



Evolving software system families in space and time with feature revisions

Gabriela Karoline Michelon¹ · David Obermann² · Wesley K. G. Assunção^{2,3} · Lukas Linsbauer⁴ · Paul Grünbacher² · Stefan Fischer⁵ · Roberto E. Lopez-Herrejon⁶ · Alexander Egyed²

Accepted: 10 December 2021 / Published online: 30 May 2022
© The Author(s) 2022

Abstract

Software companies commonly develop and maintain variants of systems, with different feature combinations for different customers. Thus, they must cope with variability in space. Software companies further must cope with variability in time, when updating system variants by revising existing software features. Inevitably, variants evolve orthogonally along these two dimensions, resulting in challenges for software maintenance. Our work addresses this challenge with *ECSEST* (Extraction and Composition for Systems Evolving in Space and Time), an approach for locating feature revisions and composing variants with different feature revisions. We evaluated *ECSEST* using feature revisions and variants from six highly configurable open source systems. To assess the correctness of our approach, we compared the artifacts of input variants with the artifacts from the corresponding composed variants based on the implementation of the extracted features. The extracted traces allowed composing variants with 99–100% precision, as well as with 97–99% average recall. Regarding the composition of variants with new configurations, our approach can combine different feature revisions with 99% precision and recall on average. Additionally, our approach retrieves hints when composing new configurations, which are useful to find artifacts that may have to be added or removed for completing a product. The hints help to understand possible feature interactions or dependencies. The average time to locate feature revisions ranged from 25 to 250 seconds, whereas the average time for composing a variant was 18 seconds. Therefore, our experiments demonstrate that *ECSEST* is feasible and effective.

Keywords Feature location · Feature revisions · Variation control system · Repository mining

Communicated by: Philippe Collet, Sarah Nadi, Christoph Seidl, and Leopoldo Motta Teixeira

This article belongs to the Topical Collection: *Software Product Lines and Variability-rich Systems (SPLC)*

✉ Gabriela Karoline Michelon
gabriela.michelon@jku.at

Extended author information available on the last page of the article.

1 Introduction

Software system families can evolve in two dimensions: (i) evolution *in space*, when new product variants need to be customized, and (ii) evolution *in time*, when features of existing individual product variants need to be modified over time (Strüber et al. 2019). Throughout the life cycle of software systems, the creation of different product variants is unavoidable due to the diversity of functional and non-functional customer requirements in different market segments, usage scenarios, and platforms, leading to evolution in space. In the time dimension, each product variant continuously evolves, resulting in multiple revisions of the variant. Evolution in this dimension is the result of customers requiring enhancements, bug fixes, or scalability issues requiring implementation changes (Melo et al. 2016).

A software product line (SPL) can encompass evolution in space by systematically managing and tailoring many variants of a software system. An SPL consists of a platform with a set of defined and managed features, each implementing functionality and behavior visible to the end-user (Pohl et al. 2005). The SPL approach initially requires high upfront investment compared to traditional single system development, but in the long run pays off with high numbers of features and product variants (Strüber et al. 2019). SPLs also evolve over time due to bug fixes, refactoring, or enhancing modifications of features or the code of existing variants (Hinterreiter et al. 2019; Michelon et al. 2020a).

Features of product variants derived from an SPL can be modified to quickly meet customer demands. However, if modifications are not propagated to the SPL platform, they can hardly be reused in other products. For instance, if a particular change is required for a new hardware device acquired by a customer, reusing this new version of the feature in other variants can rapidly become cumbersome. As a result, software companies not only have to deal with different product variants but also with different revisions of product variants and even different revisions of features over time. Analogous to sequential versions of software systems, i.e., revisions (Linsbauer et al. 2021), we adopt the concept of feature revisions as implementation modifications over time that lead to new feature versions. Despite this practical need of different feature versions, there is currently no straightforward solution for the challenging problem of integrating the management and evolution of system families in space and time (Berger et al. 2019).

Nowadays, software engineers use distinct approaches and tools to deal with these dimensions of evolution. An example is the combination of an SPL with a version control system (VCS) providing capabilities to track changes (Collins-Sussman et al. 2002). However, developers need to combine additional mechanisms and tools when evolving system families in space and time. Variability mechanisms are often specific for realizing variability in certain types of artifacts, e.g., AspectJ for Java or preprocessor annotations for text files (Linsbauer et al. 2021). A well-known example is the Linux kernel, which combines several variability management techniques (Clements and Northrop 2002) to provide an integrated software platform keeping variability information consistent across different types of artifacts (Berger et al. 2019). Linux relies on a variant-aware build system (Berger et al. 2010), an interactive configurator tool (Sincero et al. 2007), and a variability model representation (Berger et al. 2013). In addition to its build rules encoded in Makefiles, the Linux kernel is implemented with preprocessor directives to customize its features and to control the compilation of entire source files or fragments (Passos et al. 2015). Still, the system is hosted in a VCS to keep track of changes over time¹.

¹<https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux>

Most existing mechanisms for variability management have a strong impact on software development, as they require adding annotations to the source code, or assume certain programming paradigms (such as feature-oriented or aspect-oriented programming). These variability mechanisms require manual placement of variation points and their concrete realization is usually specific for a certain type of artifact (e.g., AspectJ for Java) (Linsbauer et al. 2021). The vast majority of industrial SPLs are realized with preprocessor directives (Apel et al. 2013; Medeiros et al. 2015). However, they do not support revision management, which is usually handled by VCSs to preserve the evolution history (Berger et al. 2020). Although preprocessor directives can be used in combination with VCSs, they have received strong criticism regarding the separation of concerns, error proneness, and code obfuscation (Medeiros et al. 2015). Furthermore, preprocessor directives are limited to managing system variability of textual files. System families, however, rarely consist of only a single type of artifact (Linsbauer et al. 2017). Furthermore, while VCSs support *evolution in time* at the file or directory level, they do not provide adequate support for *evolution in space* as shown recently (Berger et al. 2019; Krüger et al. 2019; Linsbauer et al. 2017; Linsbauer et al. 2021). Thus, the current combination of tools in practice, unfortunately, does not allow to comprehensively and uniformly handle variants and revisions (Hinterreiter et al. 2019; Linsbauer et al. 2021; Michelon et al. 2021a).

Variation control systems (VarCSs) have been proposed to address these challenges, as discussed in a recent survey (Linsbauer et al. 2021). A VarCS supports system development based on features, reducing the complexity of changing variants and easing the maintenance and evolution of revisions by alleviating developers from manually editing variation points and integrating the changes (Linsbauer et al. 2021). For instance, the VarCS ECCO² supports the evolution of arbitrary types of artifacts (Michelon et al. 2020d) based on plug-in architecture (Linsbauer et al. 2021). However, developers are still concerned to replace popular and mature mechanisms, such as the combination of SPLs and VCS, with VarCSs providing proprietary repositories and unfamiliar operations. That is why annotation-based preprocessors combined with VCS are still the most popular variability mechanism (Linsbauer et al. 2021).

Based on the limitations and needs for properly dealing with the evolution of systems in space and time at the level of features, this paper extends previous work (Michelon et al. 2020d) on feature revision location implemented in ECCO to recover information of the system evolution over time at the feature level. We present and evaluate *ECSEST* (Extraction and Composition for Systems Evolving in Space and Time), an approach for aiding system evolution in space and time by locating feature revisions and composing variants with different combinations of features and their revisions. *ECSEST* thus not only supports the analysis of software systems evolving in space and time by locating feature revisions, but also allows the composition of new products by using the located feature revisions.

As a novelty to support the composition of new products, *ECSEST* introduces a strategy to provide hints about feature interactions and possibly missing or surplus feature revisions for a configuration. This eases system development and the composition of new products based on new combinations of features and their revisions. Hence, *ECSEST* can aid the evolution of software families at both levels of domain engineering and application engineering (Apel et al. 2013). For example, it supports the domain implementation and product derivation by reusing and combining artifacts that correspond to feature revisions.

²<https://github.com/jku-isse/ecco>

Specifically, in this paper we extend our previous work (Michelon et al. 2020d) as follows:

- *Composition of product variants based on feature revisions:* We present an approach for composing variants and provide further details of the feature revision location. We include an illustrative example to show how our approach supports system evolution in space and time and how it is implemented in ECCO². Additionally, we further extended our approach with hints showing possible conflicts and interactions between feature revisions when composing new configurations.
- *Support for C language artifacts:* We developed a new adapter, i.e., a new ECCO plug-in, using a fine-grained tree structure to perform feature revision location, while our previous work (Michelon et al. 2020d) was using a text plug-in. Our new plug-in improves the analysis of artifact equivalence when computing traces by analyzing differences in the abstract syntax tree (AST) of feature revisions. Thus, it also enables the evaluation of the evolution of C source code. Although our approach is independent of artifact type, the new plug-in makes our approach easily adoptable in practice, given the high number of SPLs implemented in C.
- *Evaluation with additional systems and more feature revisions:* We extended our empirical evaluation by applying our approach to more systems from different domains. We now locate feature revisions and compose new configurations with the located feature revisions from six systems. Our analysis includes more points in time, leading to more feature revisions and more product variants. We thus mined more Git commits of more C preprocessor-based systems and included more system variants in our replication package³.
- *Enhanced analysis:* To evaluate the approach of this extended work, we now compute the runtime performance of composing variants with feature revisions besides precision, recall, and extraction time. Further, we computed new metrics for the hints retrieved when composing new products. These metrics indicate conflicts and interactions when composing product variants with different combinations of feature revisions. We computed further metrics allowing an in-depth analysis of feature evolution at the implementation level. These metrics count the different AST nodes used by our plug-in to store the artifacts of feature revisions.

The remainder of this paper is structured as follows. Section 2 presents background information. Section 3 discusses the motivation for our research and the problems we aim to address. Section 4 explains our *ECSEST* approach. Section 5 presents the research questions of our evaluation, as well as the methodology, subject systems, the metrics used in our experiments, and relevant implementation aspects. Section 6 summarizes and discusses the results. Section 7 discusses threats to validity and Section 8 presents related research. Finally, Section 9 concludes the paper and outlines future work.

2 Background

Variability in space and time is the consequence of collectively developing and maintaining families of software systems (Schwägerl 2018). Software systems need to evolve due to developer mistakes and unpredictable future requirements. Variability in space means that

³<http://doi.org/10.5281/zenodo.4555199>

different co-existing functional assets of a software system exist at a specific point in time. Variability in time means that different revisions of a functionality, i.e., an asset or a set of assets, exist at different points in time (Pohl et al. 2005). This section discusses existing concepts, approaches, techniques, and tools to deal with evolution in space and/or time.

Software Product Lines SPLs provide systematic reuse of assets through the development of a common core and a set of features satisfying customer needs of a particular market segment (Clements and Northrop 2002). The main advantages of systematic reuse in SPLs are the reduction of development costs and the time-to-market, and an increase in software quality (McGovern et al. 2003; Pohl et al. 2005). However, adopting an SPL strategy requires expensive up-front investment (Martinez 2016) for defining the SPL scope and product portfolio offered by a software company (Pohl and Metzger 2018). Further, it is necessary to define which set of features and which set of domain artifacts can be reused for product composition.

A product, therefore, is composed of a set of artifacts that realize the set of features constituting a valid software system (Estublier 2000). A new product variant is needed when no existing variant implements the requested features, or when some of the features were modified to update existing variants. Thus, the composed product variant will contain the same features, but not the same implementation as before (Ghabach et al. 2018). In this context, feature location techniques can ease evolution and maintenance tasks (Bennett and Rajlich 2000).

Feature Location Features are the building blocks of SPLs and are defined as a user-visible functionality of the system (Apel et al. 2013). Feature location aims at finding the artifacts responsible for implementing specific system functionalities. Additionally, feature location is used during the incremental change process to determine where a change should be done in the code and to find the affected code. Thus, feature location techniques have been used for maintenance and evolution tasks, as well as for analyzing the impact of changes (Dit et al. 2013). Feature location has received significant attention in the research community and many (semi-)automated techniques have been proposed (Assunção and Vergilio 2014; Cruz et al. 2019; Rubin and Chechik 2013), which can be classified into four categories: (i) dynamic feature location techniques examine the system during runtime and retrieve feature information through execution traces of constructed scenarios, where the feature to be located is exercised (Cruz et al. 2019); (ii) static feature location techniques rely on the source code structure to find feature code (Dit et al. 2013); (iii) textual feature location techniques use textual analysis to find feature code, such as information retrieval and natural language processing analysis (Cruz et al. 2019); and (iv) hybrid feature location techniques combine several strategies (Dit et al. 2013; Michelon et al. 2021b, d). Our approach adopts static analysis to compare the artifacts and feature revisions of existing system variants (see Section 4).

Version and Variation Control VCSs have been used to manage system evolution in time. A version control system, a.k.a revision control system, tracks incremental versions (or revisions) of files and directories over time (Collins-Sussman et al. 2002). VCSs allow the implementation of systems in a collaborative way, i.e., a system can be developed in parallel by multiple developers who can later explore the change history. Parallel development of software features is commonly handled in VCSs either with branching mechanisms or optimistic methods, such as copy-modify-merge, workspaces, and transactions (MacKay 1995). However, as explained, developers not only have to maintain and evolve revisions

of a software system but also need to maintain different products of an SPL (Conradi and Westfechtel 1998). Therefore, the evolution of software systems can be characterized with a two-dimensional view with variants incrementing along one axis and revisions incrementing through time along the other axis (MacKay 1995).

Cloning variants via branching or forking mechanisms of VCSs offers only limited support for system families because variations are not managed at a fine-granular level based on features or a similar concept (Linsbauer et al. 2021). As a consequence, multiple products need to be maintained, which leads to high maintenance efforts (Schwägerl 2018). Even branching models (MacKay 1995) for feature development require developers to manually edit variation points and to manually integrate changes. Further, as the feature is developed in isolation, it is no longer available once merged into the system branch. Hence, branching mechanisms do not offer adequate support for understanding and maintaining the evolution in space and time of system variants.

VCSs and SPLs are thus widely used in combination to support variability and evolution (Berger et al. 2019). Annotation-based SPLs combined with VCSs allow customizing different products with preprocessor directives. However, with this variability mechanism features can be delimited only in text files, and an SPL rarely consists of only a single type of artifact (Linsbauer et al. 2021). Furthermore, VCSs enable the versioning of the whole platform and can recover and keep track of changes of lines and files but not at the level of features (Berger et al. 2019). Therefore, maintenance and evolution tasks require manual analysis of tangled features in multiple files and blocks of code. This is a cumbersome and complex task since preprocessor directives are error-prone and hamper code comprehension (Medeiros et al. 2015; Michelon et al. 2021a).

Some VarCSs offer support for feature development of software systems over time by transactionally editing and automatically integrating the features back into the variant-rich system (Linsbauer et al. 2021). In a survey, Linsbauer et al. (2021) identified six VarCSs that can offer visible operations, such as externalization, modification, and internalization in a transactional way. Three VarCSs support revisions of the whole system, while only ECCO supports feature revisions (Michelon et al. 2020d). It also provides support for different programming language and artifact types and can thus version any kind of artifact. Thus, ECCO has capabilities to aid the maintenance and evolution of a software system family at the level of features with no extra costs. Previous studies (Fischer et al. 2014, 2015, 2019; Michelon et al. 2019, 2021b, d) already showed satisfactory evaluation results using ECCO in the context of SPLs and software system evolution.

