simplify presence conditions have been proposed [Von Rhein et al., 2015] and are out of the scope of our work.

Application to KBUILD

$$\begin{aligned}
\mathcal{PC}_{Int}(\texttt{dir}) &= \phi_{select_{\texttt{dir}}} \wedge expand\left(\phi_{preds_{\texttt{dir}}}\right) \wedge expand\left(\phi_{depInt_{\texttt{dir}}}\right) \\
&= BAR \wedge true \wedge true \\
&= BAR \\
\mathcal{PC}_{Int}(\texttt{foo.c}) &= FOO \wedge true \wedge expand(\mathcal{PC}_{Int}(\texttt{dir})) \\
&= FOO \wedge BAR \\
\mathcal{PC}_{Int}(\texttt{file_a.c}) &= true \wedge expand(\mathcal{PC}_{Int}(\texttt{foo.c})) \wedge expand(\mathcal{PC}_{Int}(\texttt{dir})) \\
&= (FOO \wedge BAR) \wedge BAR \\
\mathcal{PC}_{Int}(\texttt{file_b.c}) &= \mathcal{PC}_{Int}(\texttt{file_a.c}) \\
\mathcal{PC}_{Int}(\texttt{file_c.c}) &= true \wedge true \wedge expand(\mathcal{PC}_{Int}(\texttt{dir})) \\
&= BAR
\end{aligned}$$

Definition 8.3 (External presence condition)**.** By evaluating $\mathcal{PC}_{Int}$, we check that the asset can be selected given the constraints of its space. However, other external constraints may prevent the selection the asset. We call *context* the set of these constraints. The external presence condition of an asset in a given context $\mathcal{C}$ is defined as

$$\mathcal{PC}_{Ext}(a) = \mathcal{PC}_{Int}(a) \wedge slice(\mathcal{C}, terms(\mathcal{PC}_{Int}(a)) \cup terms(\phi_{depExt_a}))$$

Application to CPP In the Linux build system, the selection of a CPP block is conditioned by constraints on both the features used in the `#if` instructions (which are determined at the KCONFIG level) and the file containing the block (which are determined at the KBUILD level). Thus, the context $\mathcal{C}$ to express the external presence condition of a block is the union of the KCONFIG and KBUILD contexts $\mathcal{C} = \mathcal{C}_{\text{KCONFIG}} \cup \mathcal{C}_{\text{KBUILD}}$. Let us take an example with

$$\mathcal{C}_{\text{KCONFIG}} = \{FOO \rightarrow BAR, BAZ \rightarrow (\neg F1), F1 \rightarrow (\neg FOO), F3 \rightarrow F4\}$$
$$\mathcal{C}_{\text{KBUILD}} = \{file \leftrightarrow FOO\}$$

then

$$\begin{aligned}\mathcal{PC}_{Ext}(b_1) &= \mathcal{PC}_{Int}(b_1) \wedge slice(\mathcal{C}, terms(\mathcal{PC}_{Int}(b_1)) \cup terms(\phi_{depExt_{b_1}})) \\ &= \mathcal{PC}_{Int}(b_1) \wedge slice(\mathcal{C}, \{FOO, BAR, BAZ\} \cup \{file\}) \\ &= \mathcal{PC}_{Int}(b_1) \wedge ((FOO \rightarrow BAR) \wedge (BAZ \rightarrow (\neg F1)) \\ &\qquad \wedge (F1 \rightarrow (\neg FOO)) \wedge (file \leftrightarrow FOO))\end{aligned}$$

8.1.1.1 Internal consistency

To express defects, we define dead, core, and full-mandatory assets, relying on definitions of dead and false-optional features introduced by Benavides et al. [2010], and full-mandatory features from Trinidad et al. [2008].

Definition 8.4 (Dead asset)**.** An asset a of $\mathcal{A}$ is dead if it can never be selected. The set of dead assets of $\mathcal{A}$ is noted $deads(\mathcal{A})$.

$$a \in deads(\mathcal{A}) \Leftrightarrow \neg sat(\mathcal{PC}_{Int}(a))$$

Note. This consistency check includes the more specific case where an asset is dead because of an inconsistency with the condition to select its internal dependencies (*i.e.*, $expand(\phi_{depInt_s}) \rightarrow \neg\phi_{select_s}$) as in this case $\mathcal{PC}_{Int}(a)$ is inconsistent.

Definition 8.5 (Core asset)**.** An asset a of $\mathcal{A}$ is a core asset if it is always selected. The set of core assets of $\mathcal{A}$ is noted $core(\mathcal{A})$.

$$a \in core(\mathcal{A}) \Leftrightarrow \neg sat(\neg(\mathcal{PC}_{Int}(a)))$$

8.1.1.2 External consistency

Definition 8.6 (Externally dead asset)**.** An asset a is an externally dead asset if it is never selected due to inconsistencies with its context. The set of externally dead assets of $\mathcal{A}$ is noted $deadsExt(\mathcal{A})$.

$$a \in deadsExt(\mathcal{A}) \Leftrightarrow \neg sat(\mathcal{PC}_{Ext}(a))$$

Definition 8.7 (Externally core asset)**.** An asset a of $\mathcal{A}$ is an externally core asset if it is always selected independently of the constraints of the context. The set of core assets of $\mathcal{A}$ is noted $coreExt(\mathcal{A})$.

$$a \in coreExt(\mathcal{A}) \Leftrightarrow \neg sat(\neg(\mathcal{PC}_{Ext}(a)))$$

Definition 8.8 (Externally full-mandatory asset)**.** An asset a is an externally full-mandatory asset if the selection of its parent dependencies implies its selection due to the formulas in its context. The set of externally full-mandatory assets of $\mathcal{A}$ is noted $mandExt(\mathcal{A})$.

$$\begin{aligned} a \in mandExt(\mathcal{A}) &\Leftrightarrow expand(\phi_{depInt_a}) \rightarrow \mathcal{PC}_{Ext}(a) \\ &\Leftrightarrow \neg sat(\neg\mathcal{PC}_{Ext}(a) \wedge expand(\phi_{depInt_a})) \end{aligned}$$

Definition 8.9 (Missing dead asset)**.** An asset a is missing dead if a feature in its presence condition is not defined in the context $\mathcal{C}$. The set of assets of $\mathcal{A}$ with missing features is noted $missing(\mathcal{A})$.

$$a \in missing(\mathcal{A}) \Leftrightarrow \exists m \in terms(\mathcal{PC}_{Ext}(a)) \mid (m \notin terms(\mathcal{C}))$$

8.1.2 Configurator Model

The configurator represents the model element that checks the selection of features. It is represented by a set of features $\mathcal{F}$. We will illustrate the formalization here with its application to the KCONFIG.

Definition 8.10 (Feature)**.** A feature $F = \langle \phi_{enable}, \phi_{deps}, \mathcal{F}_{select} \rangle$ from a set of features $\mathcal{F}$ is defined as follows:

- ϕ_{enable} is a propositional formula representing the ability to select the feature ;
- ϕ_{deps} is a propositional formula on features on which F is dependent ;
- $\mathcal{F}_{select}$ is a set of features automatically selecting F. If a feature from $\mathcal{F}_{select}$ is selected, F is also selected, regardless of the precedent conditions.

Application to KCONFIG A feature F is a configuration option defined in a KCONFIG file, with:

- ϕ_{enable} represents the ability to select the feature by user selection (`prompt`), or `default` value, as defined in Table 8.1. We represent the selection of a feature F by a user with a boolean value σ_F ;
- ϕ_{deps} represents the boolean formula on features defined in the `depends on` statement ;
- $\mathcal{F}_{select}$ is a set of features selecting F with a `select` statement ;

In the KCONFIG file presented in Figure 2.2, three features are defined: `FOO`, `BAR` and `F_SEL`. Existing work on the semantics of the KCONFIG files [She and Berger, 2010] inline the conditions from the `menu` items surrounding the definition of a feature in the `depends on` condition. These features can be represented by the following assets:

- $FOO = \langle \sigma_{FOO}, (DEPS\_A \vee DEPS\_B) \wedge MENU\_COND, \{\} \rangle$

Table 8.1: ϕ_{enable} truth table for a KCONFIG feature F

Presence of `prompt`	Presence of `default`	ϕ_{enable}
yes	activated	σ_F (user selection)
	not activated	
no	activated	$true$
	not activated	$false$

- $BAR = \langle \sigma_{BAR}, MENU\_COND, \{\} \rangle$
- $F\_SEL = \langle false, true, \{FOO\} \rangle$

Definition 8.11 (Presence condition)**.** The presence condition of a feature $F \in \mathcal{F}$ represents the boolean formula which needs to be satisfied for the feature to be selected.

$$\mathcal{PC}(F) = (\phi_{enable} \wedge expand(\phi_{deps})) \vee directSelect(F)$$

with $directSelect(F) = \bigvee_{F_s \in \mathcal{F}_{select}} \mathcal{PC}(F_s)$.

Note. The selection of a feature implies that its presence condition is satisfied: $F \rightarrow \mathcal{PC}(F)$. There is no biimplication as we consider that a user can manually interfere in the selection. Therefore, the information extracted from the model can only express if a feature *can be* selected, and not its effective selection.

Application to KCONFIG

$$\begin{aligned}
\mathcal{PC}(FOO) &= (\phi_{enable_{FOO}} \wedge expand(\phi_{deps_{FOO}})) \vee directSelect(FOO) \\
&= \sigma_{FOO} \wedge ((\mathcal{PC}(DEPS_A) \vee \mathcal{PC}(DEPS_B)) \wedge \mathcal{PC}(MENU_COND)) \\
\mathcal{PC}(BAR) &= (\phi_{enable_{BAR}} \wedge expand(\phi_{deps_{BAR}})) \vee directSelect(BAR) \\
&= \sigma_{BAR} \wedge \mathcal{PC}(MENU_COND) \\
\mathcal{PC}(F_SEL) &= (\phi_{enable_{F_SEL}} \wedge expand(\phi_{deps_{F_SEL}})) \vee directSelect(F_SEL) \\
&= (false \wedge true) \vee \mathcal{PC}(FOO) \\
&= \mathcal{PC}(FOO)
\end{aligned}$$

Note. Due to the size and complexity of the KCONFIG model, obtaining a sound and complete abstraction of its semantics is still a challenge. The latest studies on boolean translation are not able to represent the whole complexity of the language [Fernandez-Amoros et al., 2019]. Because of these limitations, the accuracy of variability reasoning approaches is also limited and acknowledged by researchers [Franz et al., 2021]. Therefore, we aim here to provide a model allowing us to synthesize the current work, and do not pretend to present a complete model of KCONFIG[2].

[2]For example, although KCONFIG's syntax allows adding conditions to `select` statements, no defect described in our model requires to express this behaviour.

8.1.2.1 Consistency

Definition 8.12 (Dead feature)**.** A feature F of $\mathcal{F}$ is dead if it can never be selected. The set of dead features is noted $deadFeatures()$.

$$F \in deadFeatures() \Leftrightarrow \neg sat(\mathcal{PC}(F))$$

Definition 8.13 (Core feature)**.** A feature F of $\mathcal{F}$ is a core feature if it is always selected. The set of core features is noted $coreFeatures()$.

$$F \in coreFeatures() \Leftrightarrow \neg sat(\neg\mathcal{PC}(F))$$

Note. If $F_S \in \mathcal{F}_{select_F}$ is a core feature, then F is also a core feature, as $\mathcal{PC}(F_S) \rightarrow \mathcal{PC}(F)$.

Definition 8.14 (Missing dead feature)**.** A feature F is missing dead if a feature in its presence condition is not defined. The set of missing dead features is noted $missingDeadFeatures()$.

$$F \in missingDeadFeatures() \Leftrightarrow (m \in terms(\mathcal{PC}(F)) \wedge (m \notin \mathcal{F})$$

8.2 Instantiation on the Linux kernel

We now instantiate our model on the kernel build system. The configurator is used to model the KCONFIG, while the derivator concept is used to model source files selected by the KBUILD Makefiles, with a form of positive variability [Voelter and Groher, 2007]: the core is represented by the `obj-y` entries, where the additional parts are added in composite objects and feature dependent entries. The same derivator concept also represents the selection of code blocks from the source files by CPP, implementing this time negative variability [Voelter and Groher, 2007].

8.2.1 Model on CPP

The constraints that can influence the selection of a block can be of two natures. First, constraints can issue from the CPP space. For example, on Figure 2.2, the selection of the `B3` block is conditioned by the selection of the `B1` block and the non selection of the `B2` block. Second, constraints can come from other spaces of the build system. Similarly, on Figure 2.2, all blocks from the `lib/dir/foo.c` file can only be selected if the file is selected in the KBUILD step, and if the constraints on the involved features are satisfied. We describe them hereafter.

8.2.1.1 Compliance with presence conditions from Sincero et al. [2010]

For conciseness and to prevent confusion, we name this definition $\mathcal{PC}_{Sin}$ and use the more compact expression given in [Tartler et al., 2011]:

$$\mathcal{PC}_{Sin}(b_i) = expr(b_i) \wedge noPredecessors(b_i) \wedge parent(b_i)$$

We can express $\mathcal{PC}_{Sin}$ using our definition of asset from Definition 8.1. Let us apply $\mathcal{PC}_{Sin}$ on an asset b as defined in Section 8.1.1.

$$\begin{aligned}\mathcal{PC}_{Sin}(b) &= expr(b) \wedge noPredecessors(b) \wedge parent(b) \\ &= \phi_{select_b} \wedge \neg\left(pred_1 \vee pred_2 \vee \cdots \vee pred_n\right) \wedge \phi_{depInt_b} \\ &= \phi_{select_b} \wedge \phi_{preds_b} \wedge \phi_{depInt_b}\end{aligned}$$

ϕ_{preds_b} and ϕ_{depInt_b} are propositions on assets corresponding to the blocks themselves. However, to evaluate the presence condition, these assets have to be expanded to their logical expression.

$$\mathcal{PC}_{Sin}(b) = \phi_{select_b} \wedge expand\left(\phi_{preds_b}\right) \wedge expand(\phi_{depInt_b})$$

The definition of $\mathcal{PC}_{Sin}$ is therefore compliant with our definition of $\mathcal{PC}_{Int}$ given in Definition 8.2.

8.2.1.2 Expressing cross-space formulas

Nadi and Holt [2012] defined multiple anomalies (Anomalies 19, 21, 22 and 24) using different terms, *i.e.*, B_N, C, M, and K, which we now describe with our model.

$B_N \wedge C$ B_N represents a block, and C the constraints in the code space. This expression is true if and only if the block B_N is selected, thus it corresponds to $B_N \leftrightarrow \mathcal{PC}_{Sin}(B_N)$ using Tartler et al. [2011]'s notation and $\mathcal{PC}_{Int}(B_N)$ in our model.

$parent(B_N)$ $parent(B_N)$ represents the selection of the parent of a block, *i.e.*, its enclosing block. This expression corresponds to $expand(\phi_{depInt_{B_N}})$ in our model.

The KCONFIG space (K) K represents the set of constraints in the KCONFIG space, *i.e.*, the constraints on features that allow them to be selected. Tartler et al. [2011] do not use the whole feature model expression as the solving would not scale. They instead identify the features impacting the selection of a given code block using a slicing algorithm to build a minimal but sufficient subset of the configuration space through a recursive application on each new feature found in the presence condition expression. While this mechanism is not made explicit in the formalisms they provide in the paper, these relationships are detailed in the formulas of the presence conditions by their construction.

The make space (M) M represents the set of constraints in the make space, *i.e.*, the constraints on features that allow the selection of source files in the Makefiles. In her Ph.D. thesis [Nadi, 2014], Nadi states: *"since the conflicts in Anomaly 21 arise from looking at the block presence condition as well as the file's presence condition, we call this category of anomalies* code-build anomalies*"*. Thus, to detect defects involving the make space, it is only necessary to have the presence condition of the file containing the analyzed block.

In Section 8.1.1, we instantiate on CPP the definition of external presence condition given in Definition 8.3, using for context $\mathcal{C} = \mathcal{C}_{\text{KCONFIG}} \cup \mathcal{C}_{\text{KBUILD}}$. Thus, $\mathcal{C}_{\text{KCONFIG}} = K$ and $\mathcal{C}_{\text{KBUILD}} = M$, and:

$$slice(\mathcal{C}_{\text{KCONFIG}}, terms(\mathcal{PC}_{Int}(a)) \cup terms(\phi_{depExt_a})) \models K$$
$$slice(\mathcal{C}_{\text{KBUILD}}, terms(\mathcal{PC}_{Int}(a)) \cup terms(\phi_{depExt_a})) \models M$$

We can then express the different formulas in our model.

Instantiation 1 (Expressing code–Make–KCONFIG anomalies – Anomaly 22)**.**

$$\begin{aligned}
&\neg sat(B_N \wedge C \wedge M \wedge K)\\
&\quad\Leftrightarrow \neg sat\left(\mathcal{PC}_{Int}(B_N) \wedge \mathcal{C}\right)\\
&\quad\Leftrightarrow \neg sat\left(\mathcal{PC}_{Ext}(B_N)\right) \text{ (Definition 8.3)}\\
&\neg sat(\neg B_N \wedge parent(B_N) \wedge C \wedge M \wedge K)\\
&\quad\Leftrightarrow \neg sat(\mathcal{C} \wedge \neg\mathcal{PC}_{Int}(B_N) \wedge expand(\phi_{depInt_{B_N}}))\\
&\quad\Leftrightarrow \neg sat\left(\neg\mathcal{PC}_{Ext}(B_N) \wedge expand(\phi_{depInt_{B_N}})\right)
\end{aligned}$$

Anomaly 22 thus expresses dead (Definition 8.6) and full-mandatory defects (Definition 8.8).

Instantiation 2 (Expressing code–Make defects – Anomaly 21)**.** Same as Instantiation 1 with $\mathcal{C} = \mathcal{C}_{\text{KBUILD}}$.

Instantiation 3 (Expressing configurability defects – Anomaly 13)**.** Same as Instantiation 1 with $\mathcal{C} = \mathcal{C}_{\text{KCONFIG}}$.

Instantiation 4 (Dead block – Anomaly 1)**.** Same as Instantiation 3.

Instantiation 5 (Expressing code–KCONFIG defects – Anomaly 19)**.** Same as Instantiation 3.

Instantiation 6 (Expressing configuration-implementation defects – Anomaly 15)**.** $\mathcal{V}$ corresponds to the minimum but sufficient set of constraints from the configuration space. Thus:

$$sat\left((b_i \leftrightarrow \mathcal{PC}(b_i)) \wedge \mathcal{V}\right) \Leftrightarrow sat\left(\mathcal{PC}_{Int}(b_i) \wedge \mathcal{C}_{\text{KCONFIG}}\right)$$

We can then express *dead* and *undead* configuration-implementation defects. Given $\mathcal{B}$ the set of blocks and $\mathcal{C} = \mathcal{C}_{\text{KCONFIG}}$:

$$\begin{aligned}
\neg sat\left((b_i \leftrightarrow \mathcal{PC}(b_i)) \wedge \mathcal{V}\right) &\Leftrightarrow \neg sat\left(\mathcal{PC}_{Int}(b_i) \wedge \mathcal{C}\right)\\
&\Leftrightarrow \neg sat\left(\mathcal{PC}_{Ext}(b_i)\right) \text{ (Definition 8.3)}\\
\neg sat\left(\neg\left(b_i \leftrightarrow \mathcal{PC}(b_i)\right) \wedge \mathcal{V}\right) &\Leftrightarrow \neg sat\left(\neg\mathcal{PC}_{Int}(b_i) \wedge \mathcal{C}\right)\\
&\Leftrightarrow \neg sat\left(\neg\mathcal{PC}_{Ext}(b_i)\right) \text{ (Definition 8.3)}
\end{aligned}$$

Anomaly 15 thus expresses dead (Definition 8.6) and core defects (Definition 8.7).

From Anomaly 15 we can derive Anomaly 14.

Instantiation 7 (Expressing Implementation-only defects – Anomaly 14)**.** Given $\mathcal{B}$ the set of blocks:

$$\begin{aligned}
\neg sat\left(b_i \leftrightarrow \mathcal{PC}_{Sin}(b_i)\right) &\Leftrightarrow \neg sat\left(\mathcal{PC}_{Int}(b_i)\right)\\
&\Leftrightarrow b_i \in deads(\mathcal{B}) \text{ (Definition 8.4)}\\
\neg sat\left(\neg\left(b_i \leftrightarrow \mathcal{PC}_{Sin}(b_i)\right)\right) &\Leftrightarrow \neg sat\left(\neg\left(\mathcal{PC}_{Int}(b_i)\right)\right)\\
&\Leftrightarrow b_i \in core(\mathcal{B}) \text{ (Definition 8.5)}
\end{aligned}$$

Additionally, Anomalies 14 and 15 are equivalent to Anomalies 2 and 3, respectively.

Instantiation 8 (Expressing Internal consistency – Anomaly 2)**.** $\bigwedge_{i=1..m} b_i \leftrightarrow \mathcal{PC}(b_i)$ corresponds to the set of constraints of the code space, and b_i the selection of the b_i block. Therefore, $\left(\bigwedge_{i=1..m} b_i \leftrightarrow \mathcal{PC}(b_i)\right) \wedge b_i$ can be simplified to $b_i \leftrightarrow \mathcal{PC}(b_i)$, as done by Tartler et al. [2011] in Anomaly 14. Thus, Anomaly 2 $\Leftrightarrow$ Anomaly 14.

Instantiation 9 (Expressing External consistency – Anomaly 3)**.** In Instantiation 8, we showed:

$$\left(\bigwedge_{i=1..m} b_i \leftrightarrow \mathcal{PC}(b_i)\right) \wedge b_i \Leftrightarrow (b_i \leftrightarrow \mathcal{PC}(b_i))$$

FM in Anomaly 3 and $\mathcal{V}$ in Anomaly 15 both represent the KCONFIG space constraints. Therefore:

$$satisfiable\left(\left(\bigwedge_{i=1..m} b_i \leftrightarrow \mathcal{PC}(b_i)\right) \wedge b_i \wedge FM\right) \Leftrightarrow sat\left((b_i \leftrightarrow \mathcal{PC}(b_i)) \wedge \mathcal{V}\right)$$

i.e., Anomaly 3 $\Leftrightarrow$ Anomaly 15.

Instantiation 10 (Expressing code anomalies – Anomaly 18)**.** Same as Instantiation 8.

Instantiation 11 (Expressing code–Make–KCONFIG missing defects – Anomaly 23)**.** We showed in Instantiation 1 that $B_N \wedge C \wedge M \wedge K \models \mathcal{PC}_{Ext}(B_N)$. If a feature m from the formula is not defined in the KCONFIG files, it means that $m \notin terms(K) \cup terms(M)$, *i.e.*, $m \notin terms(\mathcal{C}_{\text{KCONFIG}} \cup \mathcal{C}_{\text{KBUILD}})$. Therefore:

$$\exists m \in terms(\mathcal{PC}_{Ext}(B_N)) \mid (m \notin terms(\mathcal{C}))$$

thus B_N is dead by missing feature.

