The Name of the Title Is Hope

Federico Bruzzone
federico.bruzzone@unimi.it
Università degli Studi di Milano
Computer Science Department
Milan, Italy, Europe

Walter Cazzola*
cazzola@di.unimi.it
Università degli Studi di Milano
Computer Science Department
Milan, Italy, Europe

Abstract

A clear and well-documented LaTeX document is presented as an article formatted for publication by ACM in a conference proceedings or journal publication. Based on the "acmart" document class, this article presents and explains many of the common variations, as well as many of the formatting elements an author may use in the preparation of the documentation of their work.

CCS Concepts

• **Theory of computation** → **Program analysis**; • **Software and its engineering** → *Compilers*.

Keywords

Do, Not, Us, This, Code, Put, the, Correct, Terms, for, Your, Paper

ACM Reference Format:
Federico Bruzzone and Walter Cazzola. 2018. The Name of the Title Is Hope. In *Proceedings of 29th International Systems and Software Product Line Conference (SPLC'25)*. ACM, New York, NY, USA, 2 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

BF [Acronyms: SPL]

2 Background

In this section, we introduce some preliminary concepts that are necessary to understand the rest of the paper. We start by introducing the Rust programming language and its ownership system. Then, we introduce the concept of software product lines (SPLs) and the importance of testing SPLs. Finally, we provide an overview of centrality measures in graph theory, which are necessary to understand the approach we propose in this paper.

2.1 The Rust Programming Language

Rust is a systems programming language that focuses on safety, speed, and concurrency. It is designed to be memory-safe without using garbage collection. This implies that pure Rust programs are free of null pointer dereferences, double frees as well as data races. The *linear logic* [3, 4] and *linear types* [9, 14]—which force the use of resources exactly once—inspired the *ownership* system. Rust incorporates it into its type system as relaxed form of pure linear types to ensure type soundness. The ownership system ensures that there is only one *owner* (the variable binding) for each piece of memory (a value) at any given time, and when the owner goes out of scope or is otherwise deallocated, the memory is deallocated as well. By leveraging the latter property, Rust supports user-defined destructors, enabling *resource acquisition is initialization* (RAII) pattern proposed by Stroustrup [13]. The lifetime of the owned value is determined by the scope in which the owner takes ownership. An owner can *move* (transfer) the ownership of the value to a new owner or *borrow* the value to another part of the program. By *moving* the ownership, the previous owner can no longer access the value. On the other hand, Rust support *references* that allow the owner to *borrow* the value avoiding the its invalidation. Two kind of *borrows* are supported: *immutable* and *mutable*. Multiple *immutable borrow* can coexist, but only one *mutable borrow* can exist at a time. These restrictions allow Rust to guarantee memory safety. Furthermore, the lifetime of a reference can not outlive (exceed) the lifetime of the owner, which ensures no dangling pointers. The Rust compiler enforces all these rules at compile time also by performing *borrow checking*, preserving the runtime performance of the compiled code. Despite the notable progress in the field of safe systems programming, Rust allows `unsafe` blocks to perform low-level operations that are not safe, such as dereferencing raw pointers. In Rust, the `unsafe` keyword signifies that the responsibility for preventing undefined behavior shifts from the compiler to the programmer. This ensures that undefined behavior cannot occur in safe Rust code, as the compiler enforces strict safety guarantees in all safe contexts.

2.2 Product Families

In product families the similarities and differences are characterized by a set of features $F = \{f_1, f_2, \ldots, f_m\}$ where each feature $f_i \subseteq A$. A feature is a unit that provides a piece of functionality that satisfies a requirement or represents a design decision and fixed i and j such that $i \neq j \implies f_i \cap f_j = \emptyset$. A product line, or rather a family of products, is a set of products $P = \{p_1, p_2, \ldots, p_k\}$ where each product $p_i \subseteq F$ is a set of features and for fixed i and j such that $i \neq j$, it does not necessarily follow that $p_i \cap p_j = \emptyset$. A key task in SPL engineering is feature modeling, which involves creating and maintaining a *feature model*. The concept of feature model was first introduced by Kang et al. [5] in the FODA method and serves to represent the variability of a system through its features and their interdependencies. In SPLs, the feature model formalism is essential for configuring software products by defining valid feature sets, known as *configurations*. A configuration $c : F \longrightarrow \{0, 1\}$ is a

*Corresponding author.