3 Motivation

In an SPL the evolution in time is more complex than evolving single variants. For instance, developers need to consider all variants at the same time when using preprocessor directives (Melo et al. 2016). Although annotation-based SPLs can be versioned in VCSs, the evolution in time is tracked for the whole platform at coarse granularity (Berger et al. 2019; Linsbauer et al. 2021; Michelon et al. 2021a, c). Even if systematic reuse is realized by annotation-based SPLs in VCSs, manual analysis and propagation of changes of feature revisions in different releases of a system are highly challenging. An example can be seen in SQLite⁴, a C-language library implementing the most widely used database engine in the

⁴<https://www.sqlite.org/index.html>

world. The feature `SQLITE_TEST` was modified for the release *branch-3.9*⁵. The same set of changes, i.e., the feature revision `SQLITE_TEST` committed in the release *branch-3.9*, had to be propagated to three newer releases: *branch-3.18*, *branch-3.19*, and *branch-3.22*. This example confirms that features have different implementations, with different behaviors at different points in time, which is of interest for developers combining different revisions of a feature in existing configurations.

We illustrate the challenges of evolving the source code of a real system implemented with preprocessor directives in Git VCS. For that, we consider LibSSH⁶, a C multiplatform library implementing the SSHv2 protocol on the client and server-side. We used our mining tool (Michelon et al. 2021a, c) to retrieve LibSSH's feature revisions from all 5022 commits, covering 48 releases and around 16 years of development. The analysis shows significant system evolution in both space and time. Over the system life cycle, 511 features were introduced, 302 were changed at least once, representing a total of 6242 feature revisions, also including the changes of the system core as feature revisions. Dealing with all releases of a system and considering the huge configuration space with feature revisions leads to complex maintenance tasks. Usually, multiple features change in a single commit, and commit messages not always reflect the changes performed (Herzig et al. 2016). Finding which parts of the source code of specific annotated features are causing problems and should be changed requires deep developer's knowledge, in particular if other developers do not comprehend earlier design decisions (Krüger et al. 2021; Nassif and Robillard 2017), mainly when many developers are involved in open-source projects.

Regarding feature evolution, we present an analysis of the features with the highest number of revisions in the LibSSH system. The feature `WITH_SERVER` changed in 21 releases, resulting in a total number of 241 feature revisions. In this case, 21 system releases contain this feature, i.e., each possible configuration including this feature could have 241 different implementations, if we consider all commits of its development. However, even if a developer has to analyze a smaller number of feature revisions, the code has to be manually analyzed and retrieved. We analyzed some of the commits changing `WITH_SERVER`: usually multiple files and lines of source code changed in a single commit and also between releases of the system, thus resulting in different implementations of the feature at different points in time. Some of the changes represent bug fixes, e.g., commits 3b8c4dc⁷ and 9546b20d⁸, some represent new system functionality, e.g., commits b9b7174d⁹ and 9b2eefe6¹⁰, implemented by this feature, some represent deletions of functionality of this feature, e.g., commits 55846a4c¹¹ and 636432e4¹². Furthermore, we analyzed how many files and lines were part of the feature implementation in the first and last releases of the system. The feature was first added in two source code files and comprised 119 lines, while it covered 30 source code files and 4734 lines in the last release.

Now, suppose a software company has a product with a configuration containing a couple of features and needs to use another specific revision of the feature `WITH_SERVER`. If the

⁵<https://www.sqlite.org/src/info/7b4583f932ff0933>

⁶Analysis based on all commits of all releases of LibSSH: <https://gitlab.com/libssh>

⁷<https://gitlab.com/libssh/commit/3b8c4dc>

⁸<https://gitlab.com/libssh/commit/9546b20>

⁹<https://gitlab.com/libssh/commit/b9b7174>

¹⁰<https://gitlab.com/libssh/commit/9b2eefe>

¹¹<https://gitlab.com/libssh/commit/55846a4>

¹²<https://gitlab.com/libssh/commit/636432e>

SPL is implemented in a VCS, a developer may have to rely on documentation describing the kinds of changes performed in different commits of the feature `WITH_SERVER`, which is usually not available. Thus, the developer would have to manually select the feature code, then, copy and paste it to specific configuration releases. To retrieve another feature, for instance the feature `HAVE_SSH1` at a specific point in time from commit `5f7c84f9`¹³, the developer would need to analyze 29 files, 1339 additions, and 188 deletions. Thus, this manual task is very time-consuming, even more so if multiple features are committed at a single point in time, which is common in practice: for instance, the revisions of the feature `HAVE_SSH1` in the first 50 commits happened together with revisions of 13 other features impacting 73 source code files. Therefore, in this context, *ECSEST* aids maintenance and evolution tasks in annotation-based SPLs in VCSs by retrieving information of the system evolution in space and time at the feature granularity (Michelon et al. 2021c).

4 ECSEST Approach

We now present details of *ECSEST* (**E**xtraction and **C**omposition for **S**ystems **E**volving in **S**pace and **T**ime), our approach supporting software systems evolving in space and time. Figure 1 presents an overview of *ECSEST*. We first outline the feature revision location for extraction in Section 4.2 (Step 1 in Fig. 1), i.e., to map feature revisions to artifacts from existing software system variants. Locating feature revisions is an incremental process, which receives as input a product implementation and a configuration characterizing its features at a specific point in time. This step creates new traces and refines existing ones in the ECCO repository for every new input variant. We explain our approach for variant composition in Section 4.3 (Step 2 in Fig. 1), which requires as input a configuration provided by the user and the output traces stored in the ECCO repository created when locating feature revisions. The variant composition results in the product implementation and a file with hints to help the product completion. In the following, we give details of the data structures and processes of the feature revision location and variant composition.

4.1 Data Structures

Variants (Input) A variant $v \in V$ is a pair (F, A) , where F is a set of feature revisions and A is a set of implementation artifacts.

Features and Revisions Every feature f exists in multiple revisions r , denoted as f_r , where f and r are arbitrary unique identifiers for the feature and the revision, respectively. Two variants v_1 and v_2 with the same feature f have the same revision r of feature f , i.e., feature revision f_r , if the feature is implemented in the exact same way in both variants.

Implementation Artifacts A variant's implementation consists of a set of artifacts that are organized in a hierarchical tree structure, which we refer to as artifact tree. An artifact can represent a folder, a file, or any other element of a system's implementation. For example, in case of C source code, an artifact could represent a file, a field, a function, a block or a line of code inside a function, a header, or a define statement. We assume that any two artifacts a_1, a_2 can be compared for equivalence ($a_1 \equiv a_2$) as follows: two artifacts $a_1, a_2 \in A$ are

¹³<https://gitlab.com/libssh/commit/5f7c84f>

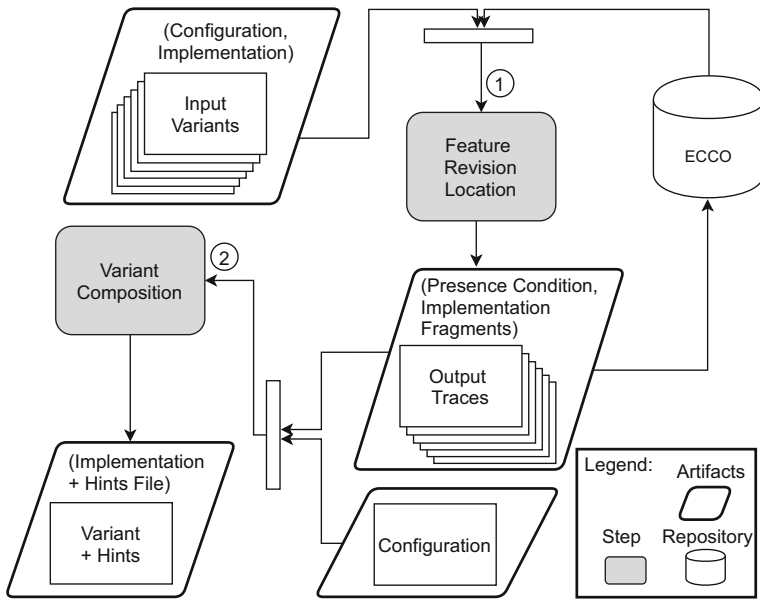


Fig. 1 The ECSEST approach overview

equivalent ($a_1 \equiv a_2$) if a_1 and a_2 are equal, e.g., textually equal in the case of programming artifacts ($a_1 = a_2$) and their parent artifacts are equivalent, i.e., their position in the artifact tree is the same. Thus, for programming artifacts, we compare if two nodes contain the same text-based artifact and if they have the same parent nodes. Here the same rule applies, parent nodes are equivalent if they are syntactically equal.

Traces (Output) The goal of our approach for feature revision location is to compute a presence condition C for every artifact a . The *output* therefore is a set of traces T . A trace $t \in T$ is a pair (C, A) that maps a set of artifacts A to a presence condition C . The traces can abstract where a feature is implemented, e.g., in which files and lines; as well as abstract feature interactions, i.e., artifacts that always appear together when specific feature revisions are present in a configuration. Furthermore, traces show which artifacts are common between feature revisions.

4.2 Feature Revision Location

For extraction of the evolution in space and time, the first step of our approach locates feature revisions (see Fig. 1). The *input* of the step is a *set of variants* V , with each variant v consisting of a configuration, i.e., a *set of feature revisions* F , and an implementation, i.e., a set of *artifacts* A . This is an incremental step where the existing traces T (output) stored in a repository are refined for every new input variant. A trace t consists of a presence condition C for a (set of) artifact(s). The input, as we explain in Fig. 3, for instance, can be a partial configuration of a variant, containing the set of feature revisions that changed in a specific commit. The commits of a system in a VCS thus represent points in time of new revisions of features. As *output*, our approach retrieves a *set of traces* T' , each mapping the implementation artifact fragments to a presence condition. Every artifact is then mapped

to a (set of) feature revision(s). This is necessary for composing the variants later on, i.e., when joining all artifacts in a product, the configuration must include the feature revisions containing the artifacts of the required core and functionalities.

ECSEST is independent of the artifact type by using a common structure for data in the location process. Thus, our approach can locate feature revisions in any artifact type if provided the input for our internal data structure. The approach can be extended with plug-ins (adapters) as long as different implementation languages and kinds of artifacts can be represented in a tree structure. In Fig. 2 we show the tree structure of the new plug-in implemented for parsing C source code artifacts. The tree structure adopted here is due to the type of artifacts of our ground truth variants used to evaluate our approach. For example, our approach already supports plug-ins for Java, text, UML models, PNG images, and LilyPond music artifacts, as shown in previous work (Michelon et al. 2019, b, d, Grünbacher et al. 2021).

Figure 3 shows the analysis of two dimensions: space and time of existing system variants for obtaining input variants necessary for the feature revision location process. For characterizing different points in time, i.e., when features were changed, numbers are added incrementally to the features' name. Figure 3 depicts such a situation: at a specific point in time T1, the software system was developed from scratch. For T1 we know that the features of the system were in their first revision and thus assigned the revision number 1. At the second point in time T2, we see a change of a specific feature ($Feat \in \mathcal{Y}$), which already existed at T1. Thus, the revision number of the feature was incremented to 2.

In Fig. 3, we can also see that each variant contains a feature called BASE, which represents the common code of the variants and represents the core of an SPL, i.e., parts of the system not related to features of the SPL. However, the core of the system is subject to frequent changes, and thus knowing the versions of the common code is also important for managing and evolving the system artifacts. Therefore, the core of the system is also mapped to a feature revision, which can have any name, but is represented here as the feature BASE. The feature revision location then analyzes in how many variants a feature revision appears, in how many variants a (set of) artifact(s) appears, and in how many variants a pair of feature revision(s) and a (set of) artifact(s) appear together. In this way, all artifacts are mapped to feature revisions.

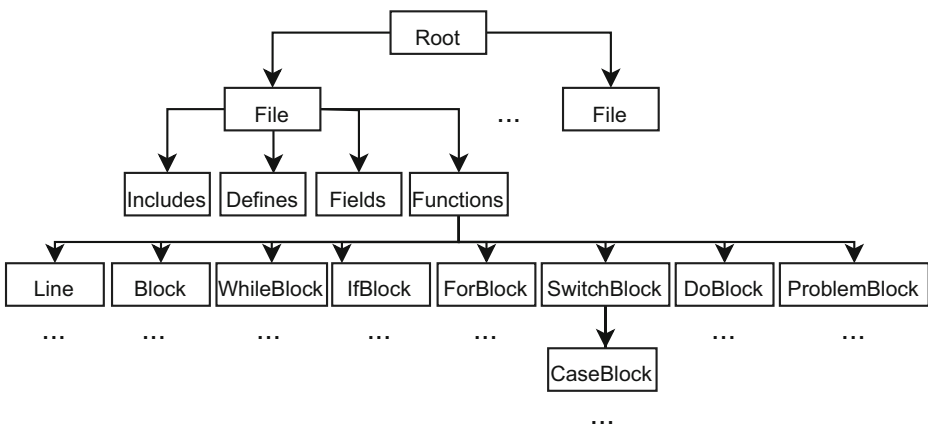


Fig. 2 Tree structure of the new ECCO plug-in for parsing C source code

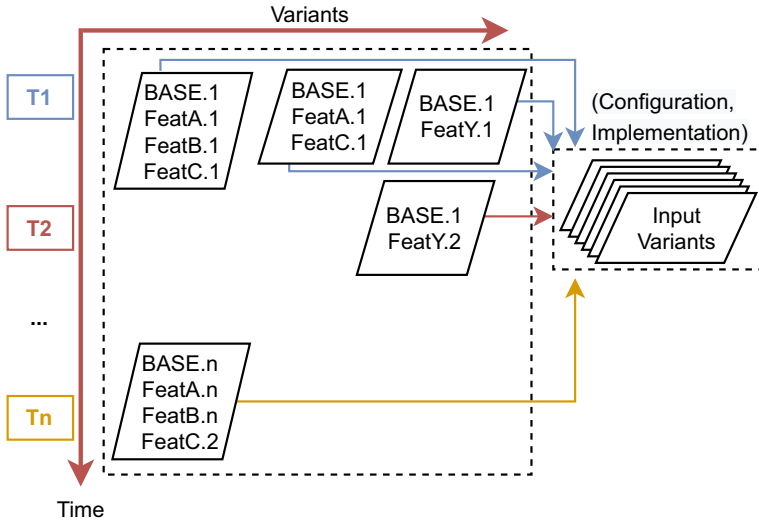


Fig. 3 Input variants of our approach for the two dimensions of variability analysis

4.2.1 Trace Computation

Based on the aforementioned data structures, we now explain how the traces and presence conditions are computed based on the running example shown in Table 1. This example was extracted from a code snippet from file *connect.c* of the commit [c65f56ae](https://gitlab.com/libssh/commit/c65f56ae)¹⁴ (Listing 1). Listing 2 shows the code of a variant v_1 containing features BASE and HAVE_POLL (Lines 1, 2, 4, 8, 9, 11, 12 and 23 from Listing 1) at point T1. Listing 3 shows the code of a variant v_2 containing BASE and absence of the features HAVE_SELECT and HAVE_POLL (Lines 1, 2, 6, 8, 9, 20, 21 and 23 from Listing 1) at point T1. Listing 4 shows the code of a variant v_3 containing the features BASE and HAVE_SELECT (Lines 1, 2, 6, 8, 9, 14-18 and 23 from Listing 1 at point T1). Listing 5 shows the code of a variant v_4 containing BASE at point T1 and HAVE_SELECT at point T2, where the Line 15 from Listing 1 changed.

Presence Conditions We compute the presence condition C for every artifact a in the form of a disjunctive normal form (DNF) formula, whose literals are features, i.e., a set of feature revisions as we will show. A DNF formula is a disjunction of clauses, where a clause is a conjunction of literals. We treat presence conditions as a set of such clauses. Every clause can be considered a feature interaction, i.e., a static interaction of the features contained in the clause. This aligns with previous research in feature algebra (Liu et al. 2006), feature location (Linsbauer et al. 2013), and the analysis of variable systems (Fischer et al. 2016; Angerer et al. 2019). We denote the set of all conjunctive clauses that can be formed given a set of feature revisions $v.F$ of variant v as $clauses(v.F)$.