Instantiation 12 (Expressing code–KCONFIG missing defects – Anomaly 20)**.** Same as Instantiation 11 with $\mathcal{C} = \mathcal{C}_{\text{KCONFIG}}$.

Instantiation 13 (Expressing referential defects – Anomaly 17)**.** If the feature is missing in the configuration space, then the definition corresponds Definition 8.9 with $\mathcal{C} = \mathcal{C}_{\text{KCONFIG}}$ as context. A feature missing in the implementation space can mean that the feature is used in the Make space only. It is characterized as a defect as Tartler et al. [2011] do not consider this space, but it is not a defect for us.

8.2.2 Model on KBUILD

Instantiation 14 (Expressing Feature Not Defined – Anomaly 11)**.** Given m a feature not being defined in any KCONFIG files, and a a file referenced a KBUILD Makefile whose presence is conditioned by m. Thus, m is present in ϕ_{select_a}, however is not present in the features defined in the KCONFIG files, obtained with $terms(\mathcal{C}_{\text{KCONFIG}})$.

$$(m \in terms(\phi_{select_a})) \wedge (m \notin terms(\mathcal{C}_{\text{KCONFIG}}))$$

As $terms(\phi_{selecta_i}) \subseteq terms(\mathcal{PC}_{Ext}(a_i))$, Anomaly 11 is a special case of Definition 8.9, therefore a is a missing dead file.

Instantiation 15 (Expressing Variable Not Used – Anomaly 12)**.** Given a an asset and $\phi_{preds_a} = comp$.

$$\begin{aligned}\mathcal{PC}_{Int}(a) &= \phi_{select_a} \wedge expand\left(\phi_{preds_a}\right) \wedge expand\left(\phi_{depInt_a}\right) \text{ (Definition 8.2)}\\ &= \phi_{select_a} \wedge \mathcal{PC}_{Int}(comp) \wedge expand\left(\phi_{depInt_a}\right)\end{aligned}$$

However, $comp$ is never used, therefore $\phi_{select_{comp}}$ is $false$, implying $\neg\mathcal{PC}(comp)$. Consequently:

$$\begin{aligned}\mathcal{PC}_{Int}(a) &= \phi_{select_a} \wedge false \wedge expand\left(\phi_{depInt_a}\right)\\ &= false\end{aligned}$$

Thus, a is a dead asset.

Instantiation 16 (Expressing Make–KCONFIG anomalies – Anomaly 24)**.** Let us consider s the asset that represents the file F_N, and $\mathcal{C} = \mathcal{C}_{\text{KCONFIG}}$.

$$\begin{aligned}\mathcal{PC}_{Int}(s) &= \phi_{select_s} \wedge expand\left(\phi_{preds_s}\right) \wedge expand\left(\phi_{depInt_s}\right)\\ &= \phi_{select_s} \wedge \mathcal{PC}_{Int}(comp) \wedge \mathcal{PC}_{Int}(dir)\end{aligned}$$

To build M as it appears in Anomaly 24, Nadi and Holt [2012] extract for every file a presence condition consisting of a conjunction of the features conditioning the selection of the file $(\bigvee f_i = \phi_{select_s})$, the composite object if present $(\mathcal{PC}_{Int}(comp))$ and its parent directory $(\mathcal{PC}_{Int}(dir))$ in the corresponding Makefiles. Therefore, $\mathcal{PC}_{Int}(s) \models (F_N \wedge M)$.

$$\begin{aligned}\mathcal{PC}_{Ext}(s) &= \mathcal{PC}_{Int}(s) \wedge slice\left(\mathcal{C}, terms(\mathcal{PC}_{Int}(s)) \cup terms(\phi_{depExt_s})\right)\\ &\models (F_N \wedge M) \wedge K \text{ (\textit{cf.} Section 8.2.1.2)}\end{aligned}$$

We can then express Anomaly 24 in our model:

$$\begin{aligned}\neg sat(F_N \wedge M \wedge K) &\Leftrightarrow \neg sat\left(\mathcal{PC}_{Ext}(s)\right)\\ \neg sat(\neg F_N \wedge M \wedge K) &\Leftrightarrow \neg sat\left(\neg\mathcal{PC}_{Ext}(s)\right)\end{aligned}$$

Anomaly 24 thus expresses dead (Definition 8.6) and core defects (Definition 8.7).

Instantiation 17 (Expressing Make–KCONFIG missing defects – Anomaly 25)**.** Same as Instantiation 11 with $\mathcal{C} = \mathcal{C}_{\text{KCONFIG}}$, and relying on formulas from Instantiation 16.

Instantiation 18 (Expressing File Not Used – Anomaly 10)**.** This definition may not be valid anymore, since the syntax of KBUILD Makefiles allows them to explore subdirectories too[3]. However, we can generalise this definition with:

A .c file exists in the codebase but is not used in any Makefile.

Given S the set of source files of the Linux kernel code base, and $\mathcal{A}_{\text{KBUILD}}$ the set of assets representing source files in the KBUILD Makefiles. A file $s \in S$ is a file not used if no asset in $\mathcal{A}_{\text{KBUILD}}$ corresponds to s:

$$\nexists a_i \in \mathcal{A}_{\text{KBUILD}} \mid s \equiv a_i$$

[3] https://www.kernel.org/doc/html/latest/kbuild/makefiles.html#descending-down-in-directories

8.2.3 Model on KCONFIG

Given the configurator model and the application example already given in Section 8.1.2, we just have to instantiate the anomalies.

Instantiation 19 (Expressing dead feature – Anomaly 4)**.** Given F a dead feature. The definition can be expressed in our model as $\neg sat(\phi_{deps_F})$, which itself implies $\neg sat(\mathcal{PC}(F))$, hence F is dead.

Instantiation 20 (Expressing false optional – Anomaly 5)**.** This definition corresponds to the note in Definition 8.13, thus F is a core feature.

Instantiation 21 (Expressing missing dead feature – Anomaly 6)**.** The definition limits the presence of an undefined feature in the dependencies:

$$(m \in terms(\phi_{deps_F})) \wedge (m \notin \mathcal{F})$$

As $terms(\phi_{deps_F}) \subseteq terms(\mathcal{PC}(F))$, every missing dead feature according to Anomaly 6 is also missing dead in our model.

Instantiation 22 (Expressing selects on symbols with dependencies – Anomaly 7)**.** The dependencies of the symbol are represented by $terms(\phi_{deps_F})$. Its selection by another symbol is represented by $directSelect(F)$. Therefore:

$$(terms(\phi_{deps_F}) \neq \varnothing) \wedge directSelect(F)$$

Instantiation 23 (Expressing unreachable symbol – Anomaly 8)**.** Given F a symbol. If the symbol does not have a prompt neither a default value allowing its selection, then $\neg sat(\phi_{enable_F})$. Selection by another feature is modeled with $directSelect(F)$. Thus:

$$\neg sat(\phi_{enable_F}) \wedge \neg directSelect(F)$$

Consequently:

$$\begin{aligned}\mathcal{PC}(F) &= (\phi_{enable} \wedge expand(\phi_{deps})) \vee directSelect(F) \text{ (Definition 8.11)} \\ &= (false \wedge expand(\phi_{deps})) \vee false \\ &= false\end{aligned}$$

Thus, F is dead.

Instantiation 24 (Expressing unnecessary selects on choice values – Anomaly 9)**.** To express this defect, we need to add an extra $type$ attribute to the feature. $type \in \{config, choice\}$ represents the way F is defined in the KCONFIG model, either as a simple $config$ element or in a $choice$ statement.

$$(type_F = choice) \wedge directSelect(F)$$

Instantiation 25 (Expressing configuration-only defects – Anomaly 16)**.** The function $presenceCondition(feature)$ returns the presence implication of the feature and is defined by the authors as *"the selection of the feature itself and the expression of the depends on option."* This definition, expressed by our model, corresponds to $\phi_{enable_f} \wedge expand(\phi_{deps_f}) = \mathcal{PC}(f)$. Thus

$$\neg sat(f \rightarrow presenceCondition(f)) \Leftrightarrow \neg sat(\mathcal{PC}(f))$$

Therefore, f is dead.

Table 8.2: Anomalies covered by the model (defects defined as dead and undead according to the authors)

Paper			Sincero et al. [2010]	Tartler et al. [2011]	Nadi and Holt [2011]	Nadi and Holt [2012]	Hengelein [2015]
Derivator	Internal consistency	Dead	Anomaly 2	Anomaly 14	Anomaly 12	Anomaly 18	
		Core	Anomaly 2	Anomaly 14		Anomaly 18	
	External consistency	Dead	Anomalies 1 and 3	Anomalies 13 and 15		Anomalies 19, 21, 22 and 24	
		Core	Anomaly 3	Anomaly 15		Anomaly 24	
		Full-mandatory		Anomaly 13		Anomalies 19, 21 and 22	
	Missing feature			Anomaly 17	Anomaly 11	Anomalies 20, 23 and 25	
Configurator	Dead			Anomaly 16			Anomaly 4
	Core						Anomaly 5
	Missing dead						Anomaly 6
Other properties (*e.g.*, unreachable symbol, file not used)					Anomaly 10		Anomalies 7 to 9

8.2.4 Resulting coverage

From the instantiations of the configurator model on KCONFIG and the derivator on both KBUILD and CPP, we obtain a complete expression of the different formulas and anomalies taken as input. A summary of the different anomalies for each paper and how they are expressed is presented in Table 8.2. As expected, no existing proposal expresses defects in every space of the Linux build system. The table confirms the inconsistencies that we manually observed in Section 3.5 between Anomalies 13 and 15 from [Tartler et al., 2011], with the first anomaly being characterized as a *full-mandatory* defect and the other as a *core* defect. Moreover, similar inconsistencies are exhibited in defects from [Nadi and Holt, 2012], as well as two anomalies that are described as *dead* defects but are not called as such (Anomalies 12 and 16). Additionally, the obtained presence conditions allow a better understanding at fine grain of the interactions between the assets. Table 8.3 details, for an asset of each space taken from Figure 2.2, the expressions obtained when considering different spaces of constraints using formalisms from the state of the art and our representation. The expressions of the $\mathcal{PC}$s detail with precision the different assets involved, demonstrating the capacity of our model to represent at fine-grain the interactions between assets in the build system.

8.3 Threats to validity

Internal threats to validity A first internal threat could be caused by the selection of papers made to devise the properties and the model. However, given the narrow focus of the subject and the fact that we sought additional work in the references of the obtained papers, we believe that the most important studies are included.

Another internal threat concerns the accuracy of the formalisms from the chosen studies, but their application to the kernel was demonstrated. Besides, while we provide a formalization, there

Table 8.3: Expressions of $\mathcal{PC}$s for the `B3` block, the `foo.c` file and the `FOO` feature from Figure 2.2.

Involved space(s)	**SOTA expressions** **Proposed models expressions**
CPP	$Block_N \wedge C$ [Sincero et al., 2010] $\mathcal{PC}_{Int}(\texttt{B3}) = \texttt{BAR} \wedge \neg\mathcal{PC}_{Int}(\texttt{B2}) \wedge \mathcal{PC}_{Int}(\texttt{B1})$
CPP – KBUILD	$Block_N \wedge C \wedge M$ [Nadi and Holt, 2012] $\mathcal{PC}_{Ext}(\texttt{B3}) = \mathcal{PC}_{Int}(\texttt{B3}) \wedge \mathcal{PC}_{Ext}(\texttt{foo.c})$
CPP – KCONFIG	$Block_N \wedge C \wedge K$ [Sincero et al., 2010], $\mathcal{V} \wedge Block_N$ [Tartler et al., 2011] $\mathcal{PC}_{Ext}(\texttt{B3}) = \mathcal{PC}_{Int}(\texttt{B3}) \wedge \mathcal{PC}(\texttt{BAR}) \wedge \neg\mathcal{PC}(\texttt{FOO}) \wedge (\mathcal{PC}(\texttt{FOO}) \vee \mathcal{PC}(\texttt{BAR}) \vee \mathcal{PC}(\texttt{BAZ}))$
CPP – KBUILD – KCONFIG	$Block_N \wedge C \wedge M \wedge K$ [Nadi and Holt, 2012] $\mathcal{PC}_{Ext}(\texttt{B3}) = \mathcal{PC}_{Int}(\texttt{B3}) \wedge \mathcal{PC}_{Ext}(\texttt{foo.c}) \wedge \mathcal{PC}(\texttt{BAR}) \wedge \neg\mathcal{PC}(\texttt{FOO})$*
KBUILD	$File_N \wedge M$ [Nadi and Holt, 2012] $\mathcal{PC}_{Int}(\texttt{foo.c}) = \texttt{FOO} \wedge \mathcal{PC}_{Int}(\texttt{dir})$
KBUILD – KCONFIG	$File_N \wedge M \wedge K$ [Nadi and Holt, 2012] $\mathcal{PC}_{Ext}(\texttt{foo.c}) = \mathcal{PC}_{Int}(\texttt{foo.c}) \wedge \mathcal{PC}(\texttt{FOO}) \wedge \mathcal{PC}(\texttt{BAR})$
KCONFIG	$presenceCondition(feature) = feature \rightarrow \varphi$ [Tartler et al., 2011] $\mathcal{PC}(\texttt{FOO}) = \sigma_{FOO} \wedge (\mathcal{PC}(\texttt{DEPS\_A}) \vee \mathcal{PC}(\texttt{DEPS\_B})) \wedge \mathcal{PC}(\texttt{MENU\_COND})$

* for readibility, the conjunction with $(\mathcal{PC}(\texttt{FOO}) \vee \mathcal{PC}(\texttt{BAR}) \vee \mathcal{PC}(\texttt{BAZ}))$ has been omitted as it is redundant in the condition

is no proof of correctness of the formalism nor associated syntax and semantics given in a theorem prover.

External threats to validity While we show here the application of the proposed models to the Linux build system, there is no demonstration of the applicability of the configurator and derivator concepts to other build systems. A first demonstration of the derivator concept is nevertheless done through its double instantiation as the KCONFIG and CPP preprocessor.

8.4 Conclusion

Studies on the kernel variability propose a set of definitions for the analyzed properties. However, they suffer from differences in terminology and some inconsistencies in the interpretation of similar definitions in them (*cf.* Sections 3.4 and 3.5). In this chapter, we described a formalism based on the generic concepts of configurator and derivator to express the whole set of consistency properties. We showed that the configurator can be instantiated to represent the KCONFIG, while the instantiated derivators can represent the KBUILD, selecting files, or CPP, selecting code blocks. The obtained model enables one to categorize the previous studies and to establish their coverage and divergences on the analyses. Relying on generic concepts, the model is designed to be applicable to other build systems relying on the concepts of configurator and derivator.

This unified representation at fine grain of the interactions between assets throughout the build system provides elements of answers to Challenge B1. Additionally, anomalies are characterized in this formalism, covering and aligning state-of-the-art-definitions, answering Challenge B2 on the characterization and identification of anomalies in build systems. However, this work exhibits

two limitations. First, as the existing work on build systems and their anomalies are focused on the Linux build system, we can only assess relevance of our representation on this system, and it may therefore not represent the diversity of mechanisms present in other build systems such as BusyBox [Kästner et al., 2012; Mordahl et al., 2019], JHipster [Halin et al., 2017], or the Mozilla build system [Maudoux and Mens, 2019]. Then, the proposed models have not been implemented in a model-driven framework and applied on real code assets, limiting their validity. Fully answering challenges B1 and B2 requires achieving those two steps, that we tackle in the following chapter.

Chapter 9

A generalization of the anomalies model

When build systems combine ad hoc tools adapted to manage variability, the multiplicity and entanglement of these mechanisms do not provide a global view of the whole derivation process of the build system, preventing precise determination of the constraints that condition the selection of each code asset (Challenge B1). Additionally, such tools not being designed for variability management, they do not allow checking for the consistency of the implemented constraints, potentially leading to conflicts and causing anomalies (Challenge B2).

In the previous chapter, we started tackling both challenges by providing a homogeneous representation of the Linux kernel build system derivation mechanism and the anomalies that can occur, showing coverage of existing contributions on both topics. While the proposed model shows coverage of existing work characterizing the derivation process of build systems and their anomalies, they are all focused on the Linux kernel build system. Consequently, to assess whether the model is applicable to other build systems and brings a definitive answer to Challenges B1 and B2, we propose to extend it to another build system, the Mozilla build system. We believe this build system is a good candidate for generalizing our representation as it reuses off-the-shelf tools (namely, Python and CPP) to implement a derivation process similar to the one of the Linux kernel build system (*cf.* Section 2.3.2). Moreover, the Mozilla build system is used to manage the variability implemented in the Mozilla Gecko codebase, of which more than 25%[1] is written in an OO language, C++, consequently exhibiting OO variability implementations (Section 4.1). Therefore, being able to model the Mozilla build system's derivation process is a first step towards understanding how the coexistence of those two types of variability is managed in such systems.

In Section 9.1, we detail the current limitations of the proposed model that need to be tackled to enable its application on the Mozilla build system. We then generalize the model by taking into account the diversity of mechanisms brought by the Mozilla build system and consequently refine the definitions of $\mathcal{PC}$s and anomalies (Section 9.2). Finally, we design a framework implementing our model and report on its application on the Mozilla build system and two systems using the Linux kernel build system, the Linux kernel and BusyBox[2], for which we compare our approach with a state-of-the-art workbench for analyzing the Linux kernel build system, KernelHaven (Section 9.3).

[1] https://www.openhub.net/p/firefox/analyses/latest/languages_summary
[2] https://busybox.net

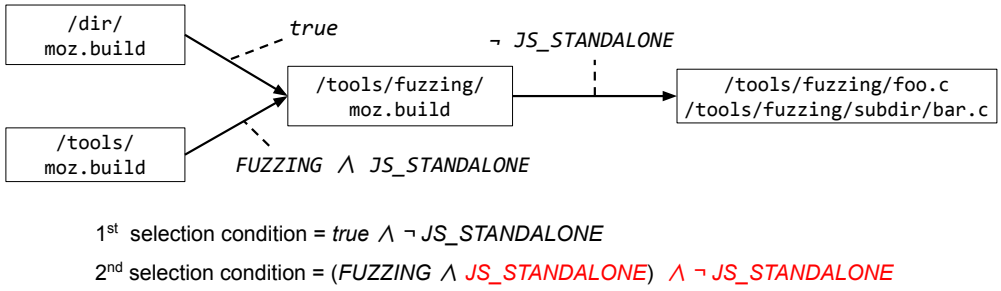


Figure 9.1: Visual representation of the inclusions shown in the MOZBUILD space on Figure 2.3.

9.1 Limitations of the anomalies model related to the Mozilla build system

While the model previously introduced covers existing work on characterizing anomalies in the Linux kernel build system, it exhibits some limitations preventing its direct application to the Mozilla build system.

Arity of parent assets Each asset from a derivator possesses a single internal $\mathcal{PC}$ ($\mathcal{PC}_{Int}$) and a single external $\mathcal{PC}$ ($\mathcal{PC}_{Ext}$), implying that each asset has a single inclusion chain allowing its selection. This is indeed the case for both derivators in the Linux kernel build system:

- in CPP, the selection of a code block is dependent on the selection of its direct parent block. Each block naturally has only one direct parent block.
- in KBUILD, each Makefile *"is only responsible for building objects in its own directory"*[3]. Therefore, each file can only be included by the Makefile in its own directory.

The Mozilla build system's MOZBUILD, however, allows more freedom in the selection of its source files as, as detailed in Section 2.3.2, a `moz.build` file can be included by multiple other ones. It is hence possible to have multiple inclusion chains leading to a single source file's selection (*i.e.*, multiple $\mathcal{PC}$s), each one of them potentially exhibiting an anomaly. For example, on Figure 9.1, `/tools/fuzzing/foo.c` has two inclusion chains, of which one is defectuous. Using conjunctions or disjunctions to assemble them as a single $\mathcal{PC}$ would lead to major clarity issues. In this same example, $\mathcal{PC}_1 \vee \mathcal{PC}_2$ would be satisfiable and not exhibit any defect, hiding the defectuous inclusion chain. On the opposite, although $\mathcal{PC}_1 \wedge \mathcal{PC}_2$ would not be satisfiable and exhibit a dead defect, the information on which inclusion is defectuous would be lost. In order to maximize the expressiveness of our representation of the build system, there is a need to update the representation of how assets include each other. Consequently, as the expressions of anomalies depend on this representation, they may also need to be updated.

Particularities of the configuration space In this model, the expression of an asset's $\mathcal{PC}_{Ext}$ depends on the external spaces of constraints that we choose to include, indifferently being derivators or configurators. Although this mechanism allows an easy combination of external sets of

[3] https://www.kernel.org/doc/Documentation/kbuild/makefiles.txt

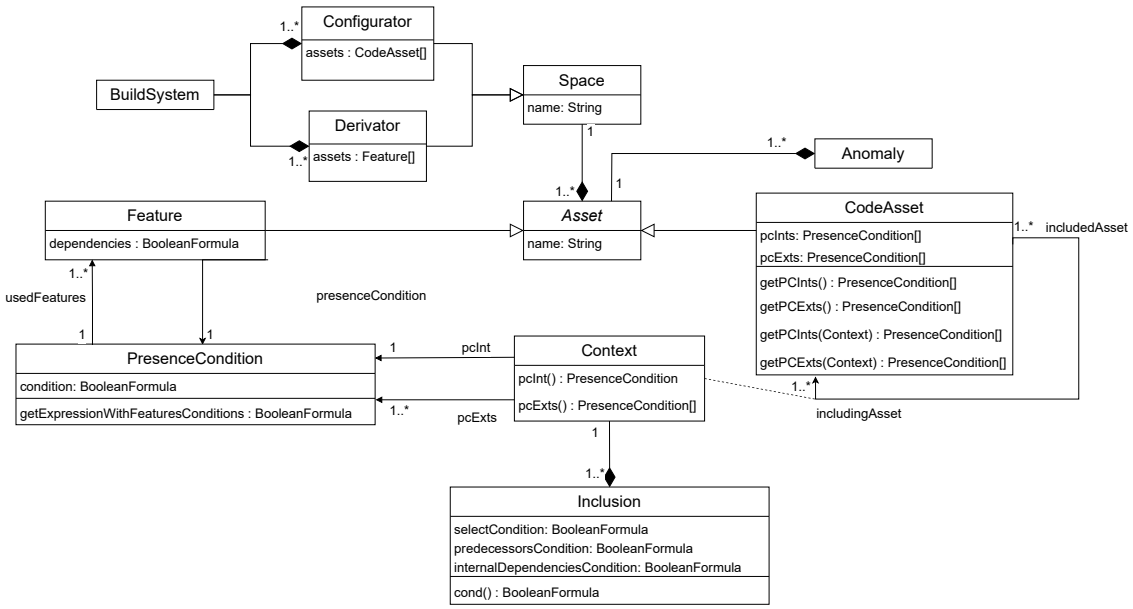


Figure 9.2: Assets model

constraints, it can potentially hamper the comprehension of an identified anomaly. For example, when analyzing the consistency of a code block considering constraints from all spaces, the $\mathcal{PC}_{Ext}$ checks the consistency of the block's condition with the condition to select its containing file in the KBUILD and the conditions to select the features of the conditions in the KCONFIG. However, if $\mathcal{PC}_{Ext}$ is not satisfiable, understanding the real cause is not straightforward as it could be due to:

- an incompatibility between the condition to select the block and the condition to select the file;
- an incompatibility between the condition to select the block and the conditions to select the features from this condition;
- an incompatibility between the file's condition and the conditions to select the features from this condition.