SPLC'25, September 01–September 05, 2025, A Coruña, Spain

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

characteristic function over F that maps each feature to a boolean value. A feature f is considered *active* if it belongs to a configuration c such that $c(f) = 1$, otherwise it is *inactive*. The structure of a feature model implicitly captures feature dependencies by specifying mandatory, optional, alternative, and grouped features. These dependencies are often represented as parent-child relationships, where a feature can only be *active* if all its parent features are also *active*. A configuration c is deemed *valid* if and only if $\forall f_i \in F \mid c(f_i) = 1 \implies \exists p \in P \mid f_i \in p$. Given a product $p_i \in P$ we say that all products $p_j \in P$ such that $p_j \neq p_i$ are variants of p_i denoted as v_j. It is worth noting that a family of products P can theoretically contain up to $2^{|F|}$ variants, as described by Krueger [6]. This exponential growth in potential configurations has paved the way for the development of techniques to manage and test SPLs effectively [12]. Numerous approaches, recently analyzed by Agh et al. [1], have been proposed to address the challenges of testing SPLs, including product sampling [2, 7, 11] and combinatorial testing [8, 10].

2.3 Centrality Measures

3 Related Work

BF [Nella sezione 4.4 di [1]] BF [Guardare le precedenti literature review]

Acknowledgments

BF [TODO]

References

[1] Halimeh Agh, Aidin Azamnouri, and Stefan Wagner. 2024. Software product line testing: a systematic literature review. *Empirical Software Engineering* 29, 6 (2024), 146.

[2] Mustafa Al-Hajjaji, Thomas Thüm, Malte Lochau, Jens Meinicke, and Gunter Saake. 2019. Effective product-line testing using similarity-based product prioritization. *Software & Systems Modeling* 18 (2019), 499–521.

[3] Jean-Yves Girard. 1987. Linear logic. *Theoretical computer science* 50, 1 (1987), 1–101.

[4] Jean-Yves Girard, Yves Lafont, and Laurent Regnier. 1995. *Advances in linear logic*. Vol. 222. Cambridge University Press.

[5] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report CMU/SEI-90-TR-21. Carnegie Mellon University, Pittsburgh, Pennsylvania, USA.

[6] Charles W. Krueger. 2006. New Methods in Software Product Line Practice. *Commun. ACM* 49, 12 (Dec. 2006), 37–40.

[7] Jihyun Lee and Sunmyung Hwang. 2019. Combinatorial Test Design Using Design-Time Decisions for Variability. *International Journal of Software Engineering and Knowledge Engineering* 29, 08 (2019), 1141–1158.

[8] Malte Lochau, Sebastian Oster, Ursula Goltz, and Andy Schürr. 2012. Model-based pairwise testing for feature interaction coverage in software product line engineering. *Software Quality Journal* 20 (2012), 567–604.

[9] Martin Odersky. 1992. Observers for linear types. In *European Symposium on Programming*. Springer, 390–407.

[10] Sebastian Oster, Florian Markert, and Philipp Ritter. 2010. Automated incremental pairwise testing of software product lines. In *International Conference on Software Product Lines*. Springer, 196–210.

[11] Sachin Patel, Priya Gupta, and Vipul Shah. 2013. Combinatorial interaction testing with multi-perspective feature models. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 321–330.

[12] Klaus Pohl and Andreas Metzger. 2006. Variability Management in Software Product Line Engineering. In *Proceedings of the 28th International Conference on Software Engineering (ICSE'06)*, Leon J. Osterwell, H. Dieter Rombach, and Mary Lou Soffa (Eds.). ACM, Shanghai, China, 1049–1050.

[13] Bjarne Stroustrup. 1994. *The design and evolution of C++*. Pearson Education India.

[14] Philip Wadler. 1990. Linear types can change the world!. In *Programming concepts and methods*, Vol. 3. Citeseer, 5.

A Appendix 1

A.1 Part One

BF [TODO]

A.2 Part Two

BF [TODO]

B Appendix 2

C Part One

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009