Whether a clause c is part of a presence condition C for an artifact a depends on five intuitive rules that have already been proven to work properly for feature location (Michelon et al. 2019). Given two variants v_1 and v_2 of a system:

1. Common artifacts in v_1 and v_2 likely trace to common features.

¹⁴<https://gitlab.com/libssh/commit/c65f56ae>

Table 1 *Input:* Set of variants $V = \{v_1, v_2, v_3, v_4\}$ and their respective feature revisions $v_i.F$

Variant v_i	Feature revisions $v_i.F$	Artifacts $v_i.A$
v_1	{HAVE_POLL ₁ , BASE ₁ }	Listing 2
v_2	{BASE ₁ }	Listing 3
v_3	{HAVE_SELECT ₁ , BASE ₁ }	Listing 4
v_4	{HAVE_SELECT ₂ , BASE ₁ }	Listing 5

```

1 #include <stdio.h>
2 int ssh_fd_poll(SSH_SESSION *session){
3 #ifdef HAVE_POLL
4     struct pollfd fdset;
5 #else
6     struct timeval sometime;
7 #endif if(session->data_to_read)
8     return(session->data_to_read);
9 #ifdef HAVE_POLL
10    fdset.fd=session->fd;
11    (...)
12 #elif HAVE_SELECT
13    (...)
14    if (select(session->fd+1, &descriptor, NULL, NULL, &sometime)<0)
15    {
16        ssh_set_error(NULL, SSH_FATAL, "select: %s", strerror (
17        errno))
18    ;
19    return -1;
20 }
21 #else
22 #error This system does not have poll() or select()
23 return 0;
24 #endif
25 }

```

Listing 1 Code snippet from file *connect.c* from LibSSH in commit c65f56ae

```

1 #include <stdio.h>
2 int ssh_fd_poll(SSH_SESSION *session){
3     struct pollfd fdset;
4     if(session->data_to_read)
5         return(session->data_to_read);
6     fdset.fd=session->fd;
7     (...)
8 }

```

Listing 2 Variant v_1 : BASE and HAVE_POLL at point T1

```

1 #include <stdio.h>
2 int ssh_fd_poll(SSH_SESSION *session){
3     struct timeval sometime;
4     if(session->data_to_read)
5         return(session->data_to_read);
6     #error This system does not have poll() or select()
7     return 0;
8 }

```

Listing 3 Variant v_2 : BASE at point T1

```

1 #include <stdio.h>
2 int ssh_fd_poll(SSH_SESSION *session){
3     struct timeval sometime;
4     if(session->data_to_read)
5         return(session->data_to_read);
6     (...)
7     if(select(session->fd+1, &descriptor, NULL, NULL, &sometime)<0){
8         ssh_set_error(NULL, SSH_FATAL, "select: %s", strerror (
9             errno)
10        );
11        return -1;
12    }
13 }

```

Listing 4 Variant v_3 : BASE and HAVE_SELECT at point T1

```

1 #include <stdio.h>
2 int ssh_fd_poll(SSH_SESSION *session){
3     struct timeval sometime;
4     if(session->data_to_read)
5         return(session->data_to_read);
6     (...)
7     if(select(session->fd + 1, &rdes, &wdes, &edes, &sometime)<0) {
8         ssh_set_error(NULL, SSH_FATAL, "select: %s", strerror (
9             errno)
10        );
11        return -1;
12    }
13 }

```

Listing 5 Variant v_4 : BASE at point T1 and HAVE_SELECT at point T2

2. Artifacts in v_1 and not v_2 *likely* trace to features that are in v_1 and not v_2 , and vice versa.
3. Artifacts in v_1 and not v_2 *cannot* trace to features that are in v_2 and not v_1 , and vice versa.
4. Artifacts in v_1 and not v_2 *can at most* trace to features that are in v_1 , and vice versa.
5. Artifacts in v_1 and v_2 *can at most* trace to features that are in v_1 or v_2 .

In our work, we build upon these rules and extend them to feature revisions. In the following, we first discuss the rules based on features, ignoring revisions for the time being. We now describe the criterion and two resulting equations based on the aforementioned five rules for including a clause in a presence condition, which composes the traces between artifacts and feature revisions.

Criterion for the Inclusion of a Clause in a Condition. For a clause c to be contained in a presence condition C of an artifact a , the artifact a must be contained in every variant $v \in V$ that contains the clause c ($c \in \text{clauses}(v.F)$) and there must be at least one variant in V that contains clause c .

$$c \in C \Leftrightarrow (\forall v \in V : c \in \text{clauses}(v.F) \implies a \in v.A) \wedge (\exists v \in V : c \in \text{clauses}(v.F)) \quad (1)$$

Criterion for Likely Clause. Our approach additionally provides a smaller and more specific set of clauses C' , that is a subset of C , to which the artifacts are more likely tracing than to others. This is based on our observation that, in practice, presence conditions with a logical OR between features are much less likely to occur than conditions with a logical AND (Michelon et al. 2019). Therefore, a clause c' is contained in the set of likely clauses C' if all variants that have clause c' also have artifact a (inclusion criterion (1)). In addition, all variants that have artifact a also have clause c' (additional criterion).

$$c' \in C' \Leftrightarrow (\forall v \in V : c' \in \text{clauses}(v.F) \iff a \in v.A) \wedge (\exists v \in V : c' \in \text{clauses}(v.F)) \quad (2)$$

Adding Revisions Extending the previous ideas to revisions is then straightforward. Only one revision of a feature can be present in any given variant. In other words, if a feature f is present in a variant v , it is present in exactly one revision r . Therefore, the set of revisions of a feature literal in a clause is the union of all revisions r of feature f that were present when the artifact a was present. Literals in clauses of a presence condition now do not refer to single features anymore but to a set of feature revisions.

Steps for Trace Computation Algorithm 1 shows the steps of the trace computation. This algorithm receives as *input* a set of variants V and computes the sets of all clauses C (Line 2) and all artifacts A (Line 3) in the input variants V . Subsequently, it computes for every artifact $a \in A$ (Line 5) a trace t with conditions C' and artifact a (Line 19) that is added to the set of traces T (Line 20) that is returned (Line 22). The set of clauses C' receives all clauses $c \in C$ that satisfy the inclusion criterion of likely clauses in (2) (Lines 7-11). If there are no such traces (Line 12) it receives all clauses $c \in C$ that satisfy the regular inclusion criterion in (1) (Lines 13-17).

Algorithm 1 Trace computation.

```

1: function COMPUTETRACES( $V$ )
2:    $C \leftarrow \bigcup_{v \in V} \text{clauses}(v.F)$ 
3:    $A \leftarrow \bigcup_{v \in V} \text{clauses}(v.A)$ 
4:    $T \leftarrow \{\}$ 
5:   for each  $a \in A$  do
6:      $C' \leftarrow \{\}$ 
7:     for each  $c \in C$  do
8:       if  $(\forall v \in V : c \in \text{clauses}(v.F) \iff a \in v.A)$  then
9:          $C' \leftarrow C' \cup \{c\}$ 
10:      end if
11:    end for
12:    if  $C' = \{\}$  then
13:      for each  $c \in C$  do
14:        if  $(\forall v \in V : c \in \text{clauses}(v.F) \implies a \in v.A)$  then
15:           $C' \leftarrow C' \cup \{c\}$ 
16:        end if
17:      end for
18:    end if
19:     $t \leftarrow (C', a)$ 
20:     $T \leftarrow T \cup \{t\}$ 
21:  end for
22:  return  $T$ 
23: end function

```

Now, to better understand the definitions, let us recall the running example (Listing 1). The computation of traces consists of computing new, or updating existing, ones after the artifacts alignment for equivalence. The variants v_3 and v_4 have equivalent artifacts under the function `ssh_fd_poll(SSH_SESSION *session)`. When the variant is used as input, the comparison for artifacts equivalence is performed between the Lines from Listings 4 and 5. Then, as feature revision location is an incremental process, the existing traces in the repository from the equivalent artifacts have to be updated. The updated traces thus include this new feature revision in the clauses of the traces containing the equivalent artifacts. In the incremental feature revision location, after input variant v_4 , the old artifact in Line 7 from variant v_3 is traced solely to the feature revision `HAVE_SELECT1`. Also, a new trace is created for the new code in Line 7 from Listing 5 to the new feature revision `HAVE_SELECT2`.

Mapping feature revisions to artifacts can be challenging when a set of variants is not sufficient to determine a unique set of traces. We thus have to consider more restrictive traces by adding negated features in presence conditions to represent artifacts of a variant that do not appear when specific features are present. A feature absent in a configuration can influence the implementation of a variant (Liu et al. 2006).

Our approach uses presence conditions to map artifacts and feature revisions with negated features when a specific artifact only exists in a variant with a specific feature absent in the configuration. We use logical negation (\neg) to express an absent feature. Despite a variant configuration contains a feature either present in a specific revision or simply absent, our approach can trace artifacts with presence conditions containing positive and negated features. The final set of clauses $\text{clauses}(v.F)$ contains all positive features and negated

features. Negated features in presence conditions are not labeled with a revision, which indicates that a specific artifact only appears in a variant when the feature negated in the presence condition is not present in the variant configuration. On the other hand, presence conditions containing positive features indicate that an artifact is present in a variant if at least one of the clauses is satisfied with the set of feature revisions of a variant configuration. For the last assumption, it does not matter if features of the other clauses of a presence condition are absent in the configuration.

In our example, a developer does not have to indicate the absence of the features HAVE_SELECT and HAVE_POLL with a logical negation (\neg) in an input configuration of a variant (such as variant v_2), as including BASE in the configuration is sufficient. However, when preprocessing the variant, our approach computes traces for the specific artifacts that do not belong to the feature BASE, i.e., the core of the system, but are part of a variant when a specific feature is not part of the configuration. For example, in an `#if` and `#else` conditional compilation, the `#else` block artifacts of a system will be part of a variant only when a feature of the `#if` is absent in the configuration, similar to an `#if ! (Feature)`. However, including BASE in the configuration cannot guarantee that the `#else` part will be in the variant as the feature from the `#if` part also has to be absent in the configuration. This is why only adding positive feature revisions in clauses is not sufficient for creating traces to artifacts that are part of a variant only when specific features are present and specific features are absent. In such cases, different possible traces can affect the variants created in the future.

The output of the feature revision location for our running example shown in Table 2 contains all clauses that satisfy the criterion for inclusion, even if initially redundant. For example, the condition in t_1 could be simplified to just HAVE_POLL₁. However, since the input variants were not sufficient to be certain that the actual condition cannot be, HAVE_POLL₁ \wedge \neg HAVE_SELECT it is still included in the condition.

4.2.2 Optimization Aspects

We do not consider every artifact individually, but cluster artifacts that never appear without each other in any variant and assign presence conditions to those clusters instead of every individual artifact. For example, since the artifacts from Lines 1-2, 8-9 and 23 in our running example in Listing 1 always appear together and never without each other, they are grouped in one artifact cluster instead of treating them individually.

Table 2 Output: Set of Traces $T = \{t_1, t_2, t_3, t_4, t_5, t_6, t_7\}$

Trace t_i	Presence condition $t_i.C$	Artifacts $t_i.A$
t_1	$F3_1 \vee (F3_1 \wedge F1_1) \vee (F3_1 \wedge \neg F2) \vee (F3_1 \wedge F1_1 \wedge \neg F2)$	Lines 4,11-12 (L 1)
t_2	$F2_1 \vee (F2_1 \wedge \neg F3) \vee (F2_1 \wedge F1_1) \vee (F2_1 \wedge F1_1 \wedge \neg F3)$	Line 15 (L 1)
t_3	$F1_1 \wedge \neg F2 \wedge \neg F3$	Lines 20-21 (L 1)
t_4	$F2_{1\vee 2} \vee (F2_{1\vee 2} \wedge \neg F3) \vee (F2_{1\vee 2} \wedge F1_1) \vee (F2_{1\vee 2} \wedge F1_1 \wedge \neg F3)$	Lines 14,16-18 (L 1)
t_5	$F1_1$	Lines 1-2,8-9,23 (L 1)
t_6	$F1_1 \wedge \neg F3$	Line 6 (L 1)
t_7	$F2_2 \vee (F2_2 \wedge \neg F3) \vee (F2_2 \wedge F1_1) \vee (F2_2 \wedge F1_1 \wedge \neg F3)$	Line 7 (L 5)

L = Listing; BASE = F1; HAVE_SELECT = F2; HAVE_POLL = F3

We use counters to evaluate the above criterion for inclusion of clauses (1) in presence conditions. For every clause c , we count in how many input variants it was contained, for every artifact cluster a in how many input variants it was contained, and for every pair (c, a) of clause and artifact cluster in how many input variants both were contained together. This has the advantage that it works incrementally, i.e., new input variants can be added whenever necessary, simply by increasing the respective counters. Hence, already computed traces do not have to be recomputed when a new variant is encountered. Instead, the counters are simply increased and the existing presence conditions are trimmed by removing the clauses for which the above conditions do not hold anymore.

Table 3 presents the counters of our running example that match the set of variants V in Table 1. The rows list the nine artifact clusters with the total number of appearances in variants. The columns list (a subset of) the clauses $c_i \in \bigcup_{v \in V} \text{clauses}(v.F)$ with the total number of appearance in variants, sorted by the number of literals, i.e., interacting features first in total without considering revisions, and then per revision. Each cell contains the number of times that the artifact cluster and the clause appear together in a variant. For example, artifacts from Lines 1, 2, 8, 9 and 23 in Listing 1 appear in four variants. The clause $F1$, which represents the feature revision BASE, also appears in four variants. Finally, the artifacts and the clause appear together also in four variants. Therefore, the criterion for likely clauses (2) is satisfied. The cells in Table 3 highlighted with gray color indicate pairs of feature revision(s) and artifact(s) that always appear together in input variant(s).

The correct presence conditions shown in Table 2 can only be created with the additional criterion for likely clause (2). This shows why trivial, i.e., less restrictive presence conditions are not complete and can result in different traces. For example, trace t_1 has a presence condition containing a disjunction of four clauses: $(\text{HAVE_POLL}_1) \vee (\text{HAVE_POLL}_1 \wedge \neg \text{HAVE_SELECT}) \vee (\text{BASE} \wedge \text{HAVE_POLL}_1) \vee (\text{BASE} \wedge \text{HAVE_POLL}_1 \wedge \neg \text{HAVE_SELECT})$. The first clause is obtained by the criterion for likely clause because artifacts from Lines 4,11-12 in Listing 1 are contained in every variant that contains the feature HAVE_POLL_1 and all variants that have artifacts from Lines 4,11-12 in Listing 1 also have the feature HAVE_POLL_1 . The second clause is also created with the criterion for likely clause, where all variants that have artifacts from Lines 4,11-12 in Listing 1 also have feature BASE_1 . Although artifacts from Lines 4,11-12 are not from BASE_1 , BASE_1

Table 3 Implementation Example: subset (cut off right) of counters for artifact clusters (rows) and clauses (columns)

		$F1$	$F2$		$F3$	$F1 \wedge F2$		$F1 \wedge F3$
		4	2		1	2		1
		$F1_1$	$F2_1$	$F2_2$	$F3_1$	$F1_1 \wedge F2_1$	$F1_1 \wedge F2_2$	$F1_1 \wedge F3_1$
		4	1	1	1	1	1	1
Lines 1-2,8-9,23 (L1)	4	4	1	1	1	1	1	1
Lines 4,11-12 (L1)	1	1	0	0	1	0	0	1
Line 6 (L1)	3	3	1	1	0	1	1	0
Line 7 (L5)	1	1	0	1	0	0	1	0
Line 15 (L1)	1	1	1	0	0	1	0	0
Lines 14,16-18 (L1)	2	2	1	1	0	1	1	0
Lines 20-21 (L1)	1	1	0	0	0	0	0	0

L = Listing; BASE = F1; HAVE_SELECT = F2; HAVE_POLL = F3

was present in the variants containing these artifacts. That is what we show in the counter table for storing this information (Table 3), while the third clause is created because all absent features for a specific artifact are negated in our approach. Thus the third clause ($HAVE_POLL_1 \wedge \neg HAVE_SELECT$) has the feature $HAVE_SELECT$ negated because this feature does not appear in variants with the artifacts from Lines 4,11-12 in Listing 1. The fourth clause is the most restrictive condition, combining the previous clauses.