This limitation calls for a need to consider differently external derivation spaces and configuration spaces.

9.2 Generalizing the model

9.2.1 Dealing with arity of parent assets

Source files being derivator assets in our model, representing the selection of a source file by multiple `moz.build` files means that we need to consider in our model the selection of an asset by multiple others. Additionally, an asset can be included more than once by another one under different conditions, as shown on Listing 9.1. `android_support` is included twice by the same `moz.build` file, under both the `CPU_ARCH__arm__` $\wedge$ `OS_TARGET__Android__` and

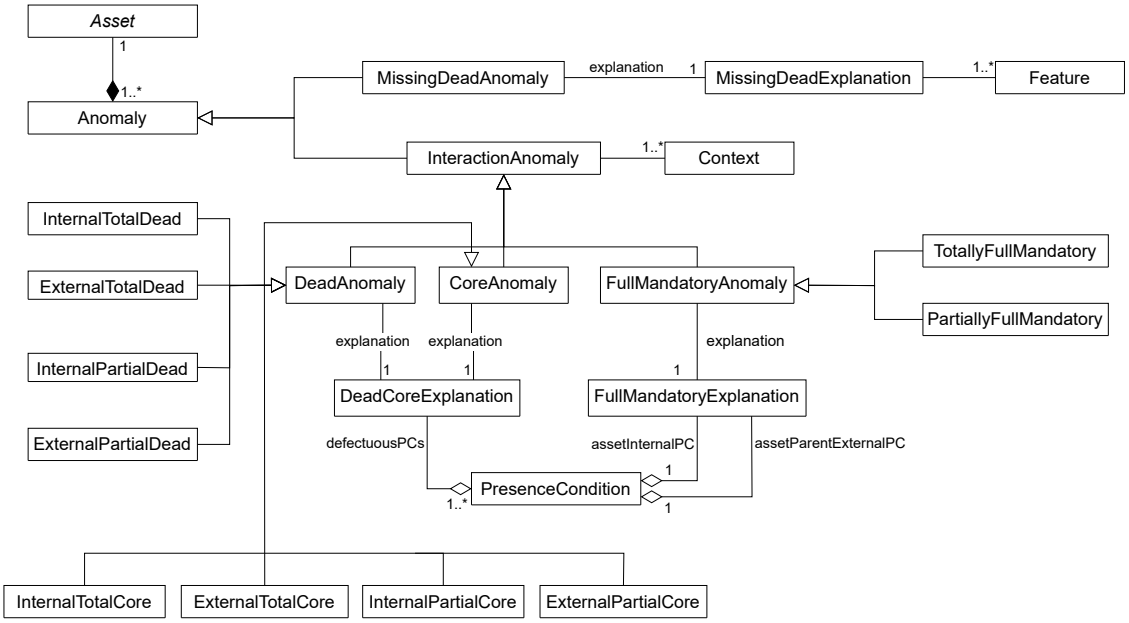


Figure 9.3: Anomalies model

`CPU_ARCH__x86__` $\wedge$ `OS_TARGET__Android__` conditions. Considering these new specificities leads to an updated representation of assets in the build system presented in Figure 9.2. A **Context** represents the inclusion of an asset by another, having one **Inclusion** for each condition that allows the asset's selection by this other asset. While having multiple inclusions in a context allows representing the different conditions allowing an asset to select another, the different contexts of an asset allow modeling its selection by multiple different assets.

Formally, a context $c = \langle a_{child}, a_{parent}, I \rangle$ can be defined as follows:

- a_{child} is the selected code asset ;
- a_{parent} is the selecting code asset ;
- I a set of inclusions.

An inclusion $i \in I$ is defined as $i = \langle \phi_{select}, \phi_{preds}, \phi_{depInt} \rangle$, whose definitions are identical as in the definition of asset in the previously presented model Chapter 8, Definition 8.1. Therefore, the $\mathcal{PC}_{Int}$ is now linked to an inclusion instead of an asset. Since an asset is selected by another when at least one of its inclusions is satisfiable, a context's $\mathcal{PC}_{Int}$ is the disjunction of its inclusions' conditions $cond$.

$$\mathcal{PC}_{Int}(c) = \bigvee_{i \in I} cond(i)$$

with $cond(i) = \phi_{select_i} \wedge expand\,(\phi_{preds_i}) \wedge expand\,(\phi_{depInt_i})$.

For each context, we can also define the $\mathcal{PC}_{Exts}$ issuing from inclusions in a context.

$$\mathcal{PC}_{Exts}(c) = \bigcup_{pc \in \mathcal{PC}_{Exts}(a_{parent})} \mathcal{PC}_{Int}(c) \wedge pc$$

Listing 9.1: Excerpt of resource_adaptation_api_gn/moz.build

```
168 if CONFIG["CPU_ARCH"] == "arm" and CONFIG["OS_TARGET"] == "Android":
169
170     OS_LIBS += [
171         "android_support",
172         "unwind"
173     ]
174
175 if CONFIG["CPU_ARCH"] == "x86" and CONFIG["OS_TARGET"] == "Android":
176
177     OS_LIBS += [
178         "android_support"
179     ]
```

Finally, an asset having now a list of contexts C, they possess several $\mathcal{PC}_{Ints}$ and $\mathcal{PC}_{Exts}$.

$$\mathcal{PC}_{Ints}(a) = \bigcup_{c \in C} \mathcal{PC}_{Int}(c) \qquad \mathcal{PC}_{Exts}(a) = \bigcup_{c \in C} \mathcal{PC}_{Exts}(c)$$

This additional decomposition leads to a refinement of dead, core and full-mandatory assets definitions shown in Figure 9.3:

- **partially dead assets**, for which some but not all $\mathcal{PC}$s are satisfiable:

$$\exists pc \in \mathcal{PC}_{Ints/Exts}(a) \quad \neg sat(pc)$$

- **totally dead assets**, for which no $\mathcal{PC}$ is satisfiable:

$$\forall pc \in \mathcal{PC}_{Ints/Exts}(a) \quad \neg sat(pc)$$

- **partially core assets**, for which some but not all $\mathcal{PC}$s are always satisfiable:

$$\exists pc \in \mathcal{PC}_{Ints/Exts}(a) \quad \neg sat(\neg pc)$$

- **totally core assets**, for which all $\mathcal{PC}$s are always satisfiable:

$$\forall pc \in \mathcal{PC}_{Ints/Exts}(a) \quad \neg sat(\neg pc)$$

- **partially full-mandatory assets**, for which the constraints from the external space imply its selection for some inclusion chains:

$$\exists pc_{Int} \in \mathcal{PC}_{Ints}(a), pc_{Ext} \in \mathcal{PC}_{Exts}(a) \quad sat(pc_{Ext}) \wedge \neg sat(\neg pc_{Int} \wedge pc_{Ext})$$

- **totally full-mandatory assets**, for which the constraints from the external space imply its selection for all inclusion chains:

$$\forall pc_{Int} \in \mathcal{PC}_{Ints}(a), pc_{Ext} \in \mathcal{PC}_{Exts}(a) \quad sat(pc_{Ext}) \wedge \neg sat(\neg pc_{Int} \wedge pc_{Ext})$$

Previously defined *dead*, *core* and *full-mandatory* anomalies can still be identified as they are special cases of *totally dead*, *totally core* and *totally full-mandatory* anomalies when there is a single inclusion chain.

Note. For dead and core anomalies, $\mathcal{PC}_{Ints}$ are used to check for internal anomalies (Section 8.1.1.1) and $\mathcal{PC}_{Exts}$ to check the for external anomalies (Section 8.1.1.2).

Table 9.1: New expressions of $\mathcal{PC}$s for the `B3` block and the `foo.c` file involving the configuration space from Figure 2.2.

Asset	Expressions
`B3` block	$\mathcal{PC}_{Int}(\texttt{B3}) = \texttt{BAR} \wedge \neg\mathcal{PC}_{Int}(\texttt{B2}) \wedge \mathcal{PC}_{Int}(\texttt{B1})$ $\mathcal{PC}_{Ext}(\texttt{B3}) = \mathcal{PC}_{Int}(\texttt{B3}) \wedge \mathcal{PC}_{Ext}(\texttt{foo.c})$ $\mathcal{PC}_{Int+config}(\texttt{B3}) = \mathcal{PC}_{Int}(\texttt{B3}) \wedge \mathcal{PC}(\texttt{BAR}) \wedge \neg\mathcal{PC}(\texttt{FOO}) \wedge (\mathcal{PC}(\texttt{FOO}) \vee \mathcal{PC}(\texttt{BAR}) \vee \mathcal{PC}(\texttt{BAZ}))$ $\mathcal{PC}_{Ext+config}(\texttt{B3}) = \mathcal{PC}_{Int+config}(\texttt{B3}) \wedge \mathcal{PC}_{Int+config}(\texttt{foo.c})$*
`foo.c` file	$\mathcal{PC}_{Int}(\texttt{foo.c}) = \texttt{FOO} \wedge \mathcal{PC}_{Int}(\texttt{dir}) = \texttt{FOO} \wedge \texttt{BAR}$ $\mathcal{PC}_{Int+config}(\texttt{foo.c}) = \mathcal{PC}_{Int}(\texttt{foo.c}) \wedge (\mathcal{PC}(\texttt{FOO}) \wedge \mathcal{PC}(\texttt{BAR}))$

* in this case, while $\mathcal{PC}_{Ext+config}(\texttt{foo.c})$ should be used, it is equivalent to $\mathcal{PC}_{Int+config}(\texttt{foo.c})$ as assets in the KBUILD space do not depend on any asset from another derivator.

9.2.2 Defining $\mathcal{PC}$s with feature constraints

In order to consider differently external derivation spaces and the configuration space when checking for anomalies in a derivation space, we refine the already existing $\mathcal{PC}_{Ints}$ and $\mathcal{PC}_{Exts}$ and define variants including the constraints for the features in the $\mathcal{PC}$, $\mathcal{PC}_{Int+config}$ and $\mathcal{PC}_{Ext+config}$. $\mathcal{PC}_{Ints}$ and $\mathcal{PC}_{Exts}$ now consider only the desired derivation spaces, while $\mathcal{PC}_{Int+config}$ and $\mathcal{PC}_{Ext+config}$ add constraints from the configuration space. This allows a finer granularity in the identification of anomalies:

- an incompatibility between the condition to select the block and the condition to select the file is identified using the $\mathcal{PC}_{Ext}$;
- an incompatibility between the condition to select the block and the conditions to select the features from this condition is identified using the $\mathcal{PC}_{Int+config}$;
- an incompatibility between the file's condition and the conditions to select the features from this condition is identified using the $\mathcal{PC}_{Ext+config}$.

To better understand how these $\mathcal{PC}$s are used in practice, we detail in Table 9.1 their expressions for the `B3` block and the `foo.c` file from Figure 2.2. This new decomposition of $\mathcal{PC}$s allows a better understanding of the considered spaces. For example, $\mathcal{PC}_{Ext}(\texttt{B3})$ is defined two times in Table 8.3, first for CPP – KBUILD constraints, and then for CPP – KBUILD – KCONFIG constraints. This double use of the same denomination is confusing as it takes into account constraints for `B3` and `foo.c` in the first case, and in the second case constraints for `B3`, `foo.c` and the features selecting `B3`. Our new representation (Figure 2.2) allows representing this second case formally under the denomination of $\mathcal{PC}_{Int+config}(\texttt{B3})$. It also enables the definition of $\mathcal{PC}_{Ext+config}(\texttt{B3})$, considering constraints for `B3`, `foo.c` and the features selecting both `B3` and `foo.c`. We instantiate on both the Linux kernel build system and the Mozilla build system the proposed models of configurator and derivator in Tables 9.2 and 9.3, respectively. Mozilla's Autoconf differs from the KCONFIG by not allowing a user to select a set of features to configure the system. Being a set of web-based applications, the configuration of the Mozilla products is mostly related to the operating system and hardware specifications of the system running them, which are therefore determined automatically. Consequently, ϕ_{enable} is considered as being always $true$. Regarding the instantiation of the derivator, we observe that integrating the MOZBUILD led to the creation

Table 9.2: Configurator instantiated on the Linux kernel build system and the Mozilla build system.

Configurator	**KCONFIG**	**Mozilla Autoconf**
Asset	Feature	
ϕ_{enable}	the ability to select the feature by user selection (`prompt`), or `default` value	$true$
ϕ_{deps}	boolean formula on features defined in the `depends on` statement	boolean formula on features
$\mathcal{F}_{select}$	set of features selecting F with a `select` statement	selection by Autoconf system analysis
Core anomaly	the feature is always selected despite having constraints on its selection	
Dead anomaly	the feature can never be selected	
Missing dead anomaly	the feature depends on another feature that is not defined	

of *partial* anomalies, and that the previously *dead*, *core* and *full-mandatory* anomalies for the KBUILD and CPP are now *totally dead*, *totally core* and *totally full-mandatory* without changing their definitions.

9.3 Evaluation

To evaluate the relevance of our anomalies identification framework, we provide a twofold evaluation. First, we evaluate the accuracy of our representation by identifying dead anomalies in systems built using the Linux kernel build system, and compare our findings to the results of a similar analysis with the actual reference tool to identify anomalies in the Linux kernel build system, KernelHaven. Then, we evaluate the generalization capabilities by instantiating our model on the Mozilla build system to identify anomalies in the Gecko codebase (*cf.* Section 2.3.2).

9.3.1 Instantiation on open source systems using the Linux kernel build system

9.3.1.1 Subject systems

Table 9.4 depicts the studied subject systems. As our approach has been initially designed as a synthesis of studies on anomalies in the **Linux kernel**, we apply our tooled approach on this system to evaluate in practice the accuracy of our representation that, in a first step, has been done only formally (*cf.* Section 8.2.4). We selected **BusyBox** to evaluate the applicability of our approach on another system as its build system also uses KCONFIG, KBUILD and CPP, making it a reference case study that has been widely used by work studying anomalies in the Linux kernel build system [Dietrich et al., 2012; Gazzillo, 2017; Oh et al., 2019; Nguyen and Nguyen, 2020].

9.3.1.2 Evaluation process

For each project, we first run KernelHaven's UnDead analyzer[4] on the codebase and obtain the list of all dead code blocks and source files, together with the cause of the anomaly. The 4.4 version

[4] https://github.com/KernelHaven/UnDeadAnalyzer

Table 9.3: Derivator instantiated on the KBUILD, MOZBUILD and CPP.

Derivator	**KBUILD**	**MOZBUILD**	**CPP**
Asset	Source file	Source file	Code block
ϕ_{select_a}	$\bigvee f_i$ with f_i being features which at least one needs to be set for the source file to be selected. If the file is always selected, $\phi_{select} = true$. If the asset is defined but never added to a list, $\phi_{select} = false$.	The condition of the `if` surrounding the block. If the file is always selected, $\phi_{select} = true$.	The condition of the `#if` surrounding the block.
ϕ_{preds_a}	$comp$, the composite variable if s is part of a composite definition. $comp$ must be selected.	$\neg(\bigvee_i b_i)$ with b_i the corresponding `if` block and the potential `elif` blocks before b if b is an `elif` or `else` block, otherwise $true$.	$\neg(\bigvee_i b_i)$ with b_i the corresponding `#if` block and the potential `#elsif` blocks before b if b is an `#elsif` or `#else` block, otherwise $true$.
ϕ_{depInt_a}	dir, the directory containing the source file which also needs to be selected.	$p \wedge dir$ with p the parent block of b if b is a nested block and dir the condition to enter this MOZBUILD file.	p, the parent block of b if b is a nested block.
ϕ_{depExt_a}	–	–	The condition to select the file containing b.
Totally dead anomaly	the KBUILD inclusion chain can never select the file	no MOZBUILD inclusion chain allows to select the file	the code block can never be selected
Partially dead anomaly	–	some but not all MOZBUILD inclusion chains never allow selection of the file	–
Totally core anomaly	the KBUILD inclusion chain always selects the file	all MOZBUILD inclusion chains always select the file	the code block is always selected
Partially core anomaly	–	some but not all MOZBUILD inclusion chains always select the file	–
Totally full-mandatory anomaly	the file can be selected regarding constraints from the KBUILD file he's declared in, but cannot be when adding constraints from the parent KBUILD files	for all inclusion chains, the file can be selected regarding constraints from the MOZBUILD file he's declared in, but cannot be when adding constraints from the parent MOZBUILD files	the code block can be selected regarding CPP constraints, but cannot be when adding constraints from the MOZBUILD space
Partially full-mandatory anomaly	–	for some but not all inclusion chains, the file can be selected regarding constraints from the MOZBUILD file he's declared in, but cannot be when adding constraints from the parent MOZBUILD files	–

Table 9.4: Studied subject systems

System	Linux Kernel	BusyBox	Mozilla Gecko
Version	4.4	1.35.0	BETA_103_BASE
# LoC	14,336,580	200,802	27,470,299 including `comm`: 1,545,331
Configurator *# features*	KCONFIG 19,031	KCONFIG 1,257	Custom Autoconf 134*
Derivator on source files *# source files*	KBUILD 15,865	KBUILD 544	MOZBUILD 20,169
Derivator on code blocks *# code blocks (ifdefs)*	CPP 134,008	CPP 2,935	CPP 105,689

* manually extracted

Table 9.5: Identified anomalies correspondence table

Space	Proposed framework	KernelHaven
CPP	InternalTotalDead	C-preprocessor condition alone is not satisfiable
	ExternalTotalDead	C-preprocessor condition combined with file PC is not satisfiable
KBUILD	InternalTotalDead	File PC alone is not satisfiable

has been chosen as KernelHaven is preconfigured for this version, thus reducing the risks of potential inaccuracy in the results due to a configuration issue. Then, we extract the presence conditions for each code block and source file and use them as input for our anomalies identification framework. To obtain them, we run two extractors bundled with KernelHaven: the UndertakerExtractor[5] and the KbuildMinerExtractor[6]. These two extractors are based on the original implementations of UNDERTAKER [Sincero et al., 2010] and KbuildMiner [Berger and Kästner, 2016]. Not only do they implement the approaches we base our model on and allow us to extract information compliant with it, but they also allow us to reuse the same data as KernelHaven's UnDeadAnalyzer phase, ensuring maximum relevance of our comparison.

We then build a mapping adapting precision and recall measures, using the anomalies identified by KernelHaven as the ground truth, to evaluate the accuracy of our representation.

Precision is used to measure the percentage of anomalies identified by the proposed framework also being identified by KernelHaven;

Recall is used to measure the percentage of anomalies identified by KernelHaven also being identified by the proposed framework.

[5] https://github.com/KernelHaven/UndertakerExtractor
[6] https://github.com/KernelHaven/KbuildMinerExtractor

Table 9.6: Results of the mapping between anomalies identified by KernelHaven and our framework.

Tool	Linux anomalies			BusyBox anomalies		
	CPP	CPP – KBUILD	KBUILD	CPP	CPP – KBUILD	KBUILD
Proposed framework	2,441	40	8	154	11	15
KernelHaven	1,214	12	8	121	0	15
Both	1,214	12	8	121	0	15
Precision	49.73%	30%	100%	78.57%	– *	100%
Recall	100%	100%	100%	100%	– *	100%

* precision and recall values are not calculated for CPP – KBUILD anomalies in BusyBox as they are all false positives according to the mapping.

As KernelHaven's UnDead analyzer identifies dead blocks but not core ones, we consider only the dead anomalies identified by the proposed framework for the mapping, and normalize the anomalies denominations as summarized in Table 9.5. Additionally, although the proposed framework allows taking into account constraints from the configuration space, KernelHaven's only variability-model extractor KConfigReaderExtractor[7] extracts constraints in the form of a feature model represented by a single CNF formula. Consequently, the coarse grain expressiveness of provided by KernelHaven regarding the KCONFIG does not allow us to consider it in our fine-grain representation.

9.3.1.3 Observations

Table 9.6 summarizes the obtained results on both systems.

Results on the Linux kernel 49.73% of CPP anomalies identified by the proposed framework are also identified by KernelHaven. After manual analysis of the results, out of the 1.227 additional dead blocks identified only by our model, 396 blocks are present in header files that are not analyzed individually by KernelHaven. 2 are dead blocks in dead files considering the KBUILD space of constraints, and 166 are located in dead files considering both KBUILD and KCONFIG spaces, and are automatically ignored by KernelHaven. Similarly, 307 dead blocks are located in files for which KernelHaven could not determine the presence condition. This is due either to a KBUILD file that is not parseable by KernelHaven, or because the file is not included in the KBUILD file of their directory, or is in a directory not included by their parent KBUILD. This pattern actually characterizes a *file not used* anomaly [Nadi and Holt, 2011] (Anomaly 10). Blocks in such files are ignored by KernelHaven. An important majority of the remaining 388 blocks correspond to `#if 0` blocks or `#else` branches of `#if 1` blocks, while the others are likely to be anomalies revealed by the expansion of macros by KernelHaven. We do not cover this operation in our framework as we designed it to be independent of the implementation language. Consequently, we solely rely on the conditions extracted by the tool we use, PILZTAKER (Section 9.3.3). Regarding the CPP – KBUILD anomalies, all 28 anomalies identified only by the proposed framework are

[7] https://github.com/KernelHaven/KconfigReaderExtractor

code blocks in dead files, and thus are ignored by KernelHaven. Regarding the KBUILD anomalies, the proposed framework and KernelHaven identify exactly the same anomalies.

Results on BusyBox Regarding the CPP anomalies, the proposed framework identifies all 121 anomalies identified by KernelHaven. 33 additional anomalies are identified and are located in dead files ignored by KernelHaven. As for the Linux kernel, the identified KBUILD anomalies are identical between both approaches, and the 11 CPP – KBUILD anomalies not identified by KernelHaven are in dead files ignored by KernelHaven. It results that our representation covers all the anomalies identified by KernelHaven, as well as additional ones that have been ignored.

Summary It results that for both subject systems, the proposed framework identifies all anomalies identified by KernelHaven. Additionally identified anomalies correspond to actual anomalies ignored by KernelHaven due to their nature, for example, dead blocks in already dead files or `#if 0` blocks that are very likely to be intentional. Our framework therefore provides good coverage for these categories of anomalies. However, we could not consider the constraints from the configuration space, therefore we cannot conclude on the capacity of our representation to identify anomalies involving it. In a further step, we aim to explore the use of slicing techniques to obtain the constraints for each feature [Acher et al., 2011; Krieter et al., 2016] and represent them in our model.

9.3.2 Application to Mozilla Gecko

9.3.2.1 Subject system

For this evaluation, we consider the whole Mozilla Gecko codebase, namely the `mozilla-central`[8] repository containing code for the Firefox web browser, the Android Firefox application and the SpiderMonkey suite and `comm-central`[9] being the code specific to the Thunderbird mail client and the SeaMonkey suite. We analyze the `BETA_103_BASE` release, being the first release of the 103 version of the codebase, the most recent at the time of this study.

9.3.2.2 Evaluation process

After assembling corresponding versions of both codebases (tag `FIREFOX_BETA_103_BASE` for `mozilla-central` and `BETA_103_BASE` for `comm-central`), we automatically extract presence conditions from the CPP and MOZBUILD spaces. For each of the products that can be built from the codebase (*cf.* Section 2.3.2), the corresponding MOZBUILD files are parsed and their constraints extracted. As features constraints are challenging to parse due to their implementation and their scattering over files (*cf.* Section 2.3.2), we manually extracted a subset of features from `build/moz.configure/init.configure`[10]. In this section, we present the results obtained with MOZBUILD constraints from build files parsed when building the Firefox web browser.

[8] https://hg.mozilla.org/mozilla-central/

[9] https://hg.mozilla.org/comm-central/

[10] https://hg.mozilla.org/mozilla-central/file/a66dcaea419641c5483a43aa7f577b70908d147f/build/moz.configure/init.configure#l503

Table 9.7: Dead, core and full-mandatory anomalies identified in the Gecko codebase when building the Firefox browser

Spaces	Dead		Core		Full-mandatory	
	Totally	Partially	Totally	Partially	Totally	Partially
Without features from configuration						
CPP	533	–	2	–	–	–
MOZBUILD	0	59	15	2	–	–
CPP – MOZBUILD	16	308	0	0	14	4
With features from configuration						
CPP	0	0	0	0	0	0
MOZBUILD	0	0	0	0	0	0
CPP – MOZBUILD	12	22	0	0	0	0

9.3.2.3 Observations

Table 9.7 summarizes the obtained results. Full-mandatory anomalies are not identified for the CPP and MOZBUILD anomalies as the expression of the full-mandatory anomaly relies on both internal and external $\mathcal{PC}$s, thus identifying them is possible only for anomalies between at least two spaces. Partially dead anomalies are not identified for the CPP anomalies as a block always has a single including asset being its source file, thus it cannot be partially dead. Although our model allows representing missing dead features, these were not computed as we know our set of features to be incomplete making the potential outcome not representative. Finally, while we analyzed the manually extracted features from the configuration to find anomalies in this space, none were identified. We therefore omitted it in the table. In the following paragraphs, when boolean formulas are given to detail anomalies, we use underlining to exhibit conflicting terms.

CPP anomalies 533 code blocks have been identified as totally dead, of which 499 have for condition `#if 0` for which we can assume that they have been implemented on purpose by developers to deactivate parts of the code, making 34 relevant identified anomalies. An example is given in Listing 9.2, where the dead block has for condition $\underline{\neg\ \texttt{ABSL\_USES\_STD\_OPTIONAL}} \wedge (\underline{\texttt{ABSL\_USES\_STD\_OPTIONAL}} \wedge \underline{\texttt{\_\_GLIBCXX\_\_}})$.

2 code blocks have been identified as totally core. One block[11] is the `else` branch of a `#if 0` and thus has as $\mathcal{PC}$ $\neg false$, and the other one[12] has for condition `# if YYDEBUG || YYERROR_VERBOSE || 1` that is translated in `YYDEBUG` $\vee$ `YYERROR_VERBOSE` $\vee$ `true` and is always true.

MOZBUILD anomalies 59 source files have been identified as partially dead, of which an example is given in Listing 9.3. `/js/app.mozbuild` (listing 9.3a) includes `/tools/fuzzing/moz.build` (listing 9.3b) under the `JS_STANDALONE` $\wedge$

[11] https://hg.mozilla.org/mozilla-central/file/a66dcaea419641c5483a43aa7f577b70908d147f/gfx/cairo/cairo/src/cairo-qt-surface.cpp#l1412 (l.1412-1447)

[12] https://hg.mozilla.org/mozilla-central/file/a66dcaea419641c5483a43aa7f577b70908d147f/third_party/rust/glslopt/glsl-optimizer/src/compiler/glsl/glsl_parser.cpp#l828 (l.828-902)

```
#if !defined(ABSL_USES_STD_OPTIONAL)
...
#if defined(ABSL_USES_STD_OPTIONAL) && defined(__GLIBCXX__)    <-- dead
// libstdc++ std::optional implementation (as of 7.2) has a bug: when T is
// trivially copyable, optional<T> is not trivially copyable (due to one of
// its base class is unconditionally nontrivial).
#define ABSL_GLIBCXX_OPTIONAL_TRIVIALITY_BUG 1
#endif
#endif
```

Listing 9.2: Anomaly internal to the CPP space (`third_party/libwebrtc/third_party/abseil-cpp/absl/types/optional_test.cc`)

`FUZZING` condition. All source files added in `/tools/fuzzing/moz.build` with the $\neg$ `JS_STANDALONE` condition thus have for $\mathcal{PC}$ (`JS_STANDALONE` $\wedge$ `FUZZING`) $\wedge \neg$ `JS_STANDALONE`, making them dead.

15 totally core files have been identified, of which 4 are false-positives resulting from limitations of our parsing capabilities. The MozBuild system being implemented in Python, some conditions are implemented in a non-standard way (*e.g.*, conditions on variables, use of built-in functions, string formatting) and are therefore not parsed by our parser. Considering these statements would require evaluating them. The remaining 11 are actually files added in different lists depending on some condition and that are considered as always being added (*cf.* Listing 9.4). Although this behavior is clearly intentional and therefore cannot be considered as an anomaly, we believe that identifying this pattern is relevant as it corresponds to usage of the file selection system to handle the source files differently. Identifying such a pattern could therefore be useful to issue a warning to the developer and ensure that the behavior is really motivated.

Out of the 2 identified partially core files. One is a `moz.build` file being included without any condition, while the other one is a compilation partially core compilation flag depicted in Listing 9.5. The `CC_TYPE__clang__` $\vee$ `CC_TYPE__clang-cl__` condition is redundant as it is first added without condition.

CPP – MozBuild anomalies 16 totally dead code blocks have been identified due to conflicts between constraints in both the CPP and MozBuild spaces. They mainly represent conflicts created on purpose to throw errors specifying that this combination is indeed not valid and prevent conflicts to happen throughout the codebase evolution. For example, such a block identified in `js/src/jit/x86-shared/Assembler-x86-shared.cpp`[13] throws a `"Wrong architecture. Only x86 and x64 should build this file!"` error. This way, the build is stopped in case the file is selected by the MozBuild when building for an architecture other that x86 or x64. On the opposite, other identified anomalies appear to be unintentional as the example given in Listing 9.6.

- B2's $\mathcal{PC}_{Int}$: $\neg$ `MOZ_SANDBOX`
- `ref_counted.cc`'s $\mathcal{PC}_{Ext}$: `MOZ_SANDBOX` $\wedge \neg$ (`OS_ARCH__Linux__` $\vee$ `OS_ARCH__Darwin__`) $\wedge$ `OS_ARCH__WINNT__`

[13] https://hg.mozilla.org/mozilla-central/file/a66dcaea419641c5483a43aa7f577b70908d147f/js/src/jit/x86-shared/Assembler-x86-shared.cpp#l16

```
40 if CONFIG['JS_STANDALONE'] and CONFIG['FUZZING']:
41     DIRS += [
42     '/tools/fuzzing/',
43     ]
```

(a) `/js/app.mozbuild`

```
12 if not CONFIG["JS_STANDALONE"]:      <-- dead
13     DIRS += [
14         "common",
15         "faulty",
16         "messagemanager",
17         "shmem",
18         "ipc",
19     ]
20
21     if CONFIG["FUZZING_SNAPSHOT"]:
22         DIRS += [
23             "nyx",
24         ]
25
26     if CONFIG["LIBFUZZER"]:
27         DIRS += [
28             "rust",
29         ]
```

(b) `/tools/fuzzing/moz.build`

Listing 9.3: Partially dead anomaly internal to the MOZBUILD space

Therefore, `B2` can never be selected whenever `ref_counted.cc` is, making it a dead block. On the opposite, `B1`'s selection condition is redundant as the file is selected if `MOZ_SANDBOX` is satisfied. `B1` is therefore one of the 18 identified full-mandatory blocks.

Regarding partially dead blocks, they mainly represent `#ifndef` statements on a variable that, if entered, define this variable with a `#define`, ensuring that it is always set (*e.g.*, in `/accessible/other/XULListboxAccessibleWrap.h`[14]).

Anomalies involving the configuration space We did not identify any anomaly between the configuration and one of the CPP or MOZBUILD spaces. However, we identified 22 partially dead and 12 totally dead code blocks due to conflicting constraints between the three spaces. Listing 9.7 illustrates 6 of the 12 identified totally dead blocks.

Constraints from the configuration space From the code excerpt of listing 9.7a, we can extract the following constraints:

- $\texttt{CANONICAL\_KERNEL\_LINUX} \rightarrow (\neg\ \texttt{OS\_ANDROID} \wedge \texttt{OS\_LINUX}) \vee \texttt{OS\_ANDROID}$
- $\texttt{CANONICAL\_OS\_GNU} \rightarrow (\neg\ \texttt{OS\_ANDROID} \wedge \texttt{OS\_LINUX})$

[14] https://hg.mozilla.org/mozilla-central/file/a66dcaea419641c5483a43aa7f577b70908d147f/accessible/other/XULListboxAccessibleWrap.h#l6

```
111 if CONFIG["MOZ_WIDGET_TOOLKIT"] == "windows":
112     SOURCES += ["DecodePool.cpp"]
113 else:
114     UNIFIED_SOURCES += ["DecodePool.cpp"]
```

Listing 9.4: False-positive core anomaly internal to the MozBuild space (`/image/moz.build`)

```
245 CFLAGS += [
246     ...
247     '-Wno-incompatible-pointer-types',
248     ...
249 ]
250 if CONFIG['CC_TYPE'] in ('clang', 'clang-cl'):
251     CFLAGS += [
252         ...
253         '-Wno-incompatible-pointer-types',    <-- core
254         ...
255     ]
```

Listing 9.5: Partially core anomaly internal to the MozBuild space (`/gfx/cairo/cairo/src/moz.build`)

- $\texttt{OS\_ARCH\_\_Linux\_\_} \rightarrow \texttt{CANONICAL\_KERNEL\_LINUX} \wedge \texttt{CANONICAL\_OS\_GNU}$

Constraints from the MozBuild space The only condition to select the `platform_thread_posix.cc` file is $\texttt{OS\_ARCH\_\_Linux\_\_}$ (listing 9.7b) that, given the constraints from the configuration space, is equivalent to $\neg\ \texttt{OS\_ANDROID} \wedge \texttt{OS\_LINUX}$.

Constraints from the CPP space The condition to select the code block at line 191 of listing 9.7d is $\texttt{OS\_FUCHSIA} \wedge \neg\ \texttt{OS\_MACOSX} \wedge \neg\ \texttt{OS\_LINUX} \wedge \neg\ \texttt{OS\_ANDROID}$, Finally, the condition to select this block by taking into account constraints from all spaces is $(\texttt{OS\_FUCHSIA} \wedge \neg\ \texttt{OS\_MACOSX} \wedge \underline{\neg\ \texttt{OS\_LINUX}} \wedge \neg\ \texttt{OS\_ANDROID}) \wedge (\neg\ \texttt{OS\_ANDROID} \wedge \underline{\texttt{OS\_LINUX}})$, that is unsatisfiable. Consequently, the successive `#elif` and `#else` blocks are also dead as their $\mathcal{PC}$s also inherit from the $\neg\ \texttt{OS\_LINUX}$ condition.

9.3.3 Implementation

The proposed framework is decomposed into several independent entities, illustrated on Figure 9.4. Extracting CPP constraints in the Mozilla Gecko codebase is done using PilzTaker[15] (itself relying on Undertaker). MozBuild files being written in Python, we designed a MozBuild parser in Python relying on the Python AST library[16] to extract a JSON representation of the MozBuild files. As Sincero et al. [2010] did for CPP when analyzing the Linux kernel, conditions are translated in boolean statements in a similar way.

The main part of the framework identifying the anomalies is the **pc-identificator** that uses as input the previously extracted constraints from all spaces. It is written in Java and relies on

[15] https://github.com/SSE-LinuxAnalysis/pilztaker
[16] https://docs.python.org/3/library/ast.html

```
39 if CONFIG['MOZ_SANDBOX']:
40     DIRS += ['/security/sandbox']
```

(a) /toolkit/toolkit.mozbuild

```
20 if CONFIG["OS_ARCH"] == "Linux":
21     DIRS += ["linux"]
22 elif CONFIG["OS_ARCH"] == "Darwin":
23     DIRS += ["mac"]
24 elif CONFIG["OS_ARCH"] == "WINNT":
25     ...
26     SOURCES += [
27         ...
28         "chromium/base/memory/ref_counted.cc",
29         ...
30     ]
```

(b) /security/sandbox/moz.build

```
84 #if defined(MOZ_SANDBOX)    <-- B1 is full-mandatory
85   return true;
86 #else    <-- B2 is dead
87   return sequence_checker_.CalledOnValidSequence() ||
88          g_cross_thread_ref_count_access_allow_count.
               load() != 0;
89 #endif
```

(c) /security/sandbox/chromium/base/memory/ref_counted.cc

Listing 9.6: Example of total dead anomaly between the CPP and MOZBUILD spaces

Sat4j[17] for solving the boolean formulas. The information characterizing the identified anomalies are then output in a JSON format. Finally, the model presented in Section 9.2 is implemented in EMF[18] and the identified anomalies are instantiated using the classes generated from the model. Every part of the framework is independent and Dockerized to ease the reproduction of the results and independent reuse on other codebases. As for scalability, the overall analysis of a product that can be derived from the Mozilla build system (extraction of constraints and identification of anomalies) takes less than two hours (Table 9.8).

9.4 Threats to validity

As we do not provide a validation by actual developers or architects of any system, the main threat to the validity of our work regards the relevance of the identified anomalies. For example, some patterns identified as anomalies in Mozilla Gecko correspond to usages of the build system for other purposes than variability management (*cf.* Section 9.3.2.3). Similarly, `#if 0` blocks can be used intentionally as a way to comment parts of the implementation. We can also imagine that

[17] https://www.sat4j.org/index.php
[18] https://www.eclipse.org/modeling/emf/

```
468 if "android" in os:
469     canonical_os = "Android"
470     canonical_kernel = "Linux"
471 elif os.startswith("linux"):
472     canonical_os = "GNU"
473     canonical_kernel = "Linux"
```

```
757 elif target.kernel == "Darwin" or (target.kernel == "Linux" and target.os == "
        GNU"):
758     os_target = target.kernel
759     os_arch = target.kernel
```

(a) /build/moz.configure/init.configure

```
20 if CONFIG["OS_ARCH"] == "Linux":
21     DIRS += ["linux"]
```

(b) /security/sandbox/moz.build

```
26 UNIFIED_SOURCES += [
27     ...
28     "../chromium/base/threading/platform_thread_posix.cc",
29     ...
30 ]
```

(c) /security/sandbox/linux/moz.build

```
176 #if defined(OS_MACOSX)
177   return pthread_mach_thread_np(pthread_self());
178 #elif defined(OS_LINUX)
179   ...
180 #elif defined(OS_ANDROID)
181   return gettid();
182 #elif defined(OS_FUCHSIA)    <-- dead
183   return zx_thread_self();
184 #elif defined(OS_SOLARIS) || defined(OS_QNX)    <-- dead
185   return pthread_self();
186 #elif defined(OS_NACL) && defined(__GLIBC__)    <-- dead
187   return pthread_self();
188 #elif defined(OS_NACL) && !defined(__GLIBC__)    <-- dead
189   // Pointers are 32-bits in NaCl.
190   return reinterpret_cast<int32_t>(pthread_self());
191 #elif defined(OS_POSIX) && defined(OS_AIX)    <-- dead
192   return pthread_self();
193 #elif defined(OS_POSIX) && !defined(OS_AIX)    <-- dead
194   return reinterpret_cast<int64_t>(pthread_self());
195 #endif
```

(d) /security/sandbox/chromium/base/threading/platform_thread_posix.cc

Listing 9.7: Example of total dead anomaly between the CPP, MozBuild and Configuration spaces

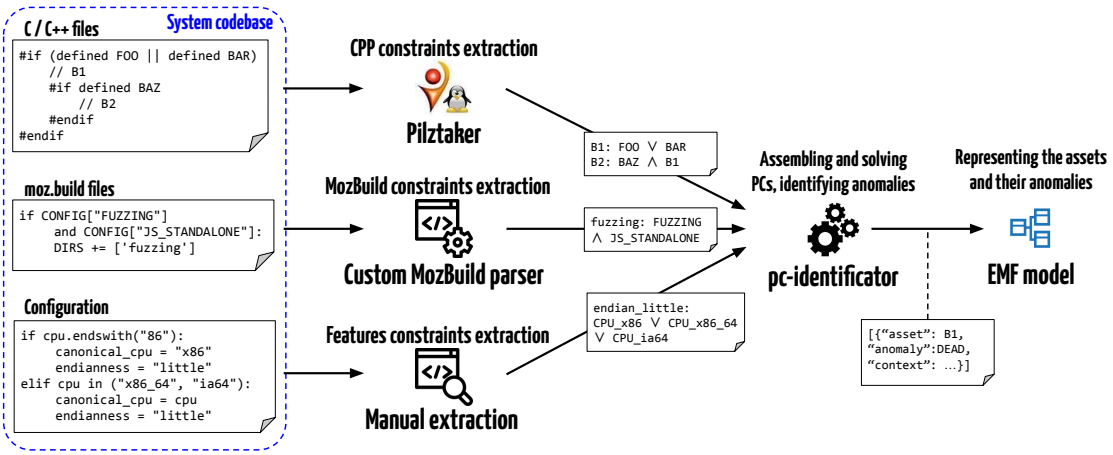


Figure 9.4: Identification process

partial dead MOZBUILD anomalies throwing exceptions are not anomalies and act as a safeguard to prevent errors. Nevertheless, we believe that knowing the location of such implementations is important for the evolution of the system, and our framework is capable of detecting them. Having feedback from the developers would help us confirm whether these behaviors are common practice and intended, or correspond to actual anomalies.

Another threat regards the evaluation of our approach on the Linux kernel build system and BusyBox, as the mapping achieved between KernelHaven and our approach was limited to dead anomalies in the implementation and did not take into account the configuration space. This limitation to dead anomalies is due to KernelHaven's UndeadAnalyzer only outputting dead blocks, while the configuration space could not be taken into account as state-of-the-art approaches represent the whole KCONFIG as a single CNF formula [She et al., 2010; El-Sharkawy et al., 2015; Kästner, 2016] (Section 9.3.1.2). However, anomalies involving the configuration space were identified in Mozilla Gecko, giving us confidence in the soundness of our representation.

Regarding the results obtained on the Mozilla Gecko, although our study considers all three spaces of constraints, we did not possess a complete variability model and we manually extracted some features. It is therefore likely that some anomalies involving other configuration constraints were missed, or on the opposite that the ones we identified are false positives as we did not understand them correctly. Feedback from developers of the Mozilla build system would help us assess the relevance of both the features and constraints we extracted, and the identified anomalies.

Finally, the MOZBUILD being designed as a sandbox on top of the Python language, the conditions present in the `moz.build` scripts make use of Python statements such as `startsWith` (*e.g.*, `CONFIG["OS_ARCH"].startswith("GNU_")`[19]) that cannot be directly transformed in boolean. Consequently, some conditions cannot be parsed and may be related to some anomalies that we therefore cannot identify. Completing our static analysis with a dynamic approach executing the MOZBUILD sandbox with known configurations could help this identification.

[19] https://github.com/mozilla/gecko-dev/blob/6a850a50bbe7b97e1a3ac8b659ec15a5fc0b623a/xpcom/reflect/xptcall/md/unix/moz.build#L42

Table 9.8: Execution time for each analysis step on Mozilla Gecko for the Firefox web browser, Thunderbird mail client, the SeaMonkey suite and the Android Firefox application.

Step	Execution time			
	Firefox browser	**Thunderbird**	**SeaMonkey**	**Android app**
Extracting constraints				
CPP space	5 min 26 s			
MozBuild space	1 min 26 s			
Total	**6 min 52 s**			
Identifying anomalies				
CPP + CPP – MozBuild	9 min 24 s	9 min 10 s	9 min 01 s	9 min 07 s
MozBuild	12 s	12 s	14 s	11 s
Configuration	300 ms	400 ms	250 ms	170 ms
CPP – Configuration	25 s	24 s	24 s	27 s
MozBuild – Configuration	14 s	13 s	12 s	12 s
CPP – MozBuild – Configuration	1 h 37 min 12 s	1 h 29 min 24 s	1 h 28 min 50 s	1 h 21 min 38 s
Total	**1 h 47 min 36 s**	**1 h 39 min 31 s**	**1 h 28 min 50 s**	**1 h 31 min 45 s**

System configuration: Ubuntu 18.04.2 LTS with Intel Xeon CPU E5-2637v2 @ 3.5GHz and 128Go memory.

9.5 Conclusion

In order to evaluate whether the Linux-centered model introduced in Chapter 8 is applicable to other build systems and answers Challenges B1 and B2, we evaluated its applicability on the Mozilla build system as another example of ad hoc variability-aware build system. Consequently, the model has been generalized to support the diversity of mechanisms it exhibits compared to the Linux kernel build system, leading to the definition of new anomalies. The model has then been implemented in a framework enabling automatic identification of anomalies in both the Linux kernel build system and the Mozilla build system. The evaluation conducted on the Linux kernel build system, showed that all dead anomalies in and between the CPP and the KBuild identified by state-of-the-art approaches are also identified by our framework, validating the coverage of existing work regarding these anomalies. The application of the complete model on the Mozilla build system led to the identification of anomalies that, after manual analysis, appear to be either (*i*) relevant anomalies, (*ii*) intentional conflicts caused for safety checks or (*iii*) uses of the build system's capacities for other purposes.

As a result, the updated representation of assets in the build system keeps a fine-grain view of the relationships that exist between them, and the differentiation between spaces involved in presence conditions increases the understanding of anomalies, improving the answer to Challenge B1 brought by our first version in Chapter 8. Additionally, the conducted evaluation demonstrates the capacity of our framework to identify anomalies and, consequently, the relevance of their representation, improving the answer to Challenge B2 given by our previous model.

Although bringing additional elements of answers for both challenges, this chapter does not yet allow answering them completely. The analysis of the Mozilla build system's mechanisms and their incorporation in our model enabled a first generalization of our model to represent the selection of assets in a build system. However, it is likely that other build systems as the one of the

Chromium platform[20] or JHipster [Halin et al., 2017] use yet other mechanisms. Fully answering Challenge B1 therefore requires a study of additional build systems to gain a better overview of the diversity of mechanisms they rely on. Regarding Challenge B2, our answer is limited by the extent of our validation, which would be improved by taking into account the Linux kernel build system's configuration space and a validation of the identified anomalies in the Mozilla build system.