Analyzing the second row of Table 3 shows that the artifacts from Lines 4, 11-12 are present in one variant, where $BASE_1$ is present. However, $BASE_1$ is present in four variants. When looking at the other columns of the second row of the Table 3, we can see that the feature revision $HAVE_POLL_1$ is also present in one variant, only in the one containing the artifacts from Lines 4, 11-12 from Listing 1. Therefore, we know the feature revision $HAVE_POLL_1$ must be traced to Line 4 and our final presence condition contains the feature revision $BASE_1$, as Line 4 appears only once and in a variant containing also the feature $BASE_1$ in its configuration. Thus, the first clause is less restrictive and the fourth clause is the most restrictive condition of the presence condition of the trace t_1 . If another input variant would exist with only the feature revision $HAVE_POLL_1$, and containing Lines 4, 11-12 from Listing 1, the trace t_1 would be refined and its presence condition would be $(HAVE_POLL_1) \vee (HAVE_POLL_1 \wedge \neg HAVE_SELECT) \vee (HAVE_POLL_1 \wedge \neg BASE) \vee (HAVE_POLL_1 \wedge \neg HAVE_SELECT \wedge \neg BASE)$. Having a clause in a presence condition with $\neg BASE$ does not mean that this feature must not exist in the configuration of a variant containing the respective trace artifact, but that $BASE$ was absent in a variant where the respective trace artifact appears and was thus not mapped to the artifact. Hence, $BASE$ would not be a mandatory feature for having these artifacts in a variant.

4.3 Variant Composition

For a given configuration containing a set of feature revisions, we compute a checkout operation, similar to the checkout in a VCS (Conradi and Westfechtel 1998). The checkout operation retrieves a working copy of the content from a repository. Thus, the checkout operation joins the artifacts of the feature revisions from a repository in order to compose a system variant.

Composition We compose a variant v from a set of traces T given a configuration F (set of feature revisions). First, the set of traces T' selected from the set of all traces T :

$$T' = \{t \mid t \in T \wedge clauses(v.F) \cap t.C \neq \emptyset\} \tag{3}$$

The final resulting variant v is then given as $v = (F, A)$, where A is the set of artifacts:

$$A = \bigcup_{t \in T'} t.A \tag{4}$$

The developer is responsible for selecting a valid configuration to compose a valid variant. We do not consider variability models, which define a set of choices and their dependencies for obtaining configurations (Rabiser et al. 2012), but rather focus on feature revision location and variant composition. The composition thus generates a product variant and a file with hints of the traces containing possible surplus and/or missing clauses used to compose the variant. With these hints, developers can analyze which artifacts may need to be added and/or removed for completing the product variant. The file with hints contains the trace identifier (hash code), which can be used to look in the ECCO repository which artifacts belong to a stored trace.

As an example, consider selecting feature revisions $HAVE_POLL_1$, $HAVE_SELECT_1$ and $BASE_1$ to compose a variant. The traces t_1, t_2, t_4, t_5 (Table 2) corresponding to these feature revisions will be retrieved from the repository and their artifacts will be joined in order to create a variant. These traces are considered in the set of traces T' because at least one clause of the presence condition is satisfied (clauses are split with the \vee logical operator). For example, t_1 contains a clause with $F3_1$, which represents the feature $HAVE_POLL$ and is one of the features of the configuration. However, this combination of feature revisions has feature interactions, as we can see in Listing 1. Then, when composing the variant we also get the hints, which we will explain next.

Computation of Hints To provide the artifacts for a variant, we retrieve the existing traces with at least one clause from the disjunction of clauses in a presence condition containing the feature revision(s) of the configuration. Traces containing clauses with negated features that are in the configuration are not considered in the set of traces T' selected to compose a variant, i.e., the artifacts of a trace will not be included in the variant, while every other trace with the positive feature(s) will. If a configuration contains a feature revision that does not exist in the repository, the composed variant will contain only the artifacts of other existing feature revisions in the configuration. Then, with the composed variant, a hint will be retrieved with *Missing Clauses* because no traces exist for unknown feature revisions in the repository. Also, a missing trace can be retrieved if a feature interaction of a configuration is missing in the existing set of traces because no trace containing the needed implementation yet exists. Therefore, the set of potential *Missing Clauses* H^- for a composed variant v with a configuration F is:

$$H^- = clauses(v.F) \setminus \bigcup_{t \in T'} t.C \quad (5)$$

From the set of selected clauses for composition, we can determine one or more *Surplus Clauses* H^+ as follows:

$$H^+ = \bigcup_{t \in T'} t.C \setminus clauses(v.F) \quad (6)$$

From a trace containing multiple clauses, when a clause of a trace contains a feature revision that should be part of a variant and some clauses of the trace contain feature revisions that are not present in the configuration, our approach issues a hint on surplus clauses. This means that all artifacts of the trace were added as artifacts of the variant. However, not all clauses contained in the trace contain only the feature revisions of the configuration, which can result in potential surplus artifacts in the variant from the other feature revisions and show possible feature interactions and/or dependencies due to the artifacts in common. As explained before, our approach computes a presence condition for a (set of) artifact(s) containing a disjunction of clauses. The trace is added to compose a product variant for every presence condition containing a feature revision in one of its clauses, unless there is at least one feature existing in the configuration that is negated in the less restrictive clause of the presence condition. In this case, the trace is not added for composing a variant, which can have missing artifacts.

The hints retrieved by our approach will inform the surplus clauses followed by the trace identifier that can have artifacts surplus and should not be in the variant. The hints retrieved by the new configuration ($HAVE_SELECT_2$, $HAVE_POLL_1$, $BASE_1$) to compose a variant, used as example, contains some of the clauses of the trace $t1$ and $t4$ as *Surplus Clauses*. Trace $t1$ is selected to compose the variant because it contains a clause with $HAVE_POLL_1$

and another with $\text{BASE}_1 \wedge \text{HAVE_POLL}_1$ from the existing ones in the presence condition of $t1$. However, two clauses are *Surplus Clauses*: $(\text{HAVE_POLL}_1 \wedge \neg \text{HAVE_SELECT})$ and $(\text{BASE}_1 \wedge \text{HAVE_POLL}_1 \wedge \neg \text{HAVE_SELECT})$, as they correspond to artifacts that never appear together within the artifacts of all existing revisions of the feature HAVE_SELECT . The same happens with trace $t4$, which results in hints for possible surplus artifacts, because some of the artifacts of the input variant containing the feature revision HAVE_SELECT_2 did not appear with some artifacts of the input variant containing the feature revision HAVE_POLL_1 . The hints can help developers that need to manually remove part of the artifacts of a specific trace from the composed variant, if the new combination of feature revisions has conflicts when used together.

5 Evaluation

We now present the research questions and the methodology adopted for evaluating the *ECSEST* approach. This evaluation covers both feature revision location in variants of software systems evolving in space and time as well as SPL evolution by reusing the located feature revisions for automatically composing new variants. We introduce the input dataset, i.e., the characteristics of our subject systems. Then, we explain the process adopted to obtain a ground truth dataset used for evaluating *ECSEST*'s efficiency for locating feature revisions and composing variants. Finally, we describe the metrics used to evaluate *ECSEST*.

5.1 Research Questions

The evaluation of *ECSEST* was guided by five research questions (RQs):

RQ1. To what extent do features evolve in space and time? This RQ investigates how features evolve in space and time in real systems to show the practical relevance and implications of our approach for evolving software systems in space and time at the level of feature revisions.

RQ2. To what extent does *ECSEST* support feature revision location of existing variants that evolved in space and time? We evaluate how effective *ECSEST* is for locating feature revisions in existing families of software systems that evolved over time.

RQ3. How effective is *ECSEST* for composing new variants with feature revisions? We investigate if *ECSEST* can effectively compose new variants by joining artifacts traced to feature revisions with our feature revision location approach from a set of system variants that have evolved in space and time.

RQ4. How useful are the hints suggested by *ECSEST* for completing new variants and finding feature interactions when creating new configurations? We estimate how helpful the hints provided by *ECSEST* are for the manual completion of new variants.

RQ5. What is *ECSEST*'s performance for extracting feature revisions and composing variants? This RQ answers the execution time of our approach for performing feature revision location per variant and for composing a variant.

5.2 Method

Figure 4 illustrates the methodology we followed to evaluate *ECSEST*. We investigated both the feature revision location and the composition of variants with feature revisions. We started by mining ground truth variants (Step 1) from feature revisions in preprocessor-based SPLs in VCSs (cf. Section 5.4). We then applied our feature revision location approach to

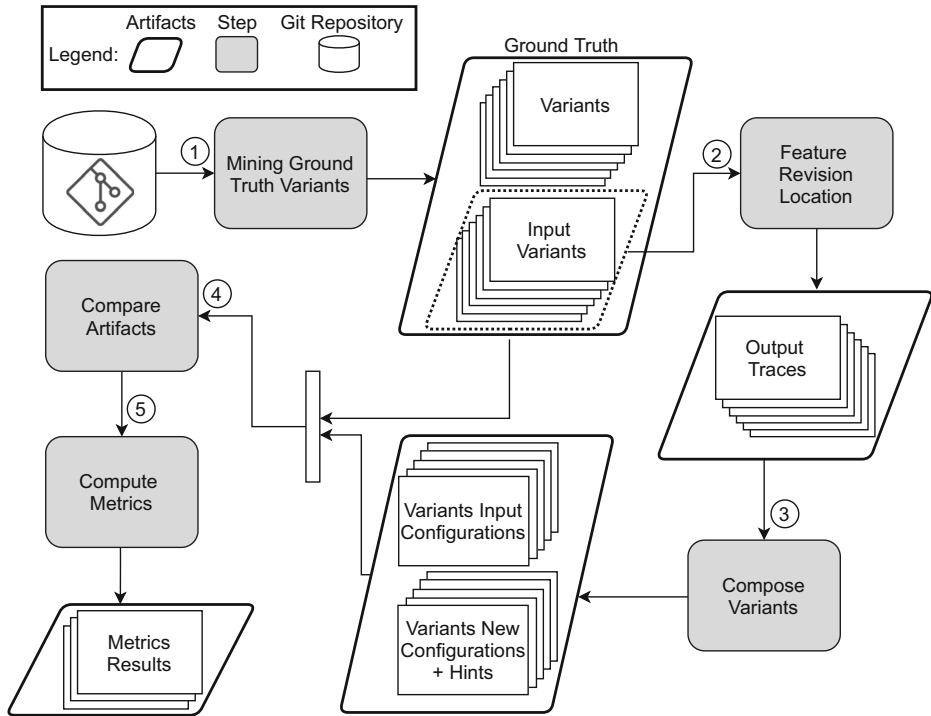


Fig. 4 Methodology for evaluating ECSEST to support software systems evolving in space and time

input product variants obtained from the ground truth generation (Step 2). The input variants are the ones generated from existing configurations, and the remaining ground truth variants are the ones generated with new configurations, which we used later on to compare the composed variants with new configurations. For variants with new configurations, we randomly choose a set of feature revisions existing for each point in time. The step of locating feature revisions was performed incrementally with the input variants. Thus, as long as we had different input variants, we used them for locating feature revisions with ECSEST, which continuously created new and/or refined existing traces.

After locating the feature revisions from all existing input variants, we used the computed traces to compose variants with existing configurations and with a new combination of feature revisions (Step 3) by joining the artifacts of the desired feature revisions. Next, we compared the composed variants with the corresponding ground truth variants containing the same configuration (Step 4). The comparison of variants was performed by comparing each composed artifact with each ground truth artifact both file-by-file and line-by-line (cf. Section 5.5). To compute differences of the artifacts of input and composed variants, we implemented a Java program for performing the comparison operations between textual data using a Java diff library¹⁵. Finally, we computed metrics (Step 5) to quantify missing relevant information or surplus information retrieved in relation to the variants composed from existing configurations (cf. Section 5.5). We also computed metrics for the hints retrieved

¹⁵<https://github.com/java-diff-utils/java-diff-utils>

when composing new configurations of possible surplus or missing source code of feature revisions in new configurations of variants composed.

5.3 Dataset

The evaluation of the proposed approach relies on six open source preprocessor-based SPLs (Liebig et al. 2010) using the VCS Git. Table 4 presents details of the SPLs: (i) Marlin, a variant-rich open-source embedded firmware for 3D printers¹⁶; (ii) LibSSH, a multiplatform C library implementing the SSHv2 protocol on client and server side¹⁷; (iii) SQLite, a library implementing an SQL database engine¹⁸; (iv) Irssi, an internet relay chat client program for Linux¹⁹; (v) Bison, a general-purpose parser generator²⁰; and (vi) Curl, a command-line tool for transferring data specified with URL syntax. We try to reduce bias by choosing different application domains. Furthermore, each system has a considerable history of development and use in research (Gargantini et al. 2016; Ha and Zhang 2019; Krüger et al. 2018; Krüger et al. 2019; Liebig et al. 2010; Medeiros et al. 2018; Vale and Almeida 2019). Moreover, we choose systems of different sizes, which we measured by counting the total number of lines of code of their last release (excluding blank lines and comments). We used variants from the first Git commits from the main trunk ordered by the date of each system to avoid bias in choosing a specific interval of commits.

The number of variants we mined (last column in Table 4) is the largest one that we could use as input for each subject system given the memory limitation of the used Java Virtual Machine (JVM) to store and manipulate data. Specifications of the machine used to run the experiments are given in Section 5.5. The number of variants used as input is influenced by the number of artifacts of a system and the degree of artifacts evolution, which determines how many traces and feature revisions have to be stored and manipulated. Therefore, for our evaluation we used input variants from a large number of commits and of more than one release for some of the systems. This is a considerable extension to our previous work (Michelon et al. 2020d), since we now apply *ECSEST* to more systems, covering more Git commits and many more variants with different features at different points in time for each system. The variability thus comes from the Git commits. The changes vary from lines in a file to multiple files affected. Some commits introduce new feature revisions, some commits change existing feature revisions, while some commits introduce new feature revisions and change existing ones in parallel. This mining process is presented in Section 6.1.

5.4 Mining Ground Truth Variants from Evolution in Space and Time

Our evaluation needs ground truth variants containing feature revisions, i.e., variants that contain features with different implementations at different points in time. We thus extracted variants of preprocessor-based SPLs in VCSs whenever a feature evolved in time, i.e., was changed via a Git commit (Michelon et al. 2020a). Figure 5 illustrates the time dimension (Git commits) on the y-axis and the space dimension representing the multiple features that originated from multiple variants, which are in the x-axis. The different colors of the

¹⁶<https://github.com/MarlinFirmware/Marlin>

¹⁷<https://gitlab.com/libssh/libssh-mirror>

¹⁸<https://github.com/sqlite/sqlite>

¹⁹<https://github.com/irssi/irssi>

²⁰<https://github.com/akimd/bison>

Table 4 Overview of the subject systems

System	Since	LoC	Commits	Features	Feature revisions	Input variants
Marlin	2011	281355	52	151	106	191
LibSSH	2005	110590	400	44	538	577
SQLite	2000	173714	337	36	388	424
Irssi	2007	85325	400	30	414	441
Bison	2002	39904	240	134	272	310
Curl	1999	22490	350	84	422	485

edges represent different points in time of features. For every change in a Git commit, we mined feature revisions that were then used to preprocess the variants. Finally, we used the resulting variants as ground truth to represent software systems with different features' artifacts at different points in time.