[20] https://chromium.googlesource.com/chromium/src/tools/gn/+/48062805e19b4697c5fbd926dc649c78b6aaa138/README.md

CHAPTER 10

Conclusion and perspectives

When large-scale variability-intensive software systems are not organized as SPLs, managing their implemented variability is challenging.

In the case of OO systems, their variability is often implemented by relying on OO mechanisms. As they are not especially designed for variability implementation, they do not allow a clear mapping with the domain variability that is, most often, missing. Consequently, information on the implemented variability and its location in such codebases is lost, hampering its comprehension and, thereby, the maintenance and evolution capabilities of the system, eventually threatening its overall quality. There is therefore a need to identify and comprehend these variability implementations to master them and their quality.

In addition, large scale highly-variable systems often rely on build systems to manage the different build steps of the system. Instead of relying on model-driven approaches to shape their variability, they often reuse off-the-shelf solutions and adapt them to implement variability to select code assets at multiple granularities. However, their variability management capacities are limited as (*i*) they do not incorporate mechanisms allowing to ensure the consistency of the variability they implement and (*ii*) they are not aware of the variability that is implemented in other steps of the build system. As a result, conflicts can happen between conditions to select assets, leading to anomalies prone to bugs in the derived variants of the system. It is therefore essential to obtain a global view of the implemented variability in such build systems to prevent these anomalies.

10.1 Summary of the contributions

This section details the challenges that we tackled in this thesis towards these goals, summarizing for each of them the proposed solutions and their limitations.

A. Comprehending variability implemented in OO software systems

A1. Identifying variability implemented in OO software systems

In Chapter 4, we assessed the relevance of the identification technique relying on the density of symmetries in OO constructs proposed by Tërnava et al. [2019] on two aspects. This method has been implemented in the *symfinder* toolchain, allowing automatic identification of variability implementations in Java systems implemented in a single codebase. They are then visualized in the shape of a graph, exhibiting concentrations of nodes corresponding to dense zones of variability

implementations. First, we extended the identification technique to another OO language, C++. The application of the toolchain on systems in both Java and C++ systems led to the identification of dense zones of symmetries that could be visually correlated to information from the domain variability. This relevance was then confirmed by an automatic mapping of these identified variability implementations with reverse-engineered feature traces from two systems, ArgoUML-SPL and Sat4j, validating that an important part of the implemented variability is identified by *symfinder*. However, a non-negligible amount of false positives were also identified. Consequently, in Chapter 5, we refined the identification technique by taking into account usage relationships between classes and, relying on them, formally characterized the notion of density of OO variability implementations. This density is parameterized, and adapting these parameters allows identifying *hotspot* classes maximizing the density and filtering out less dense zones of variability implementations.

While the density of symmetries appears as an answer to Challenge A1, this definition exhibits limitations. As for the number of identified *vp*-s and variants, we could observe that for given parameters, the number of the classes identified as hotspots can importantly differ between two projects, even of similar sizes. It is therefore likely that other mechanisms are involved in OO variability implementations and lead to their density, and need to be explored as they could be used to help determine adequate parameters for the density measure, that are actually to be determined manually. Additionally, we did not conduct an evaluation of the relevance of the identified hotspots. Tërnava et al. [2022] introduced a preliminary definition of density considering only the inheritance relationships. This measure has for only parameter a threshold on the minimum number of variants at class or method level (*i.e.*, the individual density) for a *vp* to be identified as a hotspot. They then identified hotspots in ArgoUML-SPL and Sat4j, defining multiple values of threshold and mapping for each threshold the identified hotspots to the features traces. It results that the number of false positives diminishes when increasing the threshold, showing that using this parameter to filter variability implementations allows outputting a subset of more relevant variability implementations. We, therefore, have good confidence about the relevance of *vp*-s filtered with a density measure enriched with usage relationships.

A2. Making the identified variability implementations comprehensible

The *symfinder* toolchain [Mortara et al., 2019] proposes a graph visualization of the identified *vp*-s and variants that allowed the authors to distinguish relevant zones concentrating variability implementations. However, no evaluation of the understandability of the visualization has been performed. In Chapter 4, we conducted an empirical evaluation with Daniel Le Berre, software architect of Sat4j, and gathered his feedback on using the tool and the provided visualization to comprehend the variability implemented in his system. It results that the visualization allowed him to distinguish relevant variability implementations. Henceforth, this representation has been kept and extended to visualize the additional usage relationships in *symfinder-2* (Chapter 5). However, although filtering options were added to enable focusing on particular zones of the visualization, the diversity of information that needed to be displayed showed the limits of this solution. In Chapter 6, *VariCity* was designed as a 3D visualization relying on the metaphor of the city, adapted to OO variability implementations. Two evaluations were conducted. A first evaluation based on onboarding scenarios demonstrated that the interactions provided by the view allow to gradually explore the system and gather information about the implemented variability. Then, a controlled experiment measuring the gain brought by *VariCity* compared to the use of an IDE to solve vari-

ability identification tasks showed that subjects using *VariCity* could answer more correctly to the given tasks, and were completing them faster and more easily. Finally, the visualization has been embedded (Section 7.5 in Chapter 7), enabling bidirectional between the implementation and the visualization in an integrated environment preventing context switching.

These results give us confidence in the fact that *VariCity* is a relevant answer to Challenge A2. Further validating this claim would however require an extended user experiment. Due to organizational reasons, the conducted experiment only considers subjects students with similar levels of experience. Additionally, the size of the panel constrained us to evaluate *VariCity* on a single subject system. Conducting an experiment with a larger panel of subjects of different levels of experience would allow us to consider multiple object systems and mitigate biases related to the experience of the subjects and the chosen object system.

A3. Understanding the quality of the implemented variability

Relying on the definition of *variability debt* and its causes given by Wolfart et al. [2021], we defined variability debt in the context of OO variability implementations (Section 3.3). In Chapter 7, we defined OO quality metrics allowing their identification and proposed *VariMetrics*. This solution extends the *VariCity* approach introduced in Chapter 6 and allows to visualize altogether the variability implementations and OO quality metrics on the system to reveal indebted zones dense in variability implementations. The view is configurable and allows the combination of quality metrics on multiple visual axes. The conducted quantitative and qualitative evaluations showed the capacity of *VariMetrics* to reveal indebted zones dense in variability implementations which, when refactored, allow for increasing the quality of the system.

By revealing simultaneously the zones of the system being dense in variability implementations and/or critical regarding their quality, *VariMetrics* meets Challenge A3. Although the qualitative evaluation consisting of the refactor of code identified as both variability dense and quality-critical showed, it also enlightened a limitation of our approach. Where some of the identified debt actually corresponded to improperly implemented variability, other parts were plain technical debt located in variability intense classes, but not linked to variability implementations. There is therefore a need to refine the definition of OO variability debt to isolate indebted variability implementations from technical debt present in variable code assets implementing variability.

B. Comprehending the variability managed by build systems

B1. Making explicit the derivation mechanism of build systems

State-of-the-art contributions describing the selection of code assets in build systems (*i*) all target the Linux kernel build system and (*ii*) all focus on isolated steps of the build systems, preventing a global view of the interactions that exist between assets in these steps. In Chapter 8, we synthesize these works in a formalism based on the generic concepts of configurator and derivator with assets. Presence conditions allow modeling with precision the relationships that exist between assets throughout the steps. In Chapter 9, we study the Mozilla build system and exhibit similarities between its derivation mechanism and the one of the Linux kernel build system. The model has then been generalized to support the Mozilla build system as another build system. Its relevance has been validated by being implemented in a framework and instantiated on the code assets of both build systems.

This representation enables a fine-grain understanding of the derivation mechanisms of these build systems, providing an answer to Challenge B1. Although it supports two build systems managing large codebases, it is very likely that other build systems of popular systems such as Chromium or JHipster [Halin et al., 2017] rely on other mechanisms.

B2. Characterizing and identifying anomalies in a build system

As each contribution from the state of the art characterizing anomalies in build systems focuses on isolated steps of the Linux kernel build system, the definitions and formalisms they rely on are not aligned. Although their definitions have been implemented in multiple tools designed explicitly for the Linux kernel build system [Sincero et al., 2010; Nadi and Holt, 2012; Kästner, 2016], these inconsistencies hamper the understanding of the causes of these anomalies and, consequently, their application to other build systems. In Chapter 8, we enumerate and analyze 25 definitions from work characterizing anomalies in the Linux kernel build system and rely on these definitions to define anomalies on assets, expressing the satisfiability formulas allowing their identification using the model's presence conditions. We then formally instantiate all these definitions in our representation, demonstrating coverage of these works. In Chapter 9, we extend these definitions to support anomalies induced by the diversity of mechanisms used in the Mozilla build system. The detection of these anomalies has been implemented in the model's framework and has been applied to both the Linux kernel build system and the Mozilla build system. Evaluation on the Linux kernel build system has been conducted by comparing the dead anomalies identified using our representation to anomalies identified using KernelHaven [Kröher et al., 2018a,b], a state-of-the-art approach to identify dead anomalies in the Linux kernel build system. It results that the proposed framework covers all the anomalies found by KernelHaven on the considered subset of the Linux kernel build system. Regarding the Mozilla build system, the application of the framework on its code assets led to the identification of multiple anomalies that appear to be relevant.

By being able, through its implementation as a framework, to identify anomalies the Linux kernel build system and the Mozilla build system, our model meets Challenge B2. The relevance of the approach is however limited by the extent of the conducted evaluation. On the Linux kernel build system, only the dead anomalies between the KBUILD and CPP were considered to be able to design a mapping with KernelHaven. We hence cannot conclude on the relevance of other anomalies. As to the extent of our knowledge, no approach identifying anomalies in the Mozilla build system could enable a comparison of the obtained results, we assessed the relevance of the identified anomalies by manually inspecting the codebase. Additionally, as some identified anomalies seemed to be intentional behaviors, a validation from maintainers of the Mozilla build system would be required to assess whether it is actually the case or not.

10.2 Perspectives

In this section, we detail perspectives on the continuation of this work both in the short and long terms.

10.2.1 Short-term perspectives

Bridging the gap between variability-aware build systems and OO variability implementations. While in this thesis we studied separately wild implementations of variability in OO

systems and wild management of variability in build systems, those two aspects can coexist in software systems. An example is the Mozilla Gecko codebase, for which an important part of the implementation is in C++ (analyzed in Chapter 4) while the selection of these code assets is managed by its build system (studied in Chapter 9). Therefore, in addition to the variability implemented in the build system that is resolved at pre-compile time, the OO architecture of the system exhibits by its nature variability that is resolved at runtime. First concerns about the interactions between compile-time and runtime variability had been enlightened by a recent line of work on *deep* software variability, mainly focusing on the interactions between configuration options at both level [Lesoil et al., 2021a,b,c]. We aim therefore to study how such systems manage this diversity of variability implementation mechanisms, and whether new types of anomalies can issue from these interactions between build system variability and OO variability implementations, requiring a model allowing their representation in a homogeneous way and allowing to reason on these interactions.

Improving the OO variability implementations identification technique. Due to their complex nature, identifying OO variability implementations is not trivial. Successive improvements to the technique proposed by *symfinder*, taking into account usage relationships and characterizing a parameterized density measure, allowed to improve its precision. While *vp*-s and variants are identified by their structure, their understanding is made possible through the use of visualizations, that are then interpreted by the architect or developer based on their knowledge of the system. For example, nomenclatures of the code assets were extensively used by the subjects of our controlled experiment in Chapter 6 to understand the implemented variability. There is therefore a need to explore whether other information from the domain can be extracted from the code asset or, when available, in other sources such as documentation or commits [Dintzner et al., 2016] and issues in the control version system to be then used to refine the results of the *vp*-s and variants identification.

Monitoring the evolution of the variability in the implementation and in build systems. Software systems are living ecosystems evolving in time on all their aspects, being their implementation [Mens et al., 2005], their quality [Sato et al., 2007; Hecht et al., 2015], or their variability at both domain and implementation level [Lotufo et al., 2010; Passos et al., 2013; Kröher et al., 2018c]. Understanding in what measure they evolve is important for their maintenance and multiple approaches have been designed to understand and master it, mainly relying on visualizations [Gall and Lanza, 2006; Diehl, 2007; Wettel and Lanza, 2008b; Steinbrückner and Lewerentz, 2013]. Regarding OO variability implementations, being able to track their evolution is of utmost importance as they are hidden in the codebase. Comprehending them is already achieved through a visualization and as visually comparing snapshot views can be confusing since the exhibited representations can importantly change between two versions of a system, there is a need to extend it to support several versions of a system. Regarding build systems, while recent work proposes a technique to easily monitor the evolution of dead anomalies in the Linux kernel build system [Kröher et al., 2022], such methods need to be extended to monitor the presence of anomalies throughout the evolution of the system.

10.2.2 Long-term perspectives

Studying the other symmetry patterns in OO structures. The proposed technique used by *symfinder* to identify OO variability implementations relies on the notion of symmetry in OO software constructs proposed by Coplien and Zhao [2000b]. This work is based on the original definition of local symmetry that Alexander [2002] previously defined in the *Theory of Centers* together with 14 other definitions of structural properties. Therefore, the symmetries present in these other structural properties might be a way to identify variability implementations that do not respect the common implementation techniques, such as code duplication for example. Achieving this goal requires instantiating these properties on the software structures and providing a technique to identify them.

Studying wild variability in other paradigms. Multiple large scale software systems are implemented using different programming languages. For example, the Mozilla Gecko codebase relies on 48 different programming languages[1], with C++ and JavaScript representing around 27% of the total LoCs each one. The adoption of JavaScript as a support language for large scale applications is increasing and some work from the SPL community is oriented towards using it as a support language for variable systems [Machado et al., 2014; Halin et al., 2017; Cortiñas et al., 2022]. Studying how variability is implemented in such a language would complement the study of the previously introduced perspective on the interactions between variability mechanisms in the build system and the implementation code by enabling the study of the interactions between different variability implementations in the implementation code. This study requires first, in the case of JavaScript, a characterization of how variability is implemented in systems using this language and a method to identify it. Only then a language-independent unified view of the implemented variability with a characterization of their interactions can be built.

Exploring the impact of OO variability implementations on other software properties. Variability implementations are known to have an effect on the comprehension [Galster et al., 2017; Medeiros et al., 2017] and the management of the evolution of a system [Metzger and Pohl, 2014], consequently hampering the quality of the system [Wolfart et al., 2021]. To tackle this issue, in Chapter 7, we proposed *VariMetrics* as a first approach unifying in a single representation OO variability implementations and software metrics. Additionally to their impact on software maintenance, software quality defects such as code smells are known to cause an increase in the energy consumption of an application [Carette et al., 2017]. Software energy consumption recently gained a lot of interest for the scientific community [Pinto et al., 2015] and configuring software systems taking into account energetical constraints is challenging [Götz et al., 2012, 2013]. Moreover, recent work shows that refactoring functional code impacts the energy consumption of a system [Ournani et al., 2021]. We aim therefore to extend our analysis of the effects of OO variability implementations on other properties of a system, such as its energy consumption. Achieving this goal requires being able to characterize and measure the energy consumption of variability implementations.

[1] https://www.openhub.net/p/firefox/analyses/latest/languages_summary

Bibliography

Iago Abal, Claus Brabrand, and Andrzej Wasowski. 42 variability bugs in the linux kernel: a qualitative analysis. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 421–432, 2014.

Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B France. Slicing feature models. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 424–427. IEEE, 2011.

Mathieu Acher, Roberto E Lopez-Herrejon, and Rick Rabiser. Teaching software product lines: A snapshot of current practices and challenges. *ACM Transactions on Computing Education (TOCE)*, 18(1):1–31, 2017.

Bram Adams, Herman Tromp, Kris De Schutter, and Wolfgang De Meuter. Design recovery and maintenance of build systems. In *2007 IEEE International Conference on Software Maintenance*, pages 114–123. IEEE, 2007.

Christopher Alexander. *The nature of order: an essay on the art of building and the nature of the universe. Book 1, The phenomenon of life*. Center for Environmental Structure, 2002.

Christopher Alexander and Susan Carey. Subsymmetries. *Perception & Psychophysics*, 4(2): 73–77, 1968.

Hussein Alrubaye, Mohamed Wiem Mkaouer, and Anthony Peruma. Variability in library evolution: An exploratory study on open-source java libraries. In *Software Engineering for Variability Intensive Systems - Foundations and Applications*. Taylor & Francis Group, 2019.

Berima Andam, Andreas Burger, Thorsten Berger, and Michel RV Chaudron. Florida: Feature location dashboard for extracting and visualizing feature traces. In *Proceedings of the Eleventh International Workshop on Variability Modelling of Software-intensive Systems*, pages 100–107. ACM, 2017.

Giuliano Antoniol, Umberto Villano, Ettore Merlo, and Massimiliano Di Penta. Analyzing cloning evolution in the linux kernel. *Information and Software Technology*, 44(13):755–765, 2002.

Sven Apel, Thomas Leich, Marko Rosenmüller, and Gunter Saake. Featurec++: On the symbiosis of feature-oriented and aspect-oriented programming. In *International Conference on Generative Programming and Component Engineering*, pages 125–140. Springer, 2005.

Sven Apel, Martin Kuhlemann, and Thomas Leich. Generic feature modules: Two-staged program customization. In *ICSOFT (1)*, pages 127–132, 2006.

Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines*. Springer, 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(6):2972–3016, 2017.

Paris Avgeriou, Philippe Kruchten, Ipek Ozkaya, and Carolyn Seaman. Managing technical debt in software engineering (dagstuhl seminar 16162). In *Dagstuhl Reports*, volume 6. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

Paris C Avgeriou, Davide Taibi, Apostolos Ampatzoglou, Francesca Arcelli Fontana, Terese Besker, Alexander Chatzigeorgiou, Valentina Lenarduzzi, Antonio Martini, Athanasia Moschou, Ilaria Pigazzini, et al. An overview and comparison of technical debt measurement tools. *IEEE Software*, 38(3):61–71, 2020.

Maider Azanza, Arantza Irastorza, Raul Medeiros, and Oscar Díaz. Onboarding in Software Product Lines: Concept Maps as Welcome Guides. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, pages 122–133. IEEE, 2021.

Felix Bachmann and Paul Clements. Variability in software product lines. Technical Report CMU/SEI-2005-TR-012, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2005. URL http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=7675.

Ebrahim Bagheri and Dragan Gasevic. Assessing the maintainability of software product line feature models using structural metrics. *Software Quality Journal*, 19(3):579–612, 2011.

Maria Teresa Baldassarre, Valentina Lenarduzzi, Simone Romano, and Nyyti Saarimäki. On the diffuseness of technical debt items and accuracy of remediation time when using SonarQube. *Information and Software Technology*, 128:106377, 2020.

Gergő Balogh and Arpad Beszedes. CodeMetropolis-code visualisation in MineCraft. In *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 136–141. IEEE, 2013.

Michael Balzer, Andreas Noack, Oliver Deussen, and Claus Lewerentz. Software landscapes: Visualizing the structure of large software systems. In *IEEE TCVG*, 2004.

Michael Balzer, Oliver Deussen, and Claus Lewerentz. Voronoi treemaps for the visualization of software metrics. In *Proceedings of the 2005 ACM symposium on Software visualization*, pages 165–172, 2005.

Don Batory. Feature models, grammars, and propositional formulas. In *International Conference on Software Product Lines*, SPLC '05, pages 7–20. Springer, 2005. doi: 10.1007/11554844_3.

Fabian Beck, Michael Burch, Stephan Diehl, and Daniel Weiskopf. A taxonomy and survey of dynamic graph visualization. In *Computer graphics forum*, volume 36, pages 133–159. Wiley Online Library, 2017.

Andrew Begel and Beth Simon. Struggles of new college graduates in their first software development job. In *Proceedings of the 39th SIGCSE technical symposium on Computer science education*, pages 226–230, 2008.

David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information systems*, 35(6):615–636, 2010.

Alexandre Bergel, Razan Ghzouli, Thorsten Berger, and Michel R. V. Chaudron. FeatureVista: Interactive Feature Visualization. In *Proceedings of the 25th ACM International Systems and Software Product Line Conference - Volume A*, page 196–201, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450384698. URL https://doi.org/10.1145/3461001.3471154.

Thorsten Berger and Christian Kästner. Kbuildminer, 2016. URL https://github.com/ckaestne/KBuildMiner.

Thorsten Berger, Steven She, Rafael Lotufo, Krzysztof Czarnecki, and Andrzej Wasowski. Feature-to-Code Mapping in Two Large Product Lines. In *SPLC*, pages 498–499. Citeseer, 2010.

Lucy M Berlin. Beyond program understanding: A look at programming expertise in industry. *ESP*, 93(744):6–25, 1993.

Danilo Beuche. Industrial variant management with pure::variants. In *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume B*, pages 37–39. ACM, 2019. doi: 10.1145/3307630.3342391. URL https://doi.org/10.1145/3307630.3342391.

Cor-Paul Bezemer, Shane McIntosh, Bram Adams, Daniel M German, and Ahmed E Hassan. An empirical study of unspecified dependencies in make-based build systems. *Empirical Software Engineering*, 22(6):3117–3148, 2017.

Grady Booch, Robert A Maksimchuk, Michael W Engle, Bobbi J Young, Jim Connallen, and Kelli A Houston. Object-oriented analysis and design with applications. *ACM SIGSOFT software engineering notes*, 33(5):29–29, 2008.

Paulo Borba, Leopoldo Teixeira, and Rohit Gheyi. A theory of software product line refinement. *Theoretical Computer Science*, 455:2–30, 2012.

Mark Bruls, Kees Huizing, and Jarke J Van Wijk. Squarified treemaps. In *Data visualization 2000*, pages 33–42. Springer, 2000.

Michael Burch, Corinna Vehlow, Fabian Beck, Stephan Diehl, and Daniel Weiskopf. Parallel edge splatting for scalable dynamic graph visualization. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2344–2353, 2011.

John Businge, Moses Openja, Sarah Nadi, Engineer Bainomugisha, and Thorsten Berger. Clone-based variability management in the android ecosystem. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 625–634. IEEE, 2018.

G Ann Campbell. Cognitive complexity: An overview and evaluation. In *Proceedings of the 2018 international conference on technical debt*, pages 57–58, 2018.

Rafael Capilla, Jan Bosch, Kyo-Chul Kang, et al. Systems and software variability management. *Concepts Tools and Experiences*, 2013.

Antonin Carette, Mehdi Adel Ait Younes, Geoffrey Hecht, Naouel Moha, and Romain Rouvoy. Investigating the energy impact of android smells. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 115–126. IEEE, 2017.

Pierre Caserta and Olivier Zendra. Visualization of the static aspects of software: A survey. *IEEE transactions on visualization and computer graphics*, 17(7):913–933, 2010.

Pierre Caserta, Olivier Zendra, and Damien Bodénes. 3d hierarchical edge bundles to visualize relations in a software city metaphor. In *2011 6th International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*, pages 1–8. IEEE, 2011.

Noptanit Chotisarn, Leonel Merino, Xu Zheng, Supaporn Lonapalawong, Tianye Zhang, Mingliang Xu, and Wei Chen. A systematic literature review of modern software visualization. *Journal of Visualization*, 23(4):539–558, 2020.

Dave Clarke, Michiel Helvensteijn, and Ina Schaefer. Abstract delta modeling. *ACM Sigplan Notices*, 46(2):13–22, 2010.

Paul Clements and Linda Northrop. *Software product lines*. Addison-Wesley Boston, 2002.

Fatima Nur Colakoglu, Ali Yazici, and Alok Mishra. Software product quality metrics: A systematic mapping study. *IEEE Access*, 2021.

James Coplien, Daniel Hoffman, and David Weiss. Commonality and variability in software engineering. *IEEE Software*, 15(6):37–45, 1998. doi: 10.1109/52.730836. URL https://doi.org/10.1109/52.730836.

James O. Coplien. *Multi-Paradigm Design for C++*. Addison-Wesley Longman Publishing Co., Inc., 1999.

James O. Coplien. The future of language: Symmetry or broken symmetry? In *Proceedings of VS Live 2001*, pages 1–7, 2001. URL https://sites.google.com/a/gertrudandcope.com/info/Publications/Patterns/Symmetry/FutureOfLanguage.

James O. Coplien and Liping Zhao. Symmetry breaking in software patterns. In *International Symposium on Generative and Component-Based Software Engineering*, GCSE 2000, pages 37–54. Springer, Springer, 2000a.

James O Coplien and Liping Zhao. Symmetry and symmetry breaking in software patterns. In *Proceedings Second International Symposium on Generative and Component Based Software Engineering (GCSE2000)*, pages 373–398, 2000b.

Alejandro Cortiñas, Miguel R Luaces, and Óscar Pedreira. spl-js-engine: a javascript tool to implement software product lines. In *Proceedings of the 26th ACM International Systems and Software Product Line Conference-Volume B*, pages 66–69, 2022.

Marcus Vinicius Couto, Marco Tulio Valente, and Eduardo Figueiredo. Extracting software product lines: A case study using conditional compilation. In *2011 15th European Conference on Software Maintenance and Reengineering*, pages 191–200. IEEE, 2011.

Daniel Cruz, Eduardo Figueiredo, and Jabier Martinez. A literature review and comparison of three feature location techniques using argouml-spl. In *Proceedings of the 13th International Workshop on Variability Modelling of Software-Intensive Systems*, page 16. ACM, 2019.

Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wasowski. Cool features and tough decisions: a comparison of variability modeling approaches. In *Proceedings of the sixth international workshop on variability modeling of software-intensive systems*, VaMoS'12, pages 173–182, 2012. doi: 10.1145/2110147.2110167. URL https://doi.org/10.1145/2110147.2110167.

Veronika Dashuber and Michael Philippsen. Static and dynamic dependency visualization in a layered software city. *SN Computer Science*, 3(6):1–18, 2022a.

Veronika Dashuber and Michael Philippsen. Trace visualization within the software city metaphor: Controlled experiments on program comprehension. *Information and Software Technology*, 150:106989, 2022b.

Veronika Dashuber, Michael Philippsen, and Johannes Weigend. A layered software city for dependency visualization. In *VISIGRAPP (3: IVAPP)*, pages 15–26, 2021.

Merijn de Jonge. Build-level components. *IEEE Transactions on Software Engineering*, 31(7): 588–600, 2005.

Stephan Diehl. *Software visualization: visualizing the structure, behaviour, and evolution of software*. Springer Science & Business Media, 2007.

Christian Dietrich, Reinhard Tartler, Wolfgang Schröder-Preikschat, and Daniel Lohmann. A robust approach for variability extraction from the Linux build system. In *Proceedings of the 16th International Software Product Line Conference-Volume 1*, pages 21–30, 2012.

Nicolas Dintzner, Arie Van Deursen, and Martin Pinzger. Fever: Extracting feature-oriented changes from commits. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 85–96, 2016.

Nicolas Dintzner, Arie van Deursen, and Martin Pinzger. Analysing the Linux kernel feature model changes using FMDiff. *Software & Systems Modeling*, 16(1):55–76, 2017.

Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. Feature location in source code: A taxonomy and survey. *Journal of Software: Evolution and Process*, 25(1):53–95, 2013. doi: 10.1002/smr.567. URL https://doi.org/10.1002/smr.567.

John Autor Domingue. *Software visualization: Programming as a multimedia experience*. MIT press, 1998.

Slawomir Duszynski and Martin Becker. Recovering variability information from the source code of similar software products. In *2012 Third International Workshop on Product LinE Approaches in Software Engineering (PLEASE)*, pages 37–40. IEEE, 2012.

Christof Ebert and Michel Smouts. Tricks and traps of initiating a product line concept in existing products. In *25th International Conference on Software Engineering, 2003. Proceedings.*, pages 520–525. IEEE, 2003.

Andrew David Eisenberg and Kris De Volder. Dynamic feature traces: Finding features in unfamiliar code. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 337–346. IEEE, 2005.

Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. Analysing the Kconfig Semantics and Its Analysis Tools. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, pages 45–54, 2015.

Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. An Empirical Study of Configuration Mismatches in Linux. In *Proceedings of the 21st International Systems and Software Product Line Conference-Volume A*, pages 19–28, 2017.

Eduardo Faccin Vernier, Alexandru C. Telea, and Joao Comba. Quantitative comparison of dynamic treemaps for software evolution visualization. In *2018 IEEE Working Conference on Software Visualization (VISSOFT)*, pages 96–106, 2018. doi: 10.1109/VISSOFT.2018.00018.

J-M Favre. Preprocessors from an abstract point of view. In *Proceedings of WCRE'96: 4rd Working Conference on Reverse Engineering*, pages 287–296. IEEE, 1996.

Janet Feigenspan, Maria Papendieck, Christian Kästner, Mathias Frisch, and Raimund Dachselt. FeatureCommander: colorful# ifdef world. In *Proceedings of the 15th International Software Product Line Conference, Volume 2*, pages 1–2, 2011a.

Janet Feigenspan, Michael Schulze, Maria Papendieck, Christian Kästner, Raimund Dachselt, Veit Köppen, and Mathias Frisch. Using background colors to support program comprehension in software product lines. In *15th Annual Conference on Evaluation & Assessment in Software Engineering (EASE 2011)*, pages 66–75. IET, 2011b.

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(4):699–745, 2013.

Dror G Feitelson. Using students as experimental subjects in software engineering research–a review and discussion of the evidence. *arXiv preprint arXiv:1512.08409*, 2015.

Wolfram Fenske and Sandro Schulze. Code smells revisited: A variability perspective. In *Proceedings of the Ninth International Workshop on Variability Modelling of Software-intensive Systems*, pages 3–10, 2015.

David Fernandez-Amoros, Ruben Heradio, Christoph Mayr-Dorn, and Alexander Egyed. A Kconfig translation to logic with one-way validation system. In *Proceedings of the 23rd International Systems and Software Product Line Conference-Volume A*, pages 303–308, 2019.

Eduardo Figueiredo, Nelio Cacho, Claudio Sant'Anna, Mario Monteiro, Uira Kulesza, Alessandro Garcia, Sergio Soares, Fabiano Ferrari, Safoora Khan, Fernando Castor Filho, et al. Evolving software product lines with aspects. In *2008 ACM/IEEE 30th International Conference on Software Engineering*, pages 261–270. IEEE, 2008.

Florian Fittkau, Alexander Krause, and Wilhelm Hasselbring. Hierarchical software landscape visualization for system comprehension: A controlled experiment. In *2015 IEEE 3rd Working Conference on Software Visualization (VISSOFT)*, pages 36–45. IEEE, 2015.

Florian Fittkau, Alexander Krause, and Wilhelm Hasselbring. Software landscape and application visualization for system comprehension with ExplorViz. *Information and software technology*, 87:259–277, 2017.

Nuno Flores, Diana Soares, Helder Ferreira, and Marco Rodrigues. Hotspotter: a javaml-based approach to discover framework's hotspots. *Proc. XATA*, 2005.

Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

Patrick Franz, Thorsten Berger, Ibrahim Fayaz, Sarah Nadi, and Evgeny Groshev. ConfigFix: Interactive configuration conflict resolution for the Linux kernel. In *43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. ACM, 2021.

Eric Freeman, Elisabeth Robson, Bert Bates, and Kathy Sierra. *Head First Design Patterns*. O'Reilly Media, Inc., 2008.

Cristina Gacek and Michalis Anastasopoules. Implementing product line variabilities. In *Proceedings of the 2001 Symposium on Software Reusability: Putting Software Reuse in Context*, SSR '01, pages 109–117. ACM, 2001. doi: 10.1145/375212.375269. URL https://doi.org/10.1145/375212.375269.

Harald C Gall and Michele Lanza. Software evolution: analysis and visualization. In *Proceedings of the 28th international conference on Software engineering*, pages 1055–1056, 2006.

Matthias Galster. Variability-intensive software systems: Product lines and beyond. In *Proceedings of the 13th International Workshop on Variability Modelling of Software-Intensive Systems*, VaMoS '19, pages 1–1. ACM, 2019. doi: 10.1145/3302333.3302336. URL https://doi.org/10.1145/3302333.3302336.

Matthias Galster, Danny Weyns, Dan Tofan, Bartosz Michalik, and Paris Avgeriou. Variability in software systems — a systematic literature review. *IEEE Transactions on Software Engineering*, 40(3):282–306, 2013. doi: 10.1109/TSE.2013.56. URL https://doi.org/10.1109/TSE.2013.56.

Matthias Galster, Uwe Zdun, Danny Weyns, Rick Rabiser, Bo Zhang, Michael Goedicke, and Gilles Perrouin. Variability and complexity in software design: Towards a research agenda. *ACM SIGSOFT Software Engineering Notes*, 41(6):27–30, 2017.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In Oscar M. Nierstrasz, editor, *ECOOP' 93 — Object-Oriented Programming*, pages 406–431, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, and Design Patterns. *Elements of reusable object-oriented software*, volume 99. Addison-Wesley Reading, Massachusetts, 1995.

Paul Gazzillo. Kmax: Finding all configurations of kbuild makefiles statically. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 279–290, 2017.

Sebastian Götz, Claas Wilke, Sebastian Cech, and Uwe Aßmann. Architecture and mechanisms of energy auto-tuning. In *Sustainable ICTs and Management Systems for Green Computing*, pages 45–73. IGI Global, 2012.

Sebastian Götz, Julian Mendez, Veronika Thost, and Anni-Yasmin Turhan. Owl 2 reasoning to detect energy-efficient software variants from context. Citeseer, 2013.

Orla Greevy, Michele Lanza, and Christoph Wysseier. Visualizing feature interaction in 3-d. In *3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 1–6. IEEE, 2005.

Martin L Griss. Implementing product-line features by composing aspects. In *Software Product Lines*, pages 271–288. Springer, 2000.

Axel Halin, Alexandre Nuttinck, Mathieu Acher, Xavier Devroey, Gilles Perrouin, and Patrick Heymans. Yo variability! jhipster: a playground for web-apps analyses. In *Proceedings of the Eleventh International Workshop on Variability Modelling of Software-intensive Systems*, pages 44–51, 2017.

Muhammad Hammad, Hamid Abdul Basit, Stan Jarzabek, and Rainer Koschke. A systematic mapping study of clone visualization. *Computer Science Review*, 37:100266, 2020.

Geoffrey Hecht, Omar Benomar, Romain Rouvoy, Naouel Moha, and Laurence Duchien. Tracking the software quality of android applications along their evolution (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 236–247. IEEE, 2015.

Stefan Hengelein. Analyzing the Internal Consistency of the Linux KConfig Model. Master's thesis, University of Erlangen, Dept. of Computer Science, 2015.

Kevlin Henney. The good, the bad, and the koyaanisqatsi. In *Proceedings of the Second Nordic Pattern Languages of Programs Conference, VikingPLoP*, volume 2003, pages 1–8, 2003.

Patrick Heymans, Quentin Boucher, Andreas Classen, Arnaud Bourdoux, and Laurent Demonceau. A code tagging approach to software product line development. *International Journal on Software Tools for Technology Transfer*, 14(5):553–566, 2012.

Joshua Hibschman, Darren Gergle, Eleanor O'Rourke, and Haoqi Zhang. Isopleth: Supporting Sensemaking of Professional Web Applications to Create Readily Available Learning Experiences. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 26(3):1–42, 2019.

Rich Hilliard. On representing variation. In *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, ECSA '10, page 312–315. ACM, 2010. doi: 10.1145/1842752.1842810. URL https://doi.org/10.1145/1842752.1842810.

Adrian Hoff, Lea Gerling, and Christoph Seidl. Utilizing Software Architecture Recovery to Explore Large-Scale Software Systems in Virtual Reality. In *2022 Working Conference on Software Visualization (VISSOFT)*. IEEE, 2022.

Danny Holten, Roel Vliegen, and Jarke J Van Wijk. Visual realism for the visualization of software metrics. In *3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 1–6. IEEE, 2005.

Weidong Huang, Peter Eades, and Seok-Hee Hong. Measuring effectiveness of graph visualizations: A cognitive load perspective. *Information Visualization*, 8(3):139–152, 2009.

Claus Hunsen, Bo Zhang, Janet Siegmund, Christian Kästner, Olaf Leßenich, Martin Becker, and Sven Apel. Preprocessor-based variability in open-source and industrial software systems: An empirical study. *Empirical Software Engineering*, 21(2):449–482, 2016.

Samuel Huppe, Mohamed Aymen Saied, and Houari Sahraoui. Mining complex temporal api usage patterns: an evolutionary approach. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 274–276. IEEE, 2017.

Emanuel Irrazábal, Juan Andrés Carruthers, and Juan Alberto Pinto Oppido. Modelo para curaduría de proyectos software de fuente abierta para estudios empíricos en ingeniería de software. In *XXIII Workshop de Investigadores en Ciencias de la Computación (WICC 2021, Chilecito, La Rioja)*, 2021.

Ayelet Israeli and Dror G Feitelson. The linux kernel as a case study in software evolution. *Journal of Systems and Software*, 83(3):485–501, 2010.

Ivar Jacobson, Martin Griss, and Patrik Jonsson. *Software reuse: architecture process and organization for business success*, volume 285. acm Press New York, 1997.

Yujuan Jiang, Bram Adams, and Daniel M German. Will my patch make it? and how fast? case study on the linux kernel. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 101–110. IEEE, 2013.

Isabel John, Jaejoon Lee, and Dirk Muthig. Separation of variability dimension and development dimension. In *Proocedings of the 1st International Workshop on Variability Modelling of Software-Intensive Systems*, VaMoS '07, pages 45–49, 2007.

Dennis Kafura and Sallie Henry. Software quality metrics based on interconnectivity. *Journal of Systems and Software*, 2(2):121–131, 1981.

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. Virtual separation of concerns-toward preprocessors 2.0/von christian kästner. 2010.

Christian Kästner. Kconfigreader. https://github.com/ckaestne/kconfigreader, 2016. Last access 24.02.2021.

Christian Kastner, Sven Apel, and Don Batory. A case study implementing features using aspectj. In *11th International Software Product Line Conference (SPLC 2007)*, pages 223–232. IEEE, 2007.

Christian Kästner, Salvador Trujillo, and Sven Apel. Visualizing Software Product Line Variabilities in Source Code. In *SPLC (2)*, pages 303–312, 2008.

Christian Kästner, Paolo G Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pages 805–824, 2011.

Christian Kästner, Klaus Ostermann, and Sebastian Erdweg. A variability-aware module system. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 773–792, 2012.

Andy Kenner, Christian Kästner, Steffen Haase, and Thomas Leich. Typechef: Toward type checking# ifdef variability in c. In *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development*, pages 25–32, 2010.

Brian W Kernighan and Dennis M Ritchie. *The C programming language*. Pearson Educación, 1988.

Raees Ahmad Khan, Khurram Mustafa, and Syed I Ahson. An empirical validation of object oriented design quality metrics. *Journal of King Saud University-Computer and Information Sciences*, 19:1–16, 2007.

Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European conference on object-oriented programming*, pages 220–242. Springer, 1997.

Barbara Kitchenham and Shari Lawrence Pfleeger. Software quality: the elusive target [special issues section]. *IEEE software*, 13(1):12–21, 1996.

Claire Knight and Malcolm Munro. Comprehension with [in] virtual environment visualisations. In *Proceedings Seventh International Workshop on Program Comprehension*, pages 4–11. IEEE, 1999.

Claire Knight and Malcolm Munro. Virtual but visible software. In *2000 IEEE Conference on Information Visualization. An International Conference on Computer Visualization and Graphics*, pages 198–205. IEEE, 2000.

Amy J Ko, Thomas D LaToza, and Margaret M Burnett. A practical guide to controlled experiments of software engineering tools with human participants. *Empirical Software Engineering*, 20(1):110–141, 2015.

Rainer Koschke. Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey. *Journal of Software Maintenance and Evolution: Research and Practice*, 15(2):87–109, 2003.

Julian Kratt, Hendrik Strobelt, and Oliver Deussen. Improving Stability and Compactness in Street Layout Visualizations. In *VMV*, pages 285–292, 2011.

Sebastian Krieter, Reimar Schröter, Thomas Thüm, Wolfram Fenske, and Gunter Saake. Comparing algorithms for efficient feature-model slicing. In *Proceedings of the 20th International Systems and Software Product Line Conference*, pages 60–64, 2016.

Christian Kröher, Sascha El-Sharkawy, and Klaus Schmid. Kernelhaven: an open infrastructure for product line analysis. In *Proceedings of the 22nd International Systems and Software Product Line Conference-Volume 2*, pages 5–10, 2018a.

Christian Kröher, Sascha El-Sharkawy, and Klaus Schmid. Kernelhaven: an experimentation workbench for analyzing software product lines. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 73–76, 2018b.

Christian Kröher, Lea Gerling, and Klaus Schmid. Identifying the intensity of variability changes in software product line evolution. In *Proceedings of the 22nd International Systems and Software Product Line Conference-Volume 1*, pages 54–64, 2018c.

Christian Kröher, Moritz Flöter, Lea Gerling, and Klaus Schmid. Incremental software product line verification-a performance analysis with dead variable code. *Empirical Software Engineering*, 27(3):1–41, 2022.

Philippe Kruchten, Robert L Nord, and Ipek Ozkaya. Technical debt: From metaphor to theory and practice. *Ieee software*, 29(6):18–21, 2012.

Charles Krueger and Paul Clements. Systems and software product line engineering with biglever software gears. In *Proceedings of the 17th International Software Product Line Conference co-located workshops*, pages 136–140, 2013.

Charles W Krueger. Software reuse. *ACM Computing Surveys (CSUR)*, 24(2):131–183, 1992.

Jacob Krüger, Wanzi Gu, Hui Shen, Mukelabai Mukelabai, Regina Hebig, and Thorsten Berger. Towards a better understanding of software features and their characteristics: A case study of marlin. In *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems*, VaMoS '18, pages 105–112, 2018. doi: 10.1145/3168365.3168371.

Jacob Krüger, Thorsten Berger, and Thomas Leich. Features and how to find them: A survey of manual feature location. *Software Engineering for Variability Intensive Systems*, pages 153–172, 2019a.

Jacob Krüger, Mukelabai Mukelabai, Wanzi Gu, Hui Shen, Regina Hebig, and Thorsten Berger. Where is My Feature and What is it About? A Case Study on Recovering Feature Facets. *Journal of Systems and Software*, 152:239–253, 2019b. doi: 10.1016/j.jss.2019.01.057. URL https://doi.org/10.1016/j.jss.2019.01.057.

Johannes Lampel, Sascha Just, Sven Apel, and Andreas Zeller. When life gives you oranges: detecting and diagnosing intermittent job failures at mozilla. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1381–1392, 2021.

Michael Larabel. The linux kernel enters 2020 at 27.8 million lines in git but with less developers for 2019. https://www.phoronix.com/scan.php?page=news_item&px=Linux-Git-Stats-EOY2019, 2020.

Kung-Kiu Lau and Tauseef Rana. A taxonomy of software composition mechanisms. In *2010 36th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 102–110. IEEE, 2010.

Duc Le, Eric Walkingshaw, and Martin Erwig. # ifdef confirmed harmful: Promoting understandable software variation. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 143–150. IEEE, 2011.

Duc Minh Le, Hyesun Lee, Kyo Chul Kang, and Lee Keun. Validating consistency between a feature model and its implementation. In *International Conference on Software Reuse*, pages 1–16. Springer, 2013.

Daniel Le Berre and Anne Parrain. The sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, 7(2-3):59–64, 2010. doi: 10.3233/SAT190075.

Valentina Lenarduzzi, Alberto Sillitti, and Davide Taibi. A survey on code analysis tools for software maintenance prediction. In *International Conference in Software Engineering for Defence Applications*, pages 165–175. Springer, 2018.

Valentina Lenarduzzi, Francesco Lomio, Heikki Huttunen, and Davide Taibi. Are sonarqube rules inducing bugs? In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 501–511. IEEE, 2020.

Luc Lesoil, Mathieu Acher, Arnaud Blouin, and Jean-Marc Jézéquel. Deep software variability: Towards handling cross-layer configuration. In *15th International Working Conference on Variability Modelling of Software-Intensive Systems*, pages 1–8, 2021a.

Luc Lesoil, Mathieu Acher, Arnaud Blouin, and Jean-Marc Jézéquel. The interaction between inputs and configurations fed to software systems: an empirical study. *arXiv preprint arXiv:2112.07279*, 2021b.

Luc Lesoil, Mathieu Acher, Xhevahire Tërnava, Arnaud Blouin, and Jean-Marc Jézéquel. The interplay of compile-time and run-time options for performance prediction. In *Proceedings of the 25th ACM International Systems and Software Product Line Conference-Volume A*, pages 100–111, 2021c.

Li Li, Jabier Martinez, Tewfik Ziadi, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Mining families of android applications for extractive spl adoption. In *Proceedings of the 20th International Systems and Software Product Line Conference*, pages 271–275, 2016.

Zengyang Li, Paris Avgeriou, and Peng Liang. A systematic mapping study on technical debt and its management. *Journal of Systems and Software*, 101:193–220, 2015.

Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 105–114. ACM, 2010.

Lukas Linsbauer, Stefan Fischer, Gabriela Karoline Michelon, Wesley KG Assunção, Paul Grünbacher, Roberto Erick Lopez-Herrejon, and Alexander Egyed. Systematic software reuse with

automated extraction and composition for clone-and-own. *Handbook of Re-Engineering Software Intensive Systems into Software Product Lines, Roberto Erick Lopez-Herrejon, Jabier Martinez, Tewfik Ziadi, Mathieu Acher, Wesley KG Assunção, and Silvia Regina Vergilio (Eds.). Springer*, 2022.

Roberto Erick Lopez-Herrejon, Sheny Illescas, and Alexander Egyed. A systematic mapping study of information visualization for software product line engineering. *Journal of software: evolution and process*, 30(2):e1912, 2018.

Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wasowski. Evolution of the Linux kernel variability model. In *International Conference on Software Product Lines*, pages 136–150. Springer, 2010.