Although our approach can locate feature revisions in any type of artifact of system variants, even without a variability mechanism, we choose preprocessor-based SPLs in VCSs as input variants because they are widely used to deal with system evolution in space and time (Berger et al. 2019). Therefore, every time a feature changes in a Git commit we generate variants containing a new feature revision for simulating our incremental step of locating feature revisions whenever a feature has a new implementation. In summary, our approach for generating the ground truth consists of getting changes from one commit to another for a set of Git commits. This approach can be computationally expensive but is well suited for precisely locating feature revisions. To cover all changes, a set of configurations is determined by a constraint satisfaction problem (CSP) solver. For each configuration composed of external features, we preprocess the version of the system in the specific commit, which results in an input variant for evaluating *ECSEST*. Next, we explain this process in detail.

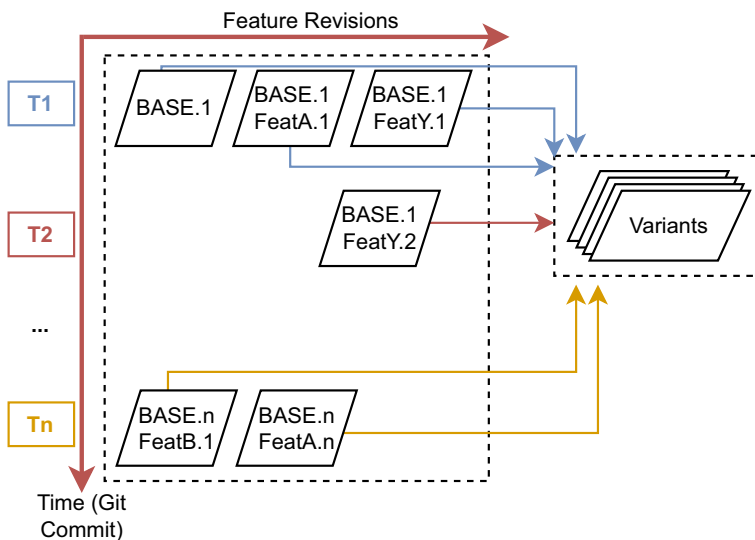


Fig. 5 The changes of Git commits represent the evolution of features in time in VCSs. They resulted in a set of feature revisions, which are used to create ground truth variants for the *ECSEST* evaluation

We use the example in Fig. 6, which contains a corner case with a feature interaction and a feature implication to explain the methodology for mining the ground truth variants. Let us consider the code of the file *main.c* presented in Fig. 6 before performing the change in Line 12 at the point in time called T1. Then, changes of a second commit (point in time T2) can be seen in Line 12 of the file *main.c* in Fig. 6. We identify the possible features in these two points in time. In this example, three features are introduced in point T1 (BASE, A, Y) and one existing feature changed in point T2 (Y revision 2). Based on that, the mining process is as follows.

Identifying feature literals As our target systems do not have a variability model available, we used the following strategy to identify possible features. We first classified all feature literals, i.e. macros annotated to characterize features of the system along all Git commits analyzed. For this, we distinguished external, internal, and transient feature literals. External feature literals can only be set externally to configure variants from the compiler command line. In Fig. 6, the feature literals A and Y are external. Internal feature literals are defined at some point in the code via a `#define` directive. Thus, we can see in Fig. 6, that the feature literals B and C defined in *main.c* as well as the feature literals X and Z defined in *header.h* are internal.

We considered feature literals as system features only if they were external in all Git commits analyzed. We cannot ensure that all identified external feature literals are actually features of the system. However, according to Berger et al. (Berger et al. 2015), features are also used for testing and debugging purposes. In addition, our approach enables the manual setting of system features if the set of features is known. Our ground truth generator

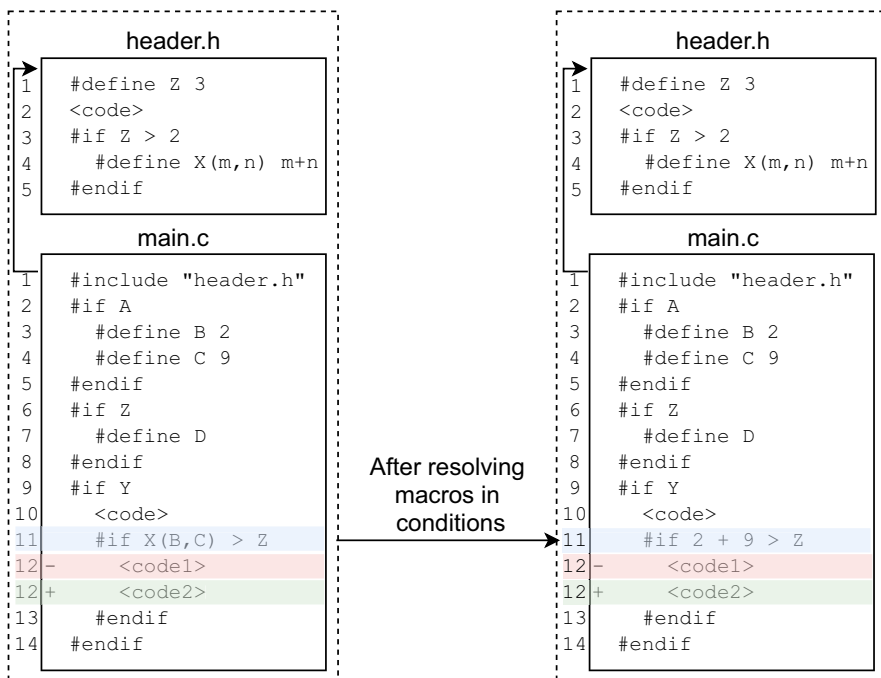


Fig. 6 Mining feature revisions from changes in time in preprocessor-based SPLs

approach is limited to systems that do not consider dependencies in Kconfig and Makefiles such as the Linux Kernel system (Michelon et al. 2020a).

Resolving macros in conditions For each analyzed Git commit, we started preprocessing the annotated code to find macros that can accept parameters and return values. The output of this step is the code from the specific commit with all macros in conditions resolved, i.e., the macro code is expanded to the degree where the conditions of the conditional statements only consist of feature literals. This step is necessary because we need only macros and their values in the expressions of conditional blocks to correctly collect all possible features from conditions. Obtaining these values from expressions and functions is important to build up the constraints and to retrieve a possible solution via a CSP Solver. After expanding macros in conditions, all `#define` and `#include` statements and conditional blocks remained in the code, as they can modify the resulting code of the variants. On the right of Fig. 6, we see that the highlighted Line 11 is the only one that changed after this step replacing `#if X(B, C) > Z` with `#if 2 + 9 > Z`.

Computing changes For each Git commit n we created a tree structure for representing variability in source code, as shown in Fig. 7. Files at a certain point in time are represented either by *SourceNodes* or *BinaryNodes*. The *SourceNodes* contain child nodes each with the content of a source code file, e.g., `.c/ .cpp`. A *SourceNode* has as a root node `BASE` that emulates the feature `BASE`, which contains *ConditionalNodes* as much as needed to represent each `#ifdef` in a file. *DefineNodes* represent the location in a file of `#define` and `#undef` preprocessor statements, while *IncludeNodes* represent the `#include` preprocessor statements in a file. The tree nodes are used to determine the differences between an actual commit and its previous one according to Git-diff²¹. The adopted tree structure has a higher level of abstraction, i.e., for every annotated block, a child stores its content in its respective node category, e.g., conditional nodes, define nodes, and include nodes. This makes our mining process computationally less expensive. We adopted the changes at Git-diff granularity, i.e., files and lines, to be able to easily inspect the correctness of the generated ground truth variants according to changes of features annotated in Git VCS.

The choice of inspecting changes from consecutive commits was to avoid bias in choosing specific commits to generate variants as any change results in new feature revision(s) as input for our approach to generate variants. Therefore, in case of the first Git commit of the project, we consider all files inserted as the difference. From the differences, we can get the tree node reflecting the changes. In case any external feature changed or differences are detected in non-code files, e.g., binary, `BASE` is considered the changed feature, i.e., for every code added/removed in the body of the project that does not belong to an external feature the root feature, i.e., `BASE` is considered as the changed node. Figure 6 shows two files, the header file (on top of the figure indicated by an arrow) and the file containing 14 lines on the bottom of the figure. At point T1 we have these two files, and at point T2 the main file (the file on the bottom of the Fig. 6) has been changed in Line 12.

Computing configurations Every changed node was then used to generate a variant containing the code activated by this node. We used the Choco solver²² library to provide the first possible solution for a given constraint to activate the conditional blocks. To find a

²¹<https://git-scm.com/docs/git-diff>

²²<https://github.com/chocoteam/choco-solver>

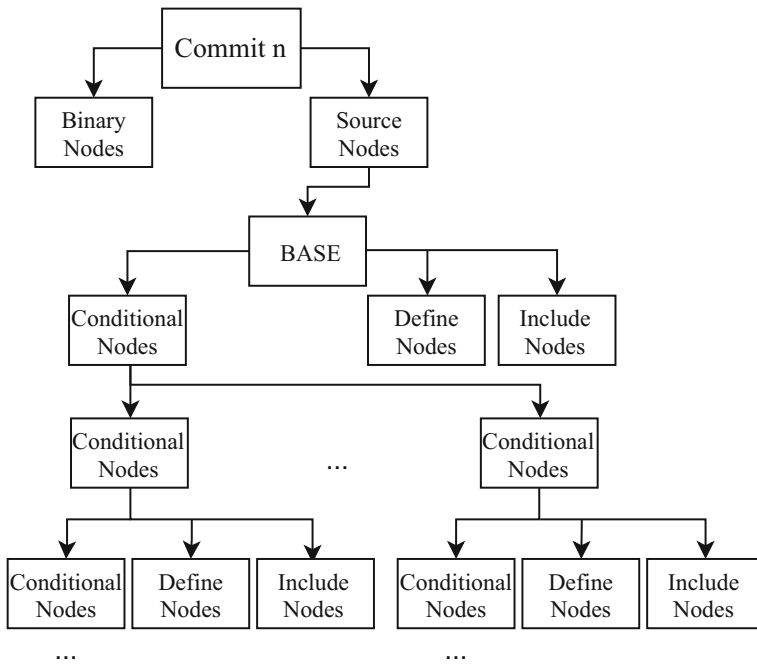


Fig. 7 Structure for computing Git commit differences to analyze changes in annotated blocks of code

configuration for the preprocessor that activates the desired block of code, we obtain an assignment for all the annotated feature literals that are part of the condition of the block. We then create a set of constraints that are handed over to a solver. The constraints we build consist of three parts, which will be explained using the example in Fig. 6. Firstly, we retrieve the local condition, i.e., the condition of the closest conditional block to the changed code. As mentioned before, point T1 is the code of the file *main.c* before the change in Line 12, and point in time T2 is when the change was performed in the code of Line 12 of the file *main.c* (Fig. 6). Thus, the logic formula of the local condition in the example at point T2 is: $2 + 9 > Z$. The second part is the global condition of the desired block, which is a conjunction of all parent conditions, i.e., all conditional blocks wrapping the closest conditional block. We obtain it by walking up the tree, starting from the changed node, which in our example results in a global condition with logic formula: $Y \wedge (2 + 9 > Z)$.

The feature implications make the third part used to create and apply a mapping of all internal feature literals to just external feature literals. We thus traverse the tree to build the feature implications. For example, in Fig. 6, we can be seen that A defines $B=2$ (Line 3, *main.c*) and $C=9$ (Line 4, *main.c*), and BASE defines $Z=3$ as there is no conditional block wrapping Line 1 in the file *header.h*. Thus, BASE implies *header.h* and the features that activate the code block that changed (Line 12) are X, B, C, and Z, which are defined by features A and BASE. Still, when walking up the file we see that there is an outermost code block with a condition expression involving the feature Y (Lines 9-14), which wraps the changed block. The feature implications are mathematically defined as follows: $(A \implies (B = 2)) \wedge (A \implies (C = 9)) \wedge (BASE \implies (Z = 3))$. The conjunction of all these parts, local and global condition and implications, are the logic expression to the problem constraint that can be handed to the solver: $(A \implies (B = 2)) \wedge (A \implies (C =$

$9)) \wedge (BASE \implies (Z = 3)) \wedge Y \wedge (2 + 9 > Z)$. The solution assigned that satisfies this formula is then: $BASE = TRUE \wedge Y = TRUE \wedge A = TRUE$. We thus know that these features must be selected to include the changed block of code in a variant.

If the solver finds no solution, the part of code we want to activate is dead as no configuration can activate it. If a solution can be found, we validate that all feature literals with assignments are external. If the set of assignments are not empty at this point, we obtain a configuration for mining a variant. Before using these variants as ground truth for evaluating *ECSEST*, it was essential to know what features should be marked as changed for the respective changed node and thus be treated as a feature revision. We assumed the features annotated closest to a change as the ones that caused it. Therefore, we got a solution using only the local condition without any implications. In cases where a local condition contains more than one feature to activate a particular changed block of code, nothing affects the ground truth generator approach because the constraint is built considering all the features of the conditional block. Then, the CSP solution is retrieved according to the constraints and can assign the change to more than one feature. Therefore, depending on the feature interactions in more complex conditional expressions comprising several features, it might happen that a changed block of code is assigned to more than one feature revision.

In case the solution did not retrieve any potentially changed feature, meaning that there were no positive external features in the closest condition, we repeated the same process with the parent conditions until we find a positive external feature from the solution. In the worst case, we reached the node corresponding to *BASE*, which is trivially a positive solution.

Generating ground truth variants After these previous steps, we generated the ground truth variants by partially preprocessing the code. Finally, the solution found by the Choco solver for the configuration was used to retrieve the variant, which could be used as input for locating feature revisions. Figure 5 illustrates the variants mined with a set of feature revisions from the changes in time T1 and T2.

5.5 Metrics

We present and discuss the metrics we used for the evaluation of our approach. We first computed metrics characterizing system evolution in space and time in real systems to show the need of such approach at the level of feature revisions. Thus, we computed feature revision characteristics showing the feature evolution over time, related to their source code artifacts. We continued by computing the metrics for evaluating the *ECSEST* approach. Furthermore, we computed metrics to evaluate *ECSEST* for composing new variants with a new combination of feature revisions. Additionally, we also measured runtime metrics to evaluate the performance for locating feature revisions and composing variants.

The *Feature Evolution Metrics* are computed to show the number of new features introduced and the number of features that were changed over the life cycle of a system. Thus, they indicate the feature evolution of the ground truth variants used in our experiments.

- *FeaturesIntroduced*. Number of new features introduced over the Git commits analyzed.
- *FeaturesChanged*. Number of features changed over the Git commits analyzed.

The *Feature Revision Metrics* are computed to characterize the differences of the source code of different revisions of a feature in terms of AST nodes. These metrics represent the variability existing in the ground truth variants used to evaluate our approach. We thus count

the number of AST nodes used to represent the feature revisions artifacts by our adapter for C language artifacts. Each of the following metrics counts the number of a specific AST node within the source code of a feature revision.

- *Header*. Number of *header files*.
- *Define*. Number of *defines*.
- *Field*. Number of *field/struct declarations*.
- *Function*. Number of *functions*.
- *If*. Number of *if conditions*.
- *For*. Number of *for loops*.
- *Do*. Number of *do loops*.
- *Switch*. Number of *switch conditions*.
- *Case*. Number of *case statements*.
- *While*. Number of *while loops*.
- *Problem*. Number of *problem blocks* not recognized in the C AST.

Feature Revision Location Metrics Precision, recall, and F1-score measure how well information is retrieved by a system in relation to the relevant information (Ting 2010). They are commonly used to evaluate feature location techniques (Cruz et al. 2019; Martinez et al. 2018; Michelon et al. 2019). In order to assess the effectiveness of *ECSEST* to correctly locate and not miss any relevant artifacts, we analyzed if the stored traces allow retrieving the artifacts belonging to a specific feature revision. We applied the aforementioned metrics by comparing artifacts of feature revisions composed by the traces of *ECSEST* with the artifacts of the ground truth (see Section 5.4). We used two levels of granularity, due to the feature evolution analyzed, and the different kinds of files that existed in the subject systems (C, binary and text files): *file-level* comparison of two complete files by matching their content; *line-level* comparison of two code files. As the C files from the input variants used for the feature revision location consist of source code after resolving preprocessor directives, the composed variants also contain the C source code files with preprocessor directives resolved. Thus, the comparison is performed on the C source code files after resolving preprocessor directives.

Precision of the file-level comparison is the percentage of correctly composed files, i.e., retrieved files with entire content matching the relevant ones. Recall measures the total percentage of entire matching of all composed files relative to all relevant files. Regarding line-level comparison, precision is the percentage of correctly retrieved lines, while recall is the percentage of matched lines retrieved relative to the total of relevant lines.

- *PrecisionFileLevel*. The percentage of correctly retrieved files in relation to the total retrieved.
- *RecallFileLevel*. The percentage of correctly retrieved files in relation to the total ground truth ones.
- *F1ScoreFileLevel*. The percentage of the weighted average of Precision and Recall at the file level.
- *PrecisionLineLevel*. The percentage of correctly retrieved lines in relation to the total retrieved.
- *RecallLineLevel*. The percentage of correctly retrieved lines in relation to the total ground truth ones.

- *F1ScoreLineLevel*. The percentage of the weighted average of Precision and Recall at the line level.

Hint Metrics. To estimate the usefulness of the hints to complete new variants we used the *ArtifactsRatio* indicating for how many new variants with hints it might be necessary to add and/or remove artifacts. Measuring the *InteractionsRatio* shows the ratio of variants with hints that say there is no trace with this new combination. This can help to analyze feature interactions as two specific feature revisions that never appeared together often cannot co-exist in the same configuration. The *ArtifactsRatio* is used to present how helpful our hints can be for showing possible feature interactions when composing a product with a new combination of feature revisions never used before. Thus, we evaluate if hints with surplus/missing artifacts are the result of possible feature interactions or an invalid configuration. Therefore, the correctness of the approach for composing variants is measured by precision and recall from comparing artifacts of a composed variant with the corresponding ground truth variant.

- *ArtifactsRatio*. The percentage of the number of new variants composed with hints that have artifacts missing/surplus in relation to the total of new variants with hints.
- *InteractionsRatio*. The percentage of the number of new variants composed with hints that have feature interactions and retrieved missing/surplus artifacts in relation to the total number of new variants.

As mentioned in Section 4, the invalid configurations used to compose a variant can retrieve invalid variants due to feature interactions. However, our approach is designed to trace artifacts to feature revisions and use them to compose variants. Thus, our evaluation aims to quantify the correctness of the traces computed and the feasibility of using our approach for composing new variants or new product revisions with feature revisions containing updated implementation. In this way, if artifacts are retrieved correctly, valid configurations will result in valid variants. Our evaluation does not focus on analyzing if valid configurations are created but on whether our approach can correctly locate feature revisions and compose variants given a configuration with feature revisions. Despite already providing some hints, we need to improve our approach to help users to compose valid and consistent configurations with the evolution over time. We plan to improve our approach in future work to analyze the evolution of dependencies and interactions in the source code of feature revisions, as shown by Feichtinger et al. (2021).

Performance Metrics To run the experiments we used a machine with an Intel®Core™ i7-6700U processor (3.4GHz, 4 cores), 32GB of RAM, SSD storage, and the Windows 10 operating system. Thus, under this capacity circumstances we measured the approach performance:

- *ExtractionTime*. The time in seconds for locating feature revisions, i.e., for extraction of mappings of feature revisions to artifacts from a variant.
- *CompositionTime*. The time in seconds for composing a new variant, i.e., the time needed to retrieve traces from a set of feature revisions and compose their artifacts in order to generate a variant.

Regarding the performance metrics, we are interested only in evaluating *ECSEST* and not the process of mining the ground truth. The mining process was just necessary to create

the ground truth and input variants for the evaluation. We thus evaluate our approach supporting the evolution of annotation-based SPLs in VCSs by locating feature revisions and composing variants.

5.6 Implementation Aspects

We implemented *ECSEST* on top of the VarCS ECCO² and performed some optimizations to implement the concepts of our approach presented in Section 4.

Feature Interaction Limit. We limited the maximum size of clauses in presence conditions, i.e., the number of feature literals in a conjunction, which corresponds to the number of interacting features, to a threshold based on previous empirical research (Fischer et al. 2016; Fischer et al. 2014). This provides a major improvement to the scalability of the approach, as otherwise the number of clauses would grow exponentially with the number of features.

The threshold can be freely configured, however, for the evaluation presented in this paper it was set up to three interacting features, which strikes a reasonable balance between computational effort and quality of results (Fischer et al. 2014, 2016). Considering higher-order feature interactions would yield only very little additional gain while significantly increasing cost, similar to t-wise interaction testing of product lines. Using a threshold of feature interactions limits higher orders of feature interactions in a clause in the set of clauses of a trace.

Artifact Sequence Alignment The artifact equivalence is performed by an adaptation of the Longest Common Subsequence (LCS) algorithm (Deorowicz et al. 2014) to perform multi-sequence alignment for comparing more than two variants (Fischer et al. 2014; Linsbauer et al. 2017a), e.g., if they have the same method whose statements must be aligned.

Artifact Adapters We keep the approach independent of the types of implementation artifacts by utilizing artifact type specific adapters that are responsible for parsing respective files and generating the generic artifact tree structure consisting of folders, files, and further file type-specific artifacts. The only requirement is that artifacts can be uniquely identified and compared for equivalence. In this work, we used the Eclipse CDT²³, i.e., a C/C++ Development Tooling for implementing the adapter for parsing our target systems artifacts.

The approach itself is implemented with the Java programming language and the data storage and manipulation depend on the JVM memory. The more nodes have to be created to store uncommon artifacts, the higher is the memory consumption. This is why we use fine-grained parsing to store the artifacts. With this new plug-in, compared to our previous work (Michelon et al. 2020d), we can locate more feature revisions because of the reuse of the tree structure nodes for storage and manipulation of traces and feature revisions in the repository.

²³<https://www.eclipse.org/cdt/>

6 Results and Discussion

This section discusses the results of our empirical analysis of the feature evolution in space and time from the six subject systems. Based on the results and analysis, we provide answers for the five posed RQs.

6.1 Feature evolution in space and time

Figure 8 summarizes the evolution in space and time, i.e., the number of features introduced and changed over the range of Git commits for each of the six systems. The blue line represents the evolution in space, i.e., the number of new features introduced, while the red line represents the evolution in time, i.e., the number of revisions of already existing features. First, regarding evolution in space, Fig. 8a shows the feature evolution of the Marlin system. After the product started with Git commit #1 with just feature `BASE`, the second commit introduced 16 new features. Then, later in Git commit 51, 109 new features were introduced. Furthermore, additional new features were included in four Git commits.

For the LibSSH system, shown in Fig. 8b, the initial version started in Git commit #1 with 13 features. Then, there were ten changes affecting the evolution in space in the 400+ commits analyzed resulting in a total of 39 new features.

In case of SQLite, shown in Fig. 8c, after the first Git commit introducing only the feature `BASE`, four features were added in the second commit. Along the commits analyzed, within 11 commits 33 new features introduced. Regarding the evolution over time, there were 29 Git commits with feature revisions.

In the Irssi system (Fig. 8d), six features were added in Git commit #2, eight in Git commit #32 and 10 features in Git commit #162. In three other commits evolving the system in space, only one feature was introduced. Over time, usually, one feature changed, with exceptions in eight Git commits, ranging from two up to five features introduced in the same commit.

Regarding Bison, shown in Fig. 8(e), features were introduced in 14 Git commits. The evolution over time resulted in 270 feature revisions over the 241 commits analyzed, ranging from one to four revisions per commit.

In the Curl system (Fig. 8f), 46 features were introduced in the first Git commit of the project. The next evolution in space happened in 10 Git commits. The evolution over the 350 Git commits resulted in 422 feature revisions, with the highest number of revisions (15) in a single commit happening in Git commit #189.

From the analysis of system evolution over time of these six systems, we observed that many features change over time besides the feature that represents the core of the system, i.e., the feature `BASE`. For Marlin, 22 different features changed in the Git commits analyzed. In the LibSSH and Curl systems, 30 features evolved over their Git commits analyzed. For the SQLite system, 24 features changed, and for the commits analyzed in the Irssi and Bison systems, 12 and 13 different features changed, respectively.

The evolution over time by feature revision can strongly impact product configurations of configurable software systems. For example, LibSSH had six features changed and four introduced in Git commit #38. This evolution in space and time in a single commit makes engineering tasks complex. Suppose an engineer needs to recover an older version of a specific feature introduced before commit #38, keeping the change of other features. This would require great effort and would be error-prone since other current variants of the LibSSH system could be still using the newer version of that feature. Considering these six subject systems with different domains, we can see that features have been introduced and

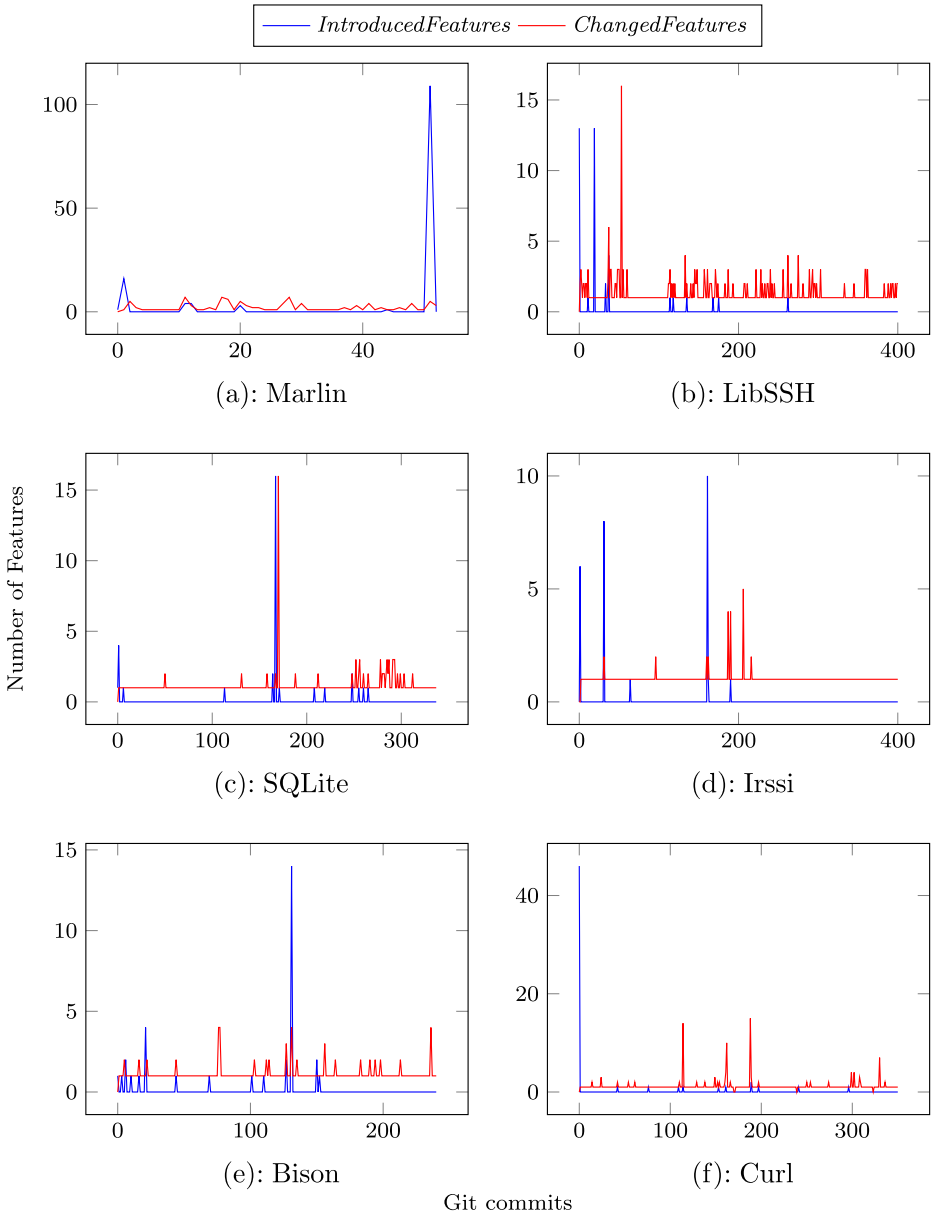


Fig. 8 Relation of the number of features introduced and changed over the Git commits analyzed for each system

changed frequently during their development, which would benefit from a mechanism to handle feature revisions, such as the *ECSEST* approach.

Regarding feature evolution, we considered not only functional features because according to Berger et al. (2015), in the industrial systems features are also needed for testing,

debugging, build, optimization, deployment, simulation, or monitoring. These atypical features can be introduced for the optimization of non-functional aspects. Therefore, features such as `YYDEBUG` from the Curl system (shown in Fig. 9f), indirectly realize customer requirements. An interesting example of a feature revision that have to be reused in previous revisions of the SQLite system can be seen in the feature `SQLITE.TEST`²⁴, which evolved meaning that its change had to be applied in four releases of the system: *branch-3.9*, *branch-3.18*, *branch-3.19* and *branch-3.22*.

In Fig. 9, we show the evolution of the C artifacts in the AST of the feature revisions that most often changed over the Git commits analyzed for each target system without considering the `BASE` feature. The number of AST nodes of the feature `HAVE_SSH1` from the LibSSH system (Fig. 9b) has constant changes, but the number of fields increased significantly in its second revision. This is also the case for the feature `SQLITE_TEST` from SQLite and the feature `MSDOS` from the Bison system in revision 6 (Fig. 9c and e), and for the feature `ADVANCE` from Marlin in revision 4 (Fig. 9a). The evolution over time of the feature `SQLITE_TEST` from the SQLite system (Fig. 9c) shows an increasing number of AST nodes up to its sixth revision. After that, the AST nodes remain constant in terms of numbers per node type, and in case of field nodes, the number decreases in the twelfth revision of the feature `SQLITE_TEST`. The feature `_GNUC_` from the Irssi system (Fig. 9d) has not been changed regarding the number of AST nodes over the three revisions, with the exception of the number of problem statements/blocks and defines AST nodes, which increased during the second revision. For example, for the feature `YYDEBUG` from the Curl system (Fig. 9f) the number of header files, define, field, if, and for AST nodes increased in its fourth revision. In the fifth revision, for example, 14 header files were removed from its implementation.

Knowing which commits have new feature revisions can make it easier for developers to find a specific revision of a feature. The chart on Fig. 9a, for example, shows that the fourth revision of `ADVANCE` from the Marlin system substantially changed compared to its predecessor. We analyzed the Git commit `094afe7`²⁵ and saw from the commit message that a merge was performed. A developer added 12 new files, changed eight, and removed two files that affected the source code of the feature `ADVANCE`, and hence, the behavior of how the movement of the printer is done with linear acceleration. For every new revision of this feature, the movement is affected. In the ninth revision (Git commit `65934ee`²⁶) many changes were performed in the planner source code, which influences the buffers movement commands and manages the acceleration profile plan.

We now use the feature `YYDEBUG` from the Curl system as an example. The revisions of this feature are from five different releases of the system. This feature is used for debugging purposes and contains many `fprintf` calls to print the debug messages in a file. Thus, depending on the revision selected, different debug messages are printed. If developers want to use the revision of this feature to get more debug messages and combine it with features of another release, it can be supported with `ECSEST` instead of manually retrieving the Git commits of the feature revision and release. Besides that, developers will need to work manually copy, paste and modify the code of the release with the code of the desired feature revision.