Angela Lozano. An overview of techniques for detecting software variability concepts in source code. In *International Conference on Conceptual Modeling*, ER '11, pages 141–150. Springer, 2011. doi: 10.1007/978-3-642-24574-9. URL https://doi.org/10.1007/978-3-642-24574-9.

Ivan do Carmo Machado, Alcemir Rodrigues Santos, Yguaratã Cerqueira Cavalcanti, Eduardo Gomes Trzan, Marcio Magalhães de Souza, and Eduardo Santana de Almeida. Low-level variability support for web-based software product lines. In *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems*, pages 1–8, 2014.

Jabier Martinez, Wesley KG Assunção, and Tewfik Ziadi. Espla: A catalog of extractive spl adoption case studies. In *Proceedings of the 21st International Systems and Software Product Line Conference-Volume B*, pages 38–41. ACM, 2017a.

Jabier Martinez, Tewfik Ziadi, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Bottom-up technologies for reuse: automated extractive adoption of software product lines. In *Proceedings of the 39th International Conference on Software Engineering Companion*, pages 67–70. IEEE Press, 2017b.

Jabier Martinez, Nicolas Ordoñez, Xhevahire Tërnava, Tewfik Ziadi, Jairo Aponte, Eduardo Figueiredo, and Marco Tulio Valente. Feature location benchmark with argouml spl. In *Proceeedings of the 22nd International Conference on Systems and Software Product Line-Volume 1*, pages 257–263. ACM, 2018.

Antonio Martini and Jan Bosch. The danger of architectural technical debt: Contagious debt and vicious circles. In *2015 12th Working IEEE/IFIP Conference on Software Architecture*, pages 1–10. IEEE, 2015.

Johan Martinson, Herman Jansson, Mukelabai Mukelabai, Thorsten Berger, Alexandre Bergel, and Truong Ho-Quang. HAnS: IDE-based editing support for embedded feature annotations. In *Proceedings of the 25th ACM International Systems and Software Product Line Conference-Volume B*, pages 28–31, 2021.

Guillaume Maudoux and Kim Mens. Lessons and pitfalls in building firefox with tup. In Anne Etien, editor, *Proceedings of the Seminar Series on Advanced Techniques & Tools for Software Evolution (SATTOSE 2019), Bolzano, Italy, July 8-10 Day, 2019*, volume 2510 of

CEUR Workshop Proceedings. CEUR-WS.org, 2019. URL http://ceur-ws.org/Vol-2510/sattose2019_paper_2.pdf.

Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4): 308–320, 1976.

Shane McIntosh, Bram Adams, Thanh HD Nguyen, Yasutaka Kamei, and Ahmed E Hassan. An empirical study of build maintenance effort. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 141–150. IEEE, 2011.

Flávio Medeiros, Márcio Ribeiro, Rohit Gheyi, Sven Apel, Christian Kästner, Bruno Ferreira, Luiz Carvalho, and Baldoino Fonseca. Discipline matters: Refactoring of preprocessor directives in the# ifdef hell. *IEEE Transactions on Software Engineering*, 44(5):453–469, 2017.

Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. *Mastering Software Variability with FeatureIDE*. Springer, 2017.

Jens Meinicke, Chu-Pan Wong, Christian Kästner, and Gunter Saake. Understanding differences among executions with variational traces. *arXiv preprint arXiv:1807.03837*, 2018.

Tom Mens, Michel Wermelinger, Stéphane Ducasse, Serge Demeyer, Robert Hirschfeld, and Mehdi Jazayeri. Challenges in software evolution. In *Eighth International Workshop on Principles of Software Evolution (IWPSE'05)*, pages 13–22. IEEE, 2005.

Andreas Metzger and Klaus Pohl. Software product line engineering and variability management: achievements and challenges. In *Future of Software Engineering Proceedings*, pages 70–84. Association for Computing Machinery, New York, NY, USA, 2014. ISBN 9781450328654. doi: 10.1145/2593882.2593888. URL https://doi.org/10.1145/2593882.2593888.

Mira Mezini and Klaus Ostermann. Variability management with feature-oriented programming and aspects. *ACM SIGSOFT Software Engineering Notes*, 29(6):127–136, 2004.

Gabriela K Michelon, Lukas Linsbauer, Wesley KG Assunção, Stefan Fischer, and Alexander Egyed. A Hybrid Feature Location Technique for Re-engineering Single Systems into Software Product Lines. In *15th International Working Conference on Variability Modelling of Software-Intensive Systems*, pages 1–9, 2021a.

Gabriela K Michelon, Bruno Sotto-Mayor, Jabier Martinez, Aitor Arrieta, Rui Abreu, and Wesley KG Assunção. Spectrum-based feature localization: a case study using argouml. In *Proceedings of the 25th ACM International Systems and Software Product Line Conference-Volume A*, pages 126–130, 2021b.

Gabriela K. Michelon, Jabier Martinez, Bruno Sotto-Mayor, Aitor Arrieta, Wesley K.G. Assunção, Rui Abreu, and Alexander Egyed. Spectrum-based feature localization for families of systems. *Journal of Systems and Software*, page 111532, 2022. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2022.111532. URL https://www.sciencedirect.com/science/article/pii/S0164121222002084.

Gabriela Karoline Michelon, Lukas Linsbauer, Wesley KG Assunção, and Alexander Egyed. Comparison-based feature location in argouml variants:[challenge solution]. In *Proceedings of*

the 23rd International Systems and Software Product Line Conference-Volume A, page 17. ACM, 2019.

Roberto Minelli, Andrea Mocci, and Michele Lanza. I Know What You Did Last Summer - An Investigation of How Developers Spend Their Time. In *2015 IEEE 23rd International Conference on Program Comprehension*, pages 25–35. IEEE, 2015.

Sanjay Misra, Adewole Adewumi, Luis Fernandez-Sanz, and Robertas Damasevicius. A suite of object oriented cognitive complexity metrics. *IEEE Access*, 6:8782–8796, 2018.

Austin Mordahl, Jeho Oh, Ugur Koc, Shiyi Wei, and Paul Gazzillo. An empirical study of real-world variability bugs detected by variability-oblivious tools. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 50–61, 2019.

David Moreno-Lumbreras, Roberto Minelli, Andrea Villaverde, Jesus M Gonzalez-Barahona, and Michele Lanza. CodeCity: A comparison of on-screen and virtual reality. *Information and Software Technology*, page 107064, 2022.

David Moreno-Lumbreras, Roberto Minelli, Andrea Villaverde, Jesus M Gonzalez-Barahona, and Michele Lanza. Codecity: A comparison of on-screen and virtual reality. *Information and Software Technology*, 153:107064, 2023.

Brice Morin, Olivier Barais, Gregory Nain, and Jean-Marc Jézéquel. Taming dynamically adaptive systems using models and aspects. In *2009 IEEE 31st International Conference on Software Engineering*, pages 122–132. IEEE, 2009.

Johann Mortara, Xhevahire Tërnava, and Philippe Collet. symfinder: A Toolchain for the Identification and Visualization of Object-Oriented Variability Implementations. In *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume B*, SPLC '19, pages 5–8, New York, NY, USA, September 2019. Association for Computing Machinery. ISBN 978-1-4503-6668-7. doi: 10.1145/3307630.3342394. URL https://hal.archives-ouvertes.fr/hal-02342730.

Johann Mortara, Philippe Collet, and Anne-Marie Dery-Pinna. Visualization of Object-Oriented Variability Implementations as Cities. In *2021 Working Conference on Software Visualization (VISSOFT)*, pages 76–87, Luxembourg (virtual), Luxembourg, September 2021a. ISBN 978-1-6654-3144-6. doi: 10.1109/VISSOFT52517.2021.00017. URL https://hal.archives-ouvertes.fr/hal-03312487.

Johann Mortara, Philippe Collet, and Anne-Marie Dery-Pinna. Visualization of Object-Oriented Variability Implementations as Cities — Reproduction package, June 2021b. URL https://doi.org/10.5281/zenodo.5034199.

Johann Mortara, Xhevahire Tërnava, Philippe Collet, and Anne-Marie Dery-Pinna. Extending the Identification of Object-Oriented Variability Implementations using Usage Relationships. In *SPLC 2021 - 25th ACM International Systems and Software Product Line Conference*, volume Volume B, pages 1–8, Leicester, United Kingdom, September 2021c. ACM. doi: 10.1145/3461002.3473943. URL https://hal.archives-ouvertes.fr/hal-03284626.

Johann Mortara, Xhevahire Tërnava, Philippe Collet, Anne-Marie Pinna-Dery, Florian Ainadou, Paul-Marie Djekinnou, and Djotiham Nabagou. Extending the Identification of Object-Oriented Variability Implementations using Usage Relationships — Reproduction Package, June 2021d. URL https://doi.org/10.5281/zenodo.4946730.

Johann Mortara, Philippe Collet, Anne-Marie Pinna-Dery, Patrick Anagonou, Guillaume Savornin, and Anton van der Tuijn. Customizable Visualization of Quality Metrics for Object-Oriented Variability Implementations - Artifact, June 2022. URL https://doi.org/10.5281/zenodo.6644634.

Richard Müller and Ulrich Eisenecker. A graph-based feature location approach using set theory. In *Proceedings of the 23rd International Systems and Software Product Line Conference-Volume A*, pages 88–92, 2019.

Richard Müller, Pascal Kovacs, Jan Schilbach, Ulrich W Eisenecker, Dirk Zeckzer, and Gerik Scheuermann. A structured approach for conducting a series of controlled experiments in software visualization. In *2014 International Conference on Information Visualization Theory and Applications (IVAPP)*, pages 204–209. IEEE, 2014.

Sarah Nadi. *Variability Anomalies in Software Product Lines*. PhD thesis, University of Waterloo, 2014.

Sarah Nadi. Software product lines, 2015. URL http://stg-tud.github.io/sedc/Lecture/ss15/SPL.pdf.

Sarah Nadi and Ric Holt. Make it or break it: Mining anomalies from Linux Kbuild. In *2011 18th Working Conference on Reverse Engineering*, pages 315–324. IEEE, 2011.

Sarah Nadi and Ric Holt. Mining Kbuild to detect variability anomalies in Linux. In *2012 16th European Conference on Software Maintenance and Reengineering*, pages 107–116. IEEE, 2012.

Sarah Nadi and Ric Holt. The Linux kernel: A case study of build system variability. *Journal of Software: Evolution and Process*, 26(8):730–746, 2014.

Sarah Nadi, Christian Dietrich, Reinhard Tartler, Richard C Holt, and Daniel Lohmann. Linux variability anomalies: What causes them and how do they get fixed? In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 111–120. IEEE, 2013.

ThanhVu Nguyen and KimHao Nguyen. Using Symbolic Execution to Analyze Linux KBuild Makefiles. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 712–716. IEEE, 2020.

Linda Northrop, Paul Clements, Felix Bachmann, John Bergey, Gary Chastek, Sholom Cohen, Patrick Donohoe, Lawrence Jones, Robert Krut, Reed Little, et al. A framework for software product line practice, version 5.0. *SEI.–2007–http://www. sei. cmu. edu/productlines/index. html*, 2007.

Joseph D Novak and Alberto J Cañas. The theory underlying concept maps and how to construct them. *Florida Institute for Human and Machine Cognition*, 1, 2006.

Alberto S Nuñez-Varela, Héctor G Pérez-Gonzalez, Francisco E Martínez-Perez, and Carlos Soubervielle-Montalvo. Source code metrics: A systematic mapping study. *Journal of Systems and Software*, 128:164–197, 2017.

Jeho Oh, Paul Gazzillo, Don Batory, Marijn Heule, and Maggie Myers. Uniform sampling from kconfig feature models. *The University of Texas at Austin, Department of Computer Science, Tech. Rep. TR-19*, 2, 2019.

Zakaria Ournani, Romain Rouvoy, Pierre Rust, and Joel Penhoat. Tales from the code# 1: The effective impact of code refactorings on software energy consumption. In *ICSOFT 2021-16th International Conference on Software Technologies*, 2021.

Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in linux device drivers. *ACM SigOps Operating Systems Review*, 42(4):247–260, 2008.

Jevgenija Pantiuchina, Michele Lanza, and Gabriele Bavota. Improving code: The (mis) perception of quality metrics. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 80–91. IEEE, 2018.

Carlos Parra, Xavier Blanc, Anthony Cleve, and Laurence Duchien. Unifying design and runtime software adaptation using aspect models. *Science of Computer Programming*, 76(12):1247–1260, 2011.

Leonardo Passos, Jianmei Guo, Leopoldo Teixeira, Krzysztof Czarnecki, Andrzej Wasowski, and Paulo Borba. Coevolution of variability models and related artifacts: A case study from the Linux kernel. In *Proceedings of the 17th International Software Product Line Conference*, pages 91–100, 2013.

Leonardo Passos, Rodrigo Queiroz, Mukelabai Mukelabai, Thorsten Berger, Sven Apel, Krzysztof Czarnecki, and Jesus Padilla. A study of feature scattering in the linux kernel. *IEEE Transactions on Software Engineering*, 2018.

Jeremy R Pate, Robert Tairas, and Nicholas A Kraft. Clone evolution: a systematic review. *Journal of software: Evolution and Process*, 25(3):261–283, 2013.

Thomas Patzke and Dirk Muthig. Product line implementation technologies. programming language view. Technical report, Fraunhofer IESE, 2002. URL http://publica.fraunhofer.de/dokumente/N-14684.html.

Norman Peitek, Sven Apel, Chris Parnin, André Brechmann, and Janet Siegmund. Program Comprehension and Code Complexity Metrics: An fMRI Study. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 524–536. IEEE, 2021.

Luca Pellegrini, Andrea Alexander Janes, and Davide Taibi. On the fault proneness of sonarqube technical debt violations. an empirical study. *Ph. D. dissertation*, 2018.

Francisca Pérez, Jorge Echeverría, Raúl Lapeña, and Carlos Cetina. Comparing manual and automated feature location in conceptual models: A controlled experiment. *Information and Software Technology*, 125:106337, 2020.

Federico Pfahler, Roberto Minelli, Csaba Nagy, and Michele Lanza. Visualizing Evolving Software Cities. In *2020 Working Conference on Software Visualization (VISSOFT)*, pages 22–26. IEEE, 2020.

Robert Pienta, James Abello, Minsuk Kahng, and Duen Horng Chau. Scalable graph exploration and visualization: Sensemaking challenges and opportunities. In *2015 International conference on Big Data and smart computing (BIGCOMP)*, pages 271–278. IEEE, 2015.

Gustavo Pinto, Francisco Soares-Neto, and Fernando Castor. Refactoring for energy efficiency: A reflection on the state of the art. In *2015 IEEE/ACM 4th International Workshop on Green and Sustainable Software*, pages 29–35. IEEE, 2015.

Klaus Pohl, Günter Böckle, and Frank J van Der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer Science & Business Media, 2005.

Christian Prehofer. Feature-oriented programming: A fresh look at objects. In *European Conference on Object-Oriented Programming*, pages 419–443. Springer, 1997.

pure-systems GmbH. pure::variants, 2020. URL https://www.pure-systems.com/products/pure-variants-9.html.

Rick Rabiser. Feature modeling vs. decision modeling: History, comparison and perspectives. In *Proceedings of the 23rd International Systems and Software Product Line Conference-Volume B*, SPLC '19, pages 134–136. ACM, 2019. doi: 10.1145/3307630.3342399. URL https://doi.org/10.1145/3307630.3342399.

Ghulam Rasool and Zeeshan Arshad. A review of code smell mining techniques. *Journal of Software: Evolution and Process*, 27(11):867–895, 2015.

Martin P Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. Automated api property inference techniques. *IEEE Transactions on Software Engineering*, 39(5): 613–637, 2012.

Joe Rosen. *Symmetry in Science*. Springer, 1995.

Joseph Rosen. *Symmetry Rules: How Science and Nature are Founded on Symmetry*. Springer Science & Business Media, 2008.

Linda H Rosenberg and Lawrence E Hyatt. Software quality metrics for object-oriented environments. *Crosstalk journal*, 10(4):1–6, 1997.

Chanchal Kumar Roy and James R Cordy. A survey on software clone detection research. *Queen's School of Computing TR*, 541(115):64–68, 2007.

Julia Rubin and Marsha Chechik. A survey of feature location techniques. In *Domain Engineering*, pages 29–58. Springer, 2013.

Julia Rubin, Krzysztof Czarnecki, and Marsha Chechik. Managing cloned variants: a framework and experience. In *Proceedings of the 17th International Software Product Line Conference*, pages 101–110, 2013.

Marc-Oliver Rüdel, Johannes Ganser, and Rainer Koschke. A Controlled Experiment on Spatial Orientation in VR-based Software Cities. In *2018 IEEE Working Conference on Software Visualization (VISSOFT)*, pages 21–31. IEEE, 2018.

Andreas Ruprecht. Lightweight Extraction of Variability Information from Linux Makefiles. Master's thesis, Citeseer, 2015.

Mohamed Aymen Saied and Houari Sahraoui. A cooperative approach for combining client-based and library-based api usage pattern mining. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–10. IEEE, 2016.

Mohamed Aymen Saied, Erick Raelijohn, Edouard Batot, Michalis Famelis, and Houari Sahraoui. Towards assisting developers in api usage by automated recovery of complex temporal patterns. *Information and Software Technology*, 119:106213, 2020.

Alcemir Rodrigues Santos and Eduardo Santana de Almeida. Do# ifdef-based Variation Points Realize Feature Model Constraints? *ACM SIGSOFT Software Engineering Notes*, 40(6):1–5, 2015.

Anita Sarma, Larry Maccherone, Patrick Wagstrom, and James Herbsleb. Tesseract: Interactive visual exploration of socio-technical relationships in software development. In *2009 IEEE 31st International Conference on Software Engineering*, pages 23–33. IEEE, 2009.

Danilo Sato, Alfredo Goldman, and Fabio Kon. Tracking the evolution of object-oriented quality metrics on agile projects. In *International Conference on Extreme Programming and Agile Processes in Software Engineering*, pages 84–92. Springer, 2007.

Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented programming of software product lines. In *International Conference on Software Product Lines*, pages 77–91. Springer, 2010.

Reinhard Schauer, Sébastien Robitaille, Francois Martel, and Rudolf K Keller. Hot spot recovery in object-oriented software with inheritance and composition template methods. In *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360)*, pages 220–229. IEEE, 1999.

Willy Scheibel, Daniel Limberger, and Jürgen Döllner. Survey of treemap layout algorithms. In *Proceedings of the 13th International Symposium on Visual Information Communication and Interaction*, pages 1–9, 2020.

Syed Muhammad Ali Shah, Marco Torchiano, VETRO' ANTONIO, and Maurizio Morisio. Exploratory testing as a source of testing technical debt. *IT Professional IEEE Computer Society Digital Library*, page 25, 2013.

Steven She. Linux variability analysis tools. https://github.com/matachi/linux-variability-analysis-tools.exconfig, 2013. Last access 24.02.2021.

Steven She and Thorsten Berger. Formal semantics of the Kconfig language. *Technical note, University of Waterloo*, 24, 2010.

Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof Czarnecki. The Variability Model of The Linux Kernel. *VaMoS*, 10(10):45–51, 2010.

Ben Shneiderman. Tree visualization with tree-maps: 2-d space-filling approach. *ACM Transactions on graphics (TOG)*, 11(1):92–99, 1992.

Ben Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *The craft of information visualization*, pages 364–371. Elsevier, 2003.

Jonathan Sillito, Gail C Murphy, and Kris De Volder. Asking and answering questions during a programming change task. *IEEE Transactions on Software Engineering*, 34(4):434–451, 2008.

Susan Elliott Sim and Richard C Holt. The ramp-up problem in software projects: A case study of how software immigrants naturalize. In *Proceedings of the 20th international conference on Software engineering*, pages 361–370. IEEE, 1998.

Julio Sincero and Wolfgang Schröder-Preikschat. The Linux Kernel Configurator as a Feature Modeling Tool. In *SPLC (2)*, pages 257–260. Citeseer, 2008.

Julio Sincero, Horst Schirmeier, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. Is the Linux Kernel a Software Product Line? In *Proc. SPLC Workshop on Open Source Software and Product Lines*, 2007.

Julio Sincero, Reinhard Tartler, Daniel Lohmann, and Wolfgang Schröder-Preikschat. Efficient extraction and analysis of preprocessor-based variability. In *Proceedings of the ninth international conference on Generative programming and component engineering*, pages 33–42, 2010.

Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 38–45, 1986.

Larissa Rocha Soares, Jens Meinicke, Sarah Nadi, Christian Kästner, and Eduardo Santana de Almeida. Exploring feature interactions without specifications: A controlled experiment. *ACM SIGPLAN Notices*, 53(9):40–52, 2018.

Iuri Santos Souza, Ivan Machado, Carolyn Seaman, Gecynalda Gomes, Christina Chavez, Eduardo Santana de Almeida, and Paulo Masiero. Investigating variability-aware smells in spls: An exploratory study. In *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*, pages 367–376, 2019.

Marcel Steinbeck, Rainer Koschke, and Marc O Rudel. Comparing the evostreets visualization technique in two-and three-dimensional environments a controlled experiment. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 231–242. IEEE, 2019.

Frank Steinbrückner and Claus Lewerentz. Representing development history in software cities. In *Proceedings of the 5th international symposium on Software visualization*, pages 193–202, 2010.

Frank Steinbrückner and Claus Lewerentz. Understanding software evolution with software cities. *Information Visualization*, 12(2):200–216, 2013.

Igor Steinmacher, Marco Aurélio Graciotto Silva, and Marco Aurélio Gerosa. Barriers faced by newcomers to open source projects: a systematic review. In *IFIP International Conference on Open Source Systems*, pages 153–163. Springer, 2014.

Srdjan Stevanetic and Uwe Zdun. Software metrics for measuring the understandability of architectural structures: a systematic mapping study. In *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*, pages 1–14, 2015.

Margaret-Anne D Storey, Davor Čubranić, and Daniel M German. On the use of visualization to support awareness of human activities in software development: a survey and a framework. In *Proceedings of the 2005 ACM symposium on Software visualization*, pages 193–202, 2005.

Mikael Svahnberg, Jilles Van Gurp, and Jan Bosch. A taxonomy of variability realization techniques. *Software: Practice and experience*, 35(8):705–754, 2005.

Fuminobu Takeyama and Shigeru Chiba. Implementing feature interactions with generic feature modules. In *International Conference on Software Composition*, pages 81–96. Springer, 2013.

Reinhard Tartler. *Mastering variability challenges in Linux and related highly-configurable system software*. PhD thesis, 2013.

Reinhard Tartler, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Dead or alive: Finding zombie features in the Linux kernel. In *Proceedings of the First International Workshop on Feature-Oriented Software Development*, pages 81–86, 2009.

Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. Feature consistency in compile-time-configurable system software: Facing the Linux 10,000 feature problem. In *Proceedings of the sixth conference on Computer systems*, pages 47–60, 2011.