²⁴<https://www.sqlite.org/src/info/7b4583f932ff0933>

²⁵<https://github.com/Marlin/commit/094afe7c1065d5663628b389f27687a5f465abb8>

²⁶<https://github.com/Marlin/commit/65934ee9c6ae792c708bc1cea9996c8a5df67f5>

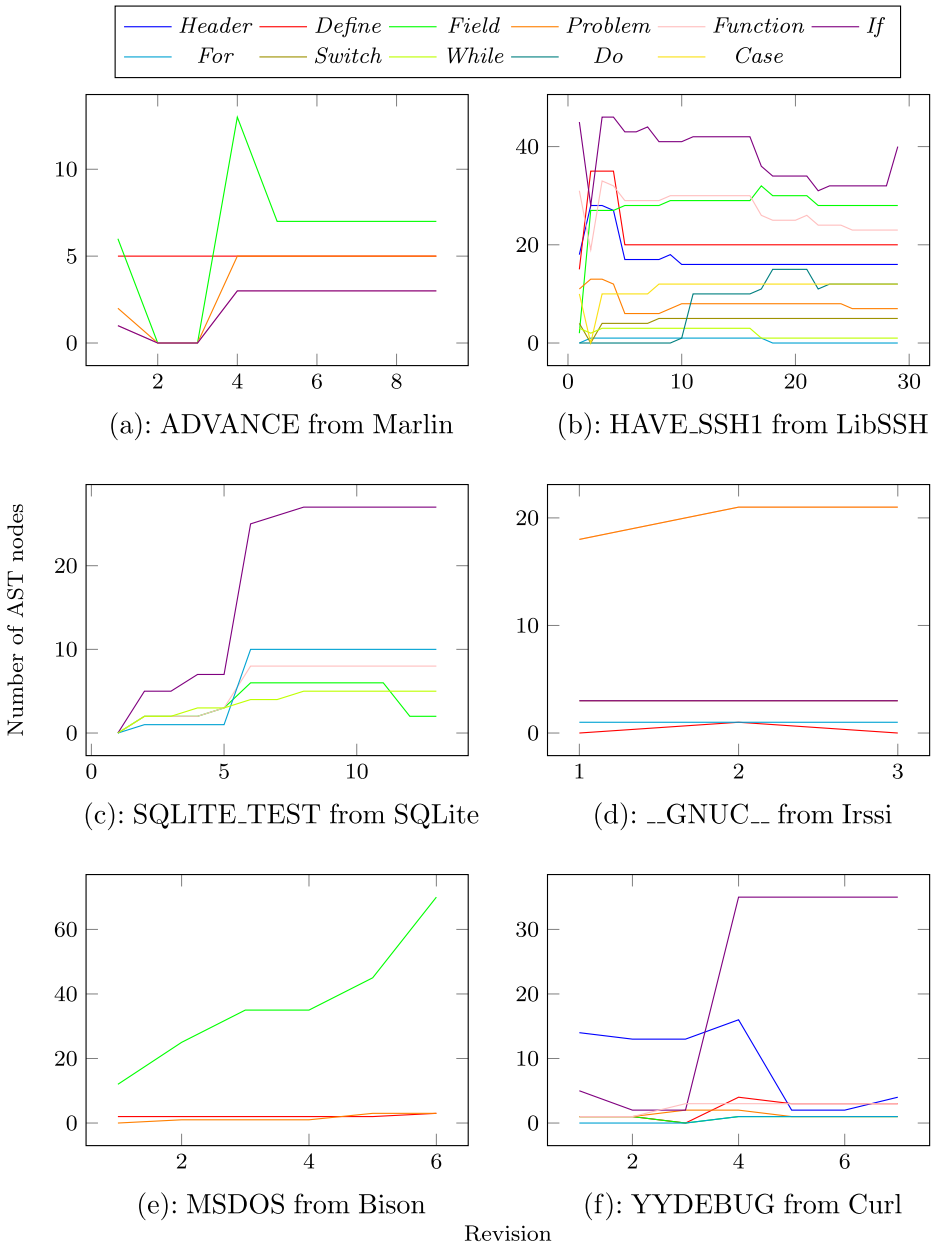


Fig. 9 Evolution over time of the number of AST nodes from feature revisions of each system

With the analysis of feature revisions and their number of AST nodes over time in Fig. 8, we can see that feature evolution happens over time, because the source code changes cover more than single lines and also affect header files, defines, fields, conditions, and loops. Thus, if a developer wants to use, for example, an older revision of a specific feature with the previous revisions of other features, ECSEST eases the process of combining features

with different revisions to obtain their different source code, thus producing different system behaviors.

RQ1. To what extent do features evolve in space and time? *We could see in the mined ground truth variants that multiple features are substantially changed and introduced in single commits, showing the need for our approach support. By analyzing the AST nodes of the feature revisions with ECSEST, we could also see that the results of the feature evolution are meaningful.*

6.2 Locating feature revisions

The results of *ECSEST* for locating feature revisions are shown in Table 5. The precision for the six subject systems was 100% at file level and 99% at line level, except for the Bison system, which was 100% at line level too. The recall values ranged from 92% up to 99% at file level and were 99% at line level for all systems. The values of F1, which consider both precision and recall, are between 96% and 99% at file level and 99% at line level, showing that *ECSEST* reliably locates feature revisions by a given set of variants in different space configurations and in many points in time. Overall, when computing the average of all systems shows that precision and recall stay above 97% at file level and 99% at line level.

Although in this work we developed an adapter more fine-grained for the specific syntax of the C programming language, there are still issues: some deleted lines are shown in the example from a code snippet in Listing 6, with comments after a source code statement. Our adapter was not developed to capture this kind of comment split into multiple lines. It thus ignores Lines 2 and 3, which are false negatives in the composed variant when comparing it with the input variant. False positives are due to some lines that are split into multiple lines by our adapter when reading the source code. For example, Listing 7 shows a *Do Block* followed by the *If Block* and followed by the *While Block* at the same line. The parser gets the statement and adds the *Do Block* in a new line, the *If Block* in another line, and the

Table 5 Average precision, recall and F1-score metrics of composed artifacts per system

Subject system	Granularity	Precision	Recall	F1 – Score
Marlin	<i>FileLevel</i>	1.00	0.95	0.98
	<i>LineLevel</i>	0.99	0.99	0.99
LibSSH	<i>FileLevel</i>	1.00	0.99	0.99
	<i>LineLevel</i>	0.99	0.99	0.99
SQLite	<i>FileLevel</i>	1.00	0.97	0.98
	<i>LineLevel</i>	0.99	0.99	0.99
Bison	<i>FileLevel</i>	1.00	0.99	0.99
	<i>LineLevel</i>	1.00	0.99	0.99
Curl	<i>FileLevel</i>	1.00	0.92	0.96
	<i>LineLevel</i>	0.99	0.99	0.99
Irssi	<i>FileLevel</i>	1.00	0.99	0.99
	<i>LineLevel</i>	0.99	0.99	0.99
All	<i>FileLevel</i>	1.00	0.97	0.99
	<i>LineLevel</i>	0.99	0.99	0.99

```

1 str = buf;          /* Default buffer to use when we write the
2                    buffer, it may be changed in the flow below
3                    before the actual storing is done. */

```

Listing 6 Code snippet from Curl, file download.c

While Block in a third line due to our tree structure for parsing the artifacts in specific types of AST nodes (Fig. 2). Therefore, the source code retrieved is correct but retrieved in more lines. Therefore, in the end, if we look for the total amount of lines of all variants of the systems we could easily get a higher number of lines in these specific cases as explained in the Listings 6 and 7.

There were only 350 false negative lines and about 300 false positive lines in the Marlin system from a total of 692,001 relevant lines across all 191 compared variants. In the LibSSH system, 39 lines were missing and 151 were surplus over all 577 composed variants of a total of 8,191,428 relevant lines. In the SQLite system, 358 lines were false positives and 152 were false negative lines in all 424 composed variants from a total of 4,187,636 relevant lines. In the Irssi system, 725 were inserted lines and 789 were missing lines from a total of 7,549,177 relevant lines. In the Bison system, there were no inserted lines and only three missing lines from 1,799,181 relevant lines. In the Curl system, there were 241 inserted lines and 5,163 deleted lines from a total of 4,556,535 relevant lines. Despite not having 100% of precision and recall, as explained before, the few false positives and false negatives resulted from comment lines ignored when parsing the C source code files or different alignments, which did not change the code semantically. Furthermore, compared to a traditional VCS, evolution is tracked at the level AST nodes of feature revisions, not at the level of text lines or entire files.

RQ2. To what extent does ECSEST support feature revision location of existing variants that evolved in space and time? *ECSEST proved to be effective for locating feature revisions in the used dataset, with high values for the measures of precision, recall, and F1-score. The approach correctly located the artifacts in an automated way, hence, it may support developers to locate feature revisions and understand their implementation in systems evolving in space and time, even in cases with many feature revisions.*

6.3 Composing variants with new configurations of existing feature revisions

Table 6 shows the precision, recall, and F1-score from the comparison of artifacts (file and line levels) of the ground truth and our composed variants according to the random configurations generated for the Git commits analyzed. *ECSEST* retrieves artifacts with 100% precision and 92%-99% recall at file-level granularity. At line-level granularity, the average precision and recall are 99% for all systems, with an exception for the recall of the system Bison that is 100%. All values for F1 are greater than 96% at file level, and as well as the F1 achieved at line level with the F1 achieved from the input variants, all systems have 99% F1 from random configurations.

```

1 do { if (...) { ... } while (0);

```

Listing 7 Code snippet from LibSSH, file client.c

Table 6 Average *Precision*, *Recall* and *F1 – Score* metrics of composed artifacts for random configurations per system at *FileLevel* and *LineLevel*

Subject system	Granularity	<i>Precision</i>	<i>Recall</i>	<i>F1 – Score</i>
Marlin	<i>FileLevel</i>	1.00	0.96	0.98
	<i>LineLevel</i>	0.99	0.99	0.99
LibSSH	<i>FileLevel</i>	1.00	0.99	0.99
	<i>LineLevel</i>	0.99	0.99	0.99
SQLite	<i>FileLevel</i>	1.00	0.99	0.99
	<i>LineLevel</i>	0.99	0.99	0.99
Bison	<i>FileLevel</i>	1.00	0.99	0.99
	<i>LineLevel</i>	0.99	1.00	0.99
Curl	<i>FileLevel</i>	1.00	0.92	0.96
	<i>LineLevel</i>	0.99	0.99	0.99
Irssi	<i>FileLevel</i>	1.00	0.99	0.99
	<i>LineLevel</i>	0.99	0.99	0.99
All	<i>FileLevel</i>	1.00	0.97	0.99
	<i>LineLevel</i>	0.99	0.99	0.99

For the Marlin system, within a total of 133,161 relevant lines, 64 were inserted and 67 were missing lines. For the LibSSH system, 1002 were inserted lines and 97 lines were deleted lines, from a total of 5,433,889 relevant lines. For the SQLite system, zero were false negative lines and 369 were false positive lines in the composed variants with random configurations from a total of 3,375,242 relevant lines of ground truth random variants. The false positive lines in the composed variants are caused by feature interactions in the chosen configurations, which we randomly chose without considering whether a selected feature excludes parts of code that can be in other features when preprocessing ground truth variants. Therefore, when the random combination of feature revisions resulted in an invalid configuration, the ground truth variant cannot be correctly preprocessed, and thus, has missing artifacts. An example of an invalid random configuration generated in our evaluation is the random variant generated in Git commit #12 of LibSSH, which contains the features `HAVE_SSH1`, `DEBUG_CRYPT0`, `HAVE_PTY_H` and `BASE`.

Listing 8 shows that when preprocessing a variant with feature `HAVE_SSH1` defined, the ground truth variant will contain Line 2 and not Line 4. Only when this feature is not defined Line 4 will be present in the variant. Our feature revision location approach correctly mapped the artifact from Line 2 to presence conditions containing feature `HAVE_SSH1` and Line 4 to presence conditions containing `BASE` and other features from the respective point in time. However, the ground truth variant does not contain artifacts of both `#ifdefs` and `#else` blocks, hence, not matching with the composed variant. Curl random variants were composed with 208 inserted lines and 3,787 deleted lines among a total of 3,266,773 relevant lines. The randomly composed variants from the Bison system retrieved 54 inserted lines and no deleted lines from a total of 1,412,709 relevant lines. From a total of 6,972,575 relevant lines in the Irssi system, 786 lines were inserted in the randomly composed variants and 1,017 lines were deleted.

We did not test the approach's capability for combining feature revisions from different points in time due to limitations of our ground truth generator. However, the efficient feature revision location assures that feature revisions are correctly traced. In addition, our results

```

1 #ifdef HAVE_SSH1
2     option->ssh1allowed=1;
3 #else
4     option->ssh1allowed=0;
5 #endif

```

Listing 8 Code snippet from LibSSH, file options.c

of precision and recall for composing new variants only presented lower values when invalid configurations were used due to feature interactions. We did not evaluate if valid configurations can be retrieved, but if our approach can correctly compose variants with the located feature revisions. Our focus is on supporting the feature evolution of annotation-based SPLs in VCSs as variability models of our target systems were not available.

RQ3. How effective is ECSEST for composing new variants with feature revisions?

The traces computed by ECSEST proved to be useful for creating new variants with random configurations, still achieving high values for the measures of precision, recall, and F1-score. ECSEST can be used to support tasks to compose new variants from an SPL with a set of feature revisions.

As explained above and in Section 4, false negatives and false positives can be retrieved in the variants depending on how features are annotated in the ground truth variants and which feature revisions are combined when composing a new configuration that did not exist so far. However, the false positives and negatives can be identified easier with the hints retrieved by ECSEST when composing a new variant. It can happen that in some composed variants, no artifacts are missing/surplus and the hints file can either have no hints retrieved or can present hints even there are no feature revision interactions. In the last case, the hints are retrieved because some of the feature revisions of the configuration never appeared together in the input variants.

Table 7 shows the *ArtifactsRatio* indicating that the retrieved hints are useful for all systems with exception of the hints retrieved for the Bison system. For Bison, only a few false positives were retrieved from the new variants, which are almost all false positives caused by the AST nodes used to store the source code in a tree structure. Thus, the false positives in the Bison system do not stem from the algorithm for locating feature revisions, and most of them were not retrieved due to feature interactions, as we see when comparing which artifacts were surplus in relation to the ground truth.

Analyzing the *InteractionsRatio*, the hints with missing traces were most useful for finding artifacts surplus/missing for Marlin (90%) but not as useful for SQLite (37%). In SQLite, this means that despite having new configurations with feature revisions never used together previously, most of the new configurations can be combined and might not have feature interactions. For the Bison system, 66% of the hints with missing traces really pointed to new configurations with surplus artifacts, which means some of them have been useful to find that some features should not be used together.

Although hints were obtained for Marlin, SQLite, and Irssi, they do not reflect actual missing or surplus artifacts. The missing or surplus artifacts were retrieved due to the differences found when parsing the C source code. For example, the SQLite system has 77 surplus lines due to feature interactions used in the random configurations from the total false positive lines retrieved. In the Curl system hints have been more useful for finding

Table 7 Hints metrics

Subject system	<i>Artifacts Ratio</i>	<i>Interactions Ratio</i>
Marlin	88%	90%
LibSSH	38%	88%
SQLite	75%	37%
Bison	2%	66%
Curl	100%	100%
Irssi	60%	83%
All	60.5%	77%

surplus/missing artifacts (100% *Artifacts Ratio*) and alerting for missing/surplus artifacts because all new variants have feature interactions due to feature revisions never used together.

RQ4. How useful are the hints suggested by *ECSEST* for completing new variants and finding feature interactions when creating new configurations? The retrieved hints support the completion of a variant by showing clauses of the presence conditions used to compose a configuration that has feature revisions not included in the configuration. Furthermore, it makes aware of possible interactions of feature revisions, when combining feature revisions that were never used together in a configuration.