Reinhard Tartler, Julio Sincero, Christian Dietrich, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Revealing and repairing configuration inconsistencies in large-scale system software. *International Journal on Software Tools for Technology Transfer*, 14(5):531–551, 2012.

Xhevahire Tërnava and Philippe Collet. On the diversity of capturing variability at the implementation level. In *Proceedings of the 21st International Systems and Software Product Line Conference - Volume B*, SPLC '17, pages 81–88. ACM, 2017a. doi: 10.1145/3109729.3109733. URL https://doi.org/10.1145/3109729.3109733.

Xhevahire Tërnava and Philippe Collet. Tracing imperfectly modular variability in software product line implementation. In *International Conference on Software Reuse*, ICSR '17, pages 112–120. Springer, 2017b. doi: 10.1007/978-3-319-56856-0_8. URL https://doi.org/10.1007/978-3-319-56856-0_8.

Xhevahire Tërnava, Johann Mortara, and Philippe Collet. Identifying and visualizing variability in object-oriented variability-rich systems. In *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A*, SPLC '19, pages 231–243, New York, NY, USA, September 2019. Association for Computing Machinery. ISBN 978-1-4503-7138-4. doi: 10.1145/3336294.3336311. URL https://hal.archives-ouvertes.fr/hal-02339296.

Xhevahire Tërnava, Johann Mortara, Philippe Collet, and Daniel Le Berre. Identification and visualization of variability implementations in object-oriented variability-rich systems: a symmetry-based approach. *Journal of Automated Software Engineering*, 29:1–51, February 2022. doi: 10.1007/s10515-022-00329-x. URL https://hal.archives-ouvertes.fr/hal-03593967.

Alfredo R Teyseyre and Marcelo R Campo. An overview of 3D software visualization. *IEEE transactions on visualization and computer graphics*, 15(1):87–105, 2008.

Thomas Thüm. A BDD for Linux? the knowledge compilation challenge for variability. In *Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A-Volume A*, pages 1–6, 2020.

Pablo Trinidad, David Benavides, Amador Durán, Antonio Ruiz-Cortés, and Miguel Toro. Automated error analysis for the agilization of feature modeling. *Journal of Systems and Software*, 81(6):883–896, 2008.

Davide Paolo Tua, Roberto Minelli, and Michele Lanza. Voronoi evolving treemaps. In *2021 Working Conference on Software Visualization (VISSOFT)*, pages 1–5. IEEE, 2021.

C. Reid Turner, Alfonso Fuggetta, Luigi Lavazza, and Alexander L. Wolf. A conceptual basis for feature engineering. *Journal of Systems and Software*, 49(1):3–15, 1999. doi: 10.1016/S0164-1212(99)00062-X. URL https://doi.org/10.1016/S0164-1212(99)00062-X.

Juraj Vincur, Pavol Navrat, and Ivan Polasek. VR City: Software Analysis in Virtual Reality Environment. In *2017 IEEE international conference on software quality, reliability and security companion (QRS-C)*, pages 509–516. IEEE, 2017.

Markus Voelter and Iris Groher. Product line implementation using aspect-oriented and model-driven software development. In *11th International Software Product Line Conference (SPLC 2007)*, pages 233–242. IEEE, 2007.

Alexander Von Rhein, Alexander Grebhahn, Sven Apel, Norbert Siegmund, Dirk Beyer, and Thorsten Berger. Presence-condition simplification in highly configurable systems. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 178–188. IEEE, 2015.

Howard Wainer and Carl M Francolini. An empirical inquiry concerning human understanding of two-variable color maps. *The American Statistician*, 34(2):81–93, 1980.

Martin Walch, Rouven Walter, and Wolfgang Küchlin. Formal analysis of the Linux kernel configuration with SAT solving. In *Configuration Workshop*, pages 131–138, 2015.

Neil Walkinshaw, Marc Roper, and Murray Wood. Feature location and extraction using landmarks and barriers. In *2007 IEEE International Conference on Software Maintenance*, pages 54–63. IEEE, 2007.

Jan Waller, Christian Wulf, Florian Fittkau, Philipp Döhring, and Wilhelm Hasselbring. Synchrovis: 3d visualization of monitoring traces in the city metaphor for analyzing concurrency. In *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*, pages 1–4. IEEE, 2013.

Markus Weninger, Lukas Makor, and Hanspeter Mössenböck. Memory Cities: Visualizing Heap Memory Evolution Using the Software City Metaphor. In *2020 Working Conference on Software Visualization (VISSOFT)*, pages 110–121. IEEE, 2020.

Richard Wettel and Michele Lanza. Visualizing software systems as cities. In *2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 92–99. IEEE, 2007.

Richard Wettel and Michele Lanza. CodeCity: 3D visualization of large-scale software. In *Companion of the 30th international conference on Software engineering*, pages 921–922, 2008a.

Richard Wettel and Michele Lanza. Visual exploration of large-scale system evolution. In *2008 15th Working Conference on Reverse Engineering*, pages 219–228. IEEE, 2008b.

Richard Wettel and Michele Lanza. Visually localizing design problems with disharmony maps. In *Proceedings of the 4th ACM Symposium on Software Visualization*, pages 155–164, 2008c.

Richard Wettel, Michele Lanza, and Romain Robbes. Empirical validation of codecity: A controlled experiment. Technical report, Università della Svizzera italiana, 2010.

Richard Wettel, Michele Lanza, and Romain Robbes. Software systems as cities: A controlled experiment. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 551–560, 2011.

Daniele Wolfart, Wesley Klewerton Guez Assunção, and Jabier Martinez. Variability Debt: Characterization, Causes and Consequences. In *XX Brazilian Symposium on Software Quality*, pages 1–10, 2021.

Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E Hassan, and Shanping Li. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering*, 44(10):951–976, 2017.

Rebecca Yates, Norah Power, and Jim Buckley. Characterizing the transfer of program comprehension in onboarding: an information-push perspective. *Empirical Software Engineering*, 25 (1):940–995, 2020.

Christoph Zengler and Wolfgang Küchlin. Encoding the Linux kernel configuration in propositional logic. In *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI 2010) Workshop on Configuration*, volume 2010, pages 51–56, 2010.

Bo Zhang, Martin Becker, Thomas Patzke, Krzysztof Sierszecki, and Juha Erik Savolainen. Variability evolution and erosion in industrial product lines: a case study. In *Proceedings of the 17th International Software Product Line Conference*, pages 168–177. ACM, 2013.

Wei Zhang, Haiyan Zhao, and Hong Mei. A propositional logic-based method for verification of feature models. In *International Conference on Formal Engineering Methods*, pages 115–130. Springer, 2004.

Liping Zhao. Patterns, symmetry, and symmetry breaking. *Communications of the ACM*, 51(3): 40–46, 2008. doi: 10.1145/1325555.1325564. URL https://dl.acm.org/doi/pdf/10.1145/1325555.1325564.

Liping Zhao and James Coplien. Understanding symmetry in object-oriented languages. *Journal of Object Technology*, 2(5):123–134, 2003.

Liping Zhao and James O Coplien. Symmetry in class and type hierarchy. In *Proceedings of the Fortieth International Conference on Tools Pacific: Objects for internet, mobile and embedded applications*, pages 181–189. Australian Computer Society, Inc., 2002.

Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. Mapo: Mining and recommending api usage patterns. In *European Conference on Object-Oriented Programming*, pages 318–343. Springer, 2009.

Zixiao Zhu, Yanzhen Zou, Bing Xie, Yong Jin, Zeqi Lin, and Lu Zhang. Mining api usage examples from test code. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 301–310. IEEE, 2014.

Tewfik Ziadi, Luz Frias, Marcos Aurélio Almeida da Silva, and Mikal Ziane. Feature identification from the source code of product variants. In *2012 16th European Conference on Software Maintenance and Reengineering*, pages 417–422. IEEE, 2012.

List of Figures

List of Tables

List of Listings

List of Definitions

Appendices

APPENDIX A

State-of-the-Art anomalies in the Linux kernel build system

In this appendix, we enumerate the 25 definitions of anomalies in the Linux kernel build system presented in the five selected state-of-the-art papers characterizing them (*cf.* Section 3.5). The definitions presented here are directly extracted from the papers, although some sentences may be added to reproduce their context. We mark with a * anomalies that are naturally inconsistent as they are directly extracted from papers. Characterizing these inconsistencies is done by instantiating the anomalies in our models (*cf.* Section 8.2).

A.1 CPP internal consistency by Sincero et al. [2010]

Sincero et al. [2010] formalize CPP directives using propositional logic and propose a framework, UNDERTAKER, to automate the derivation of presence conditions from `ifdef` directives. They define lines of code in `ifdef` blocks as *blocks* and define for a block b_i the

Presence Condition [Sincero et al., 2010]:

$$\mathcal{PC}(b_i) = expression(b_i) \wedge noPredecessors(b_i) \wedge parent(b_i)$$

with

$expression(b_i)$ Given a block b_i, the function $expression(b_i)$ returns the logical expression as specified in the block declaration.
Example. For the block `B1` in Figure 2.2, $expression(b_1)$ returns: $A \vee B \vee C$.

$parent(b_i)$ Given b_i, $parent(b_i)$ returns the logical variable that represents the selection of its parent. If the block is not nested in any other block, then the result is always *true*.
Example. For the block `B3` in Figure 2.2, $parent(b_3)$ returns: b_1.

$noPredecessors(b_i)$ Given b_i, $noPredecessors(b_i)$ returns the negation of the disjunction of all its predecessors (logical variables representing blocks) in an if-group.
Example. For the block `B4` of Figure 2.2, $noPredecessors(b_4)$ returns: $\neg(b_2 \vee b_3)$.

The authors then give a definition of dead defect:

Anomaly 1 (Dead block [Sincero et al., 2010])**.** A block is dead if:

$$\neg satisfiable(\mathcal{K} \wedge \mathcal{C} \wedge Block_N)$$

with $\mathcal{K}$ and $\mathcal{C}$ the propositional formulas representing the *problem space constraints* (*i.e.*, KCONFIG space) and *solution space constraints* (*i.e.*, Make space) respectively. $satisfiable()$ represents the boolean satisfiability problem[1].

Relying on the expression of the presence condition, the authors finally define two levels of consistency to express this definition of dead defect.

Anomaly 2 (Internal consistency [Sincero et al., 2010])**.** Internal consistency is defined as *checking for each block of a compilation unit if it is selectable by at least one valid configuration*. This property is checked with $satisfiable(C_u \wedge b_i)$ which, expanded using the definition of C_u, gives:

$$satisfiable\left(\left(\bigwedge_{i=1..m} b_i \leftrightarrow \mathcal{PC}(b_i)\right) \wedge b_i\right)$$

Anomaly 3 (External consistency [Sincero et al., 2010])**.** External consistency is defined as *checking for each block of a compilation unit if it is selectable by at least one valid configuration*. This property is checked with $satisfiable(C_u \wedge b_i \wedge FM)$ (with FM the representation of the feature model in a boolean formula) which, expanded using the definition of C_u, gives:

$$satisfiable\left(\left(\bigwedge_{i=1..m} b_i \leftrightarrow \mathcal{PC}(b_i)\right) \wedge b_i \wedge FM\right)$$

A.2 KCONFIG internal consistency by Hengelein [2015]

In his Master's thesis, Hengelein [2015] analyses the internal consistency of KCONFIG and characterises six different types of anomalies. While Anomalies 4, 5 and 6 are common anomalies resulting from conflicts between constraints on the features, Anomalies 7, 8 and 9 are related to the syntax of the KCONFIG files.

Anomaly 4 (Dead feature [Hengelein, 2015])**.** A feature is dead if there are contradictions in its dependencies.

Anomaly 5 (False optional (undead) feature * [Hengelein, 2015])**.** A false optional feature in KCONFIG is a feature that is selected by another feature that is always on or selected by a feature that is false optional itself.

Anomaly 6 (Missing dead feature [Hengelein, 2015])**.** A feature is missing dead if features in the dependencies are not defined in KCONFIG.

Anomaly 7 (Selects on Symbols with Dependencies [Hengelein, 2015])**.** `select` statements should not be used to select symbols matching the following conditions:

- The symbol has dependencies

[1] In the remainder of this chapter, we will refer to it as $sat()$.

- The symbol is selected by another symbol

Anomaly 8 (Unreachable symbol [Hengelein, 2015])**.** A symbol is unreachable if:

- The symbol is invisible (does not have a prompt)
- The symbol is not selected by another symbol
- The symbol does not have a default value (or just default values with the value `n`)

Anomaly 9 (Unnecessary Selects on Choice Values [Hengelein, 2015])**.** `select` statements are unnecessary on symbols matching the following conditions:

- The symbol is a choice value
- The symbol is selected by another symbol

A.3 KBUILD consistency by Nadi and Holt [2011]

Nadi and Holt [2011] investigate both the internal and external consistencies of the KBUILD Makefiles by studying the (non-)use of composite objects, and the non-selection of a file because of a missing feature. The absence of files from the code base in the Makefiles is also studied (Anomaly 10), but does not result from a conflict between constraints in the build system.

Anomaly 10 (File Not Used (implementation-compilation consistency) [Nadi and Holt, 2012])**.** A .c file exists in the directory but is not used in the Makefile of that directory.

Anomaly 11 (Feature Not Defined (compilation-configuration consistency) [Nadi and Holt, 2012])**.** A .c file is referenced in the Makefile, and its presence is conditioned on a KCONFIG feature being defined. However, this feature is not defined in any of the KCONFIG files.

Anomaly 12 (Variable Not Used (compilation self-consistency) [Nadi and Holt, 2012])**.** A .c file is referenced in the Makefile as part of a composite variable definition, but this variable is never used.

A.4 KCONFIG–CPP consistency by Tartler et al. [2011]

Tartler et al. [2011] characterize defects issuing from conflicts between the KCONFIG and the CPP space. They first give the following definition of *dead* and *undead* blocks.

Anomaly 13 (Configurability defect * [Tartler et al., 2011])**.** A configurability defect (short: defect) is a configuration-conditional item that is either dead (never included) or undead (always included) under the precondition that its parent (enclosing item) is included:

$$\text{dead: } \neg sat(\mathcal{C} \wedge \mathcal{I} \wedge Block_N)$$
$$\text{undead: } \neg sat(\mathcal{C} \wedge \mathcal{I} \wedge \neg Block_N \wedge parent(Block_N))$$

with $\mathcal{C}$ and $\mathcal{I}$ the formulas representing the *configuration* (*i.e.*, KCONFIG) and *implementation* (*i.e.*, Make) spaces respectively.

The authors then reuse the formalism proposed by Sincero et al. [2010] to simplify the defects with the two following definitions.

Anomaly 14 (Implementation-only defects [Tartler et al., 2011], simplification of Anomaly 2)**.** *Implementation-only defects [. . .] represent code blocks that cannot be selected regardless of the systems' selected features; the structure of the source file itself contains contradictions that impede the selection of a block. This can be determined by checking the satisfiability of the formula* $sat\left(b_i \leftrightarrow \mathcal{PC}(b_i)\right)$. We can infer the expressions for dead and undead implementation-only defects.

$$\text{dead: } \neg sat\left(b_i \leftrightarrow \mathcal{PC}(b_i)\right)$$
$$\text{undead: } \neg sat\left(\neg\left(b_i \leftrightarrow \mathcal{PC}(b_i)\right)\right)$$

Anomaly 15 (Configuration-implementation defects * [Tartler et al., 2011], simplification of Anomaly 3)**.** Configuration-implementation defects *occur when the rules from the configuration space contradict rules from the implementation space. We check for such defects by solving* $sat\left(\left(b_i \leftrightarrow \mathcal{PC}(b_i)\right) \wedge \mathcal{V}\right)$. We can infer the expressions for dead and undead configuration-implementation defects.

$$\text{dead: } \neg sat\left(\left(b_i \leftrightarrow \mathcal{PC}(b_i)\right) \wedge \mathcal{V}\right)$$
$$\text{undead: } \neg sat\left(\neg\left(b_i \leftrightarrow \mathcal{PC}(b_i)\right) \wedge \mathcal{V}\right)$$

with $\mathcal{V}$ the propositional formula representing the configuration space (*i.e.*, the feature model of KCONFIG).

The authors then define two defects internal to the KCONFIG.

Anomaly 16 (Configuration-only defects [Tartler et al., 2011])**.** Features are present in the configuration-space model but do not appear in any valid configuration of the model, which means that the presence condition of the feature is not satisfiable. We can check for such defects by solving: $sat(feature \rightarrow presenceCondition(feature))$. However, no formal definition of $presenceCondition$ was given.

Anomaly 17 (Referential defects [Tartler et al., 2011])**.** *Referential defects are caused by a missing feature* (m) *that appears in either the configuration or the implementation space only. That is:*

$$sat\left(\left(b_i \leftrightarrow \mathcal{PC}(b_i)\right) \wedge \mathcal{V} \wedge \neg\left(m_1 \vee \cdots \vee m_n\right)\right)$$

is unsatisfiable.

A.5 KCONFIG–KBUILD–CPP consistency by Nadi and Holt [2012]

Nadi and Holt [2012] improve UNDERTAKER [Tartler et al., 2011] to add constraints from the Make space and identify *dead* and *undead* artifacts at both source file and code block levels, relying on constraints from the three spaces.

Anomaly 18 (Code anomalies [Nadi and Holt, 2012])**.** Code anomalies are defined as *"Conflicting code constraints"* and are not expressed in the paper as they are already determined by the UNDERTAKER tool designed in [Tartler et al., 2011]. Thus, formulas to detect these anomalies are the ones from Anomalies 2 and 14.

Anomaly 19 (Code–KCONFIG defects ⋆ [Nadi and Holt, 2012])**.** Code–KCONFIG anomalies are defined as *"Code constraints are not consistent with constraints in Kconfig"* and detected using the following formulas:

$$Dead_{B_N} = \neg sat(Block_N \wedge C \wedge K)$$
$$Undead_{B_N} = \neg sat(\neg Block_N \wedge parent(Block_N) \wedge C \wedge K)$$

These formulas are strictly identical to Anomaly 15, thus their expressiveness in our model will be checked together.

Anomaly 20 (Code–KCONFIG missing [Nadi and Holt, 2012])**.** Such defects happen when *Code constraints are not consistent with Kconfig constraints because certain features used in the code are not defined in the Kconfig files and are, therefore, always false*.

Anomaly 21 (Code–Make [Nadi and Holt, 2012])**.** Code–Make anomalies are defined as *"Code constraints are not consistent with constraints in Makefiles"*. Although their formulas are not given in the paper, we can deduce them from Anomaly 19:

$$Dead_{B_N} = \neg sat(Block_N \wedge C \wedge M)$$
$$Undead_{B_N} = \neg sat(\neg Block_N \wedge parent(Block_N) \wedge C \wedge M)$$

Anomaly 22 (Code–Make–KCONFIG ⋆ [Nadi and Holt, 2012])**.** Code–Make–KCONFIG anomalies are defined as *"The combination of constraints in the three spaces are conflicting"* and detected using the following formulas:

$$Dead_{B_N} = \neg sat(Block_N \wedge C \wedge M \wedge K)$$
$$Undead_{B_N} = \neg sat(\neg Block_N \wedge parent(Block_N) \wedge C \wedge M \wedge K)$$

Anomaly 23 (Code–Make–KCONFIG missing [Nadi and Holt, 2012])**.** Such defects happen when *"The combination of constraints in the three spaces are conflicting because certain features used in the compilation constraints are not defined in the Kconfig files, and are therefore always false"*.

Anomaly 24 (Make–KCONFIG ⋆ [Nadi and Holt, 2012])**.** A file is dead *"if it can never be present (*i.e.*, will never get compiled) while satisfying the combination of constraints in the Make space and the* KCONFIG *space"*. These anomalies are checked by checking these formulas.

$$Dead_{F_N} = \neg sat(File_N \wedge M \wedge K)$$
$$Undead_{F_N} = \neg sat(\neg File_N \wedge M \wedge K)$$

Anomaly 25 (Make–KCONFIG missing [Nadi and Holt, 2012])**.** The definition of this type of defects is not written literally in the paper but we can derive the definition from Anomalies 20 and 23. Such defects happen when the combination of constraints in the make and KCONFIG spaces are conflicting because certain features used in the Makefiles are not defined in the KCONFIG files, and are therefore always false.

Maîtriser la variabilité enfouie dans les systèmes orientés objet et les systèmes de construction logicielle

Johann MORTARA

Résumé

La demande sans cesse croissante de solutions logicielles nouvelles et récentes oblige les professionnels du logiciel à développer et à maintenir des systèmes logiciels personnalisables tout en garantissant un niveau élevé de qualité et de fiabilité. Si les lignes de produits logiciels (LPLs) constituent une solution pour atteindre cet objectif, de nombreux systèmes logiciels riches en variabilité ne sont pas organisés de cette manière. Ils augmentent progressivement leurs parties variables, en s'appuyant sur les multiples mécanismes existants pour mettre en œuvre leur variabilité dans le code et leur chaîne d'outils de construction. Dans leur mise en œuvre, les systèmes orientés objet (OO) gèrent principalement leur variabilité dans une base de code unique en utilisant des mécanismes OO tels que l'héritage et les patrons de conception. En raison de leur nature, ces implémentations sont enfouies dans la base de code, ce qui nuit à la compréhension du système par les développeurs et, par conséquent, à sa maintenance et à son évolution, entraînant des problèmes de qualité. En outre, les grands systèmes logiciels riches en variabilité s'appuient souvent sur des systèmes de construction complexes pour sélectionner les éléments du code. Comme il s'agit de systèmes de construction ad hoc réutilisant des outils standard, aucune représentation globale du mécanisme de résolution de la variabilité n'est disponible, et des conflits peuvent survenir et causer des anomalies. Dans cette thèse, nous proposons tout d'abord les bases et les techniques pour identifier et visualiser les implémentations de la variabilité dans les grands systèmes logiciels OO riches en variabilité. Ces implémentations sont abstraites en termes de points de variation et de variantes et identifiées en s'appuyant sur la notion de densité de symétries dans les structures OO. En reprenant la métaphore d'une ville, elles sont ensuite visualisées sous la forme d'un ensemble connecté de bâtiments 3D combinés à des métriques sur leur qualité. Cela permet de distinguer les zones concentrant les implémentations de variabilité et présentant potentiellement une dette technique. Ces propositions ont été validées par un prototype sur de grands systèmes logiciels OO open-source et hautement variables, ainsi que par une étude d'utilisabilité avec deux groupes distincts de développeurs débutants. La thèse introduit également un cadre de modélisation et de raisonnement pour caractériser les anomalies dans les systèmes de construction gérant de la variabilité, permettant de raisonner sur les relations entre les actifs du code, et d'identifier toutes ces anomalies au grain le plus fin. Le framework a été instancié et partiellement implémenté à la fois sur le système de construction du noyau Linux, démontrant sa généralité sur les nombreuses détections distinctes sur ce sujet très étudié, et sur une chaîne d'outils de construction récemment étudiée de la fondation Mozilla, démontrant son applicabilité.

Mots-clés : Génie logiciel, Lignes de produits logiciels, Variabilité logicielle, Modèle de variabilité, Rétro-ingénierie, Visualisation.