6.4 Performance of *ECSEST* to locate feature revisions and compose variants

The performance of *ECSEST* to extract features from systems evolving in space and time (*ExtractionTime*) is shown in Fig. 10. The least time a variant took for extraction of feature revisions is the minimum value on the left side of each system box plot. While the highest time for extracting feature revisions of a variant can be seen in the maximum value, excluding outliers, on the right side of each box plot. On average, the analysis took around 83 seconds for Bison, 250 seconds for Curl, 25 seconds for Irssi, 249 seconds for LibSSH, 88 seconds for Marlin, and 212 seconds for SQLite. In the worst case, it took around 15 minutes for the Curl and LibSSH systems, which are the systems with the highest number of variants in relation to the other systems.

As expected, the runtime for locating feature revisions increases with the number of feature revisions and artifacts because the number of artifacts and features is greater to refine the traces. Thus, the time to create new and update traces, increases for every new input variant. For the Marlin system, the outliers (represented in Fig. 10 by circles) were caused by Git commit #52, because of the huge number of features introduced (56) and the necessary refinement of the traces of *BASE* for every input variant. In addition, it takes a long time for every new input variant to extract what is new and update from what is already in the repository. For SQLite the longer extraction time compared to Irssi is probably caused by the huge number of artifacts that had to be compared.

Thus, independently of the number of feature revisions, the size of artifacts can impact the time to refine traces. Another thing that impacts the time to refine traces is the complexity of the tree structure nodes used to store the artifacts in the repository. It is our implementation limitation, and as many more different artifacts from one commit to another in the input variants, many more tree nodes are created to store this information in a tree

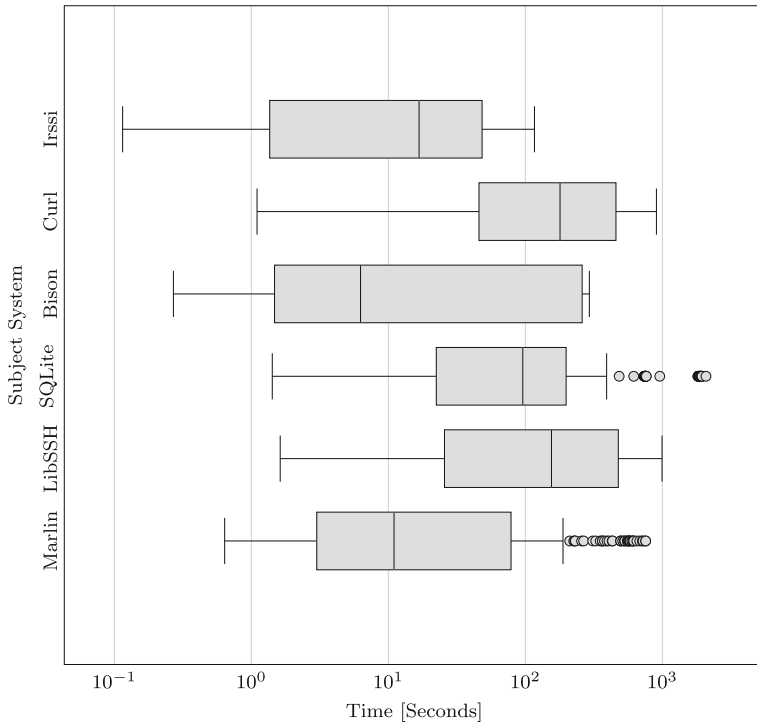


Fig. 10 *ExtractionTime* for feature revision location per variant

structure. In a real scenario, developers may limit the number of commits and variants to extract feature revisions, using only the desired ones for new combination of feature revisions. Further, despite developers may need to wait, using the *ECSEST* approach does not require developers’ time and effort, which they can use in parallel to complete other higher-level tasks and decision making.

The *CompositionTime* for composing a new variant, i.e., joining the artifacts from traces of a set of feature revisions, is presented in Fig. 11. Similar to box plots in Fig. 10, the least time to compose a variant with feature revisions is the minimum value on the left side of each system box plot, while the highest time for composing a variant with feature revisions can be seen in the maximum value, excluding outliers, on the right side of each box plot. For the system with the best runtime performance, it took around two seconds on average per variant. For the system with the worst average, it took around 45 seconds per variant. Some of the outliers (represented in Fig. 11 by circles) in the systems are due to the warm-up effect of the JVM. After the warm-up effect, the time remains constant to compose variants.

For the LibSSH and Curl systems, we had some outliers for which it took up to seven minutes to compose a variant (Fig. 11). When comparing the time between systems, we see that it is higher in repositories containing more traces and feature revisions because many clauses from many traces need to be analyzed to join the artifacts into a variant. However, the number of artifacts is also a factor that influences the composition time. For example, the Bison system has the smallest number of artifacts compared to the other systems (see

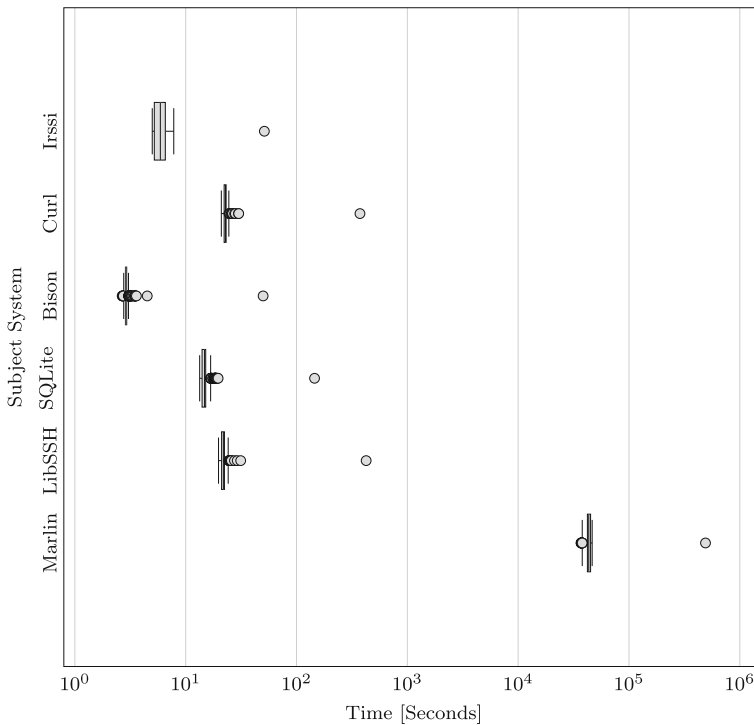


Fig. 11 *CompositionTime* for composing each variant

Table 4) and also the smallest time to compose variants. However, the Curl system is smaller than the Irssi system in terms of artifacts, but took more time to compose variants because it has more feature revisions, hence, more input variants, which results in a higher number of traces. Thus, more time is needed to compose a variant with high numbers of feature revisions and artifacts.

Regarding the composition time, the composition requires that the extraction process was already performed. Despite the extraction time, only new variants with new feature revisions have to be analyzed in our incremental process of locating feature revisions, i.e., the approach uses existing variants and refines existing and creates new traces only when needed. Optimizations of the runtime performance can be performed in the implementation aspects of the approach to store the artifacts and the counter table for mapping feature revisions to artifacts. Still, the composition time in the worst case only took seven minutes, while the approach could save significant effort of manually copying and pasting artifacts for composing variants or for propagating changes of features.

RQ5. What is ECSEST's performance for extracting feature revisions and composing variants? *The automated approach to locate feature revisions and composing variants with ECSEST may save time in recovering feature information and maintenance and evolution tasks. The average time for extraction for our target systems ranged from 25 to 250 seconds per variant, whereas the average time for composing a variant ranged from two to 45 seconds.*

7 Threats to Validity

We discuss the threats to the validity of our evaluation using the taxonomy of Wohlin et al. (2000). We also describe how we mitigated possible threats.

The threats to *construct validity* are related to the study setup. Firstly, the scenarios used to validate our approach contain changes to features, but we did not have data on the actual type of evolution, e.g. performance improvement, new hardware support, bug fixing, etc. However, the Git commit hashes of every variant and revisions of the features are available in our dataset for future replications and deeper analysis. Secondly, the methodology chosen to evaluate our approach was based on variants in space and time created by a configuration of features in specific changes of annotated code in the Git commits we analyzed. This was necessary since there is no ground truth available with variants containing feature revisions. To mitigate this threat and to demonstrate the efficiency of our approach, we generated new variants with different configurations of feature revisions, which were not used as input and randomly chosen for the points in time we analyzed.

Another constructive threat can be related to the correctness of our mining ground truth approach. To mitigate this threat, we manually inspected the ground truth variants generated for the first 50 Git commits of each target system. Yet, regarding the sufficient variability of the ground truth variants, we used variants from Git commits containing introduced and changed features. Furthermore, we also presented some results of how representative the feature evolution of the mined ground truth variants is for the real systems we used.

A further threat to construct validity is the new combinations of feature revisions, as we do not ensure that they are type-safe. However, as mentioned, our approach is intended for tracing feature revisions to artifacts and using the revisions to compose variants. Thus it is the user's responsibility to select valid configurations. Our aim in this work is to analyze if the mapping between feature revisions and artifacts is performed correctly, and if the variant composition approach works. Hence any valid configuration will be correctly composed when every feature revision is correctly traced and the variant composition results in the expected artifacts. Furthermore, work by Feichtinger et al. (Feichtinger et al. 2021), presents an approach to inform engineers about possible inconsistencies between code-level dependencies to feature models. Thus, we can improve our approach in future work by using a similar analysis to the variant composition.

A threat to *internal validity* is the limitation of the underlying tools that could have affected our results. We implemented our approach in ECCO, in which source code is available and was used in previous works (Fischer et al. 2014; 2015, 2019, Linsbauer et al. 2015; Michelon et al. 2019) that shows its efficiency to extract features and compose variants. We also used our developed adapter to parse the C source code and to write it back when composing variants. Although we did not compile the resulting code of the composed variants, we validated the correct composition of artifacts with smaller examples and subsets of the dataset. Furthermore, our implementation is also available for further comparison and to reduce possible bias in our results.

A threat to *external validity* is related to our findings be generalized beyond the cases we considered. Despite we conducted our evaluation with only six systems, these systems are from different domains and have different sizes with different behaviors in terms of how often their features change along the Git commits. Furthermore, the range of Git commits analyzed for each system varies, which makes our analysis valid for systems with a higher number of feature revisions. To mitigate bias in relation to the number of Git commits we used for each system, we performed a triage specifically for each system, allowing us

to define how far we could go with the memory limitations of our machine used to run the experiments. We thus believe that our results cover diverse enough scenarios and our approach can support a large number of feature revisions.

From the perspective of *conclusion validity*, a threat can be related to the metrics we used to evaluate our approach effectiveness. However, precision, recall, and F1-score are efficient to measure how correct is the information retrieved (Ting 2010). Furthermore, they are commonly used to evaluate feature location techniques (Cruz et al. 2019; Martinez et al. 2018; Michelon et al. 2019, 2021b), and hence, can make easier the comparison of our results.

8 Related Work

Many existing solutions for evolution in space consider only variability in space, such as SPL, and require integration with a VCS to manage the evolution of software systems in time (Berger et al. 2019). Thus, there is a lack of dedicated and mature tools supporting system evolution in both dimensions for tracing and developing features over time (Ananieva et al. 2020). Considering both dimensions, Ananieva et al. (2020) presented a conceptual model, which proposes a unified terminology for tools managing variability in space and time. The conceptual model aims at clarifying communication between researchers and developers for understanding and comparing existing tools, and for preventing duplicated tool development.

Regarding existing tools for software system evolution in space and time, ECCO was presented by Fischer et al. (2014) as an extraction and composition tool for re-engineering cloned system variants into SPL. After mapping all existing features to artifacts, developers can select the desired set of features to compose a new product variant and also provide hints for manual completion of which software artifacts would need adaptation. Then, ECCO was built upon the checkout/commit workflow for distributed software development (Linsbauer et al. 2016). It evolved to extract variability information from system variants computing traces of not only single features but also feature interactions and absence of features, non-unique traces, and dependencies between traces (Linsbauer et al. 2017b). ECCO has also been used in a large-scale industrial case study (Hinterreiter et al. 2020). In this work, we present a significant extension to the tool, with a feature revision location (Michelon et al. 2020d) and composition of variants to support system evolution and comprehension of feature evolution.

Superimposition of Models (SuperMod) (Schwägerl and Westfechtel 2016) is a tool for evolving model and implementation of SPLs in a conceptual framework for integrating revision and variation control of model-driven software projects (Schwägerl and Westfechtel 2019). Similar to VCSs, SuperMod is based on a workspace repository, where the user can store and edit the files of source code and modeling of all revisions and variants of an SPL. Thus, the workspace is populated with the feature model and the model artifacts belonging to a revision of the SPL. The tool allows artifacts and feature models to co-evolve. The SPL is evolved in iterations via commit/checkout operations, similarly to VCSs. SuperMod allows collaborative development, where each user works with a local repository and can copy and update the central remote repository. One of the tool limitations is that there is no possibility of working with different artifacts than model and text files. Furthermore, there are some adoption barriers to migrating the SPLs from external tools.

DeltaEcore (Seidl et al. 2014) is a tool for capturing variability in space by automatically defining delta languages for a variety of languages relevant for SPLs and software ecosystems (SECOs). These delta languages can be textual, graphical, or use any other representation, and are required for software systems that consist of multiple artifacts, such as design models, source code, configuration files, or documentation. Thus, the tool is able to derive syntax and semantics for custom delta languages from a specific source language's meta-model. Furthermore, the tool supports editing, parsing, and interpreting a generated delta language, which can be integrated into a mechanism for composing variants of an SPL or SECO. Despite the tool supports variability model based on a Hyper Feature Model (HFM) (Seidl et al. 2013), which depicts the dependencies and incompatibilities of features version ranges of SPLs, it is limited to the evolution of system families in time, because it does not support to evolve artifacts and/or features.

A recent survey by Linsbauer et al. (2021) describes existing VarCSs and their essential differences, the challenges, and insights for the future generation of tools to support the development of system families evolving in space and time. Among the challenges mentioned, the externalization expression of which functionalities (variable artifacts) are part of a variant can become cognitively complex to handle because a high number of revisions and variants can exist and a million thousand features. For instance, the Linux kernel has 15,000+ features (configuration options), which some of them can have 3 values: "yes", "no", or "module", with an estimated number of $3^{15,000}$ possible variants of Linux (Pereira et al. 2020). It is still unclear when and for what developers would have advantages by using a VarCS in such a context. Thus, it is important to conduct studies on what characteristics a VarCS should have to help to deal with systems evolving in space and time. This includes the types of artifacts VarCSs should allow to create and manipulate, the kind of operations they should support, and features ensuring usability to deal with the cognitive complexity involved. In this direction, studies are needed to investigate the ECCO VarCS capabilities in more detail. For example, evaluating if its characteristics and operations are useful and can be performed efficiently to support the evolution of system families. Our study shows ECCO's current utility and suggests for improvements, which can serve as a basis for new studies and the development of tools for software system variability and evolution.

An approach to support the composition of new variants based on opportunistic reuse, namely clone-and-own methodology, is presented by Ghabach et al. (2018). It supports mappings between features and artifacts in an automated and incremental way. The paper also discusses possible scenarios, constraints and cost estimation for operations to compose new variants using clone-and-own. The scenarios are given by three hints: (i) clone and retain, when developers clone an artifact and can retain it as it is, without modifying its implementation; (ii) clone and remove, when developers may clone an artifact instance, and have to remove from it the implementation artifacts that are not required by the configuration; and (iii) extract and add, when developers extract from product artifact implementation of feature required by the configuration and add it to a cloned new variant under composition. The mapping of features and composition hints and cost estimation is defined by means of correlations, indicating the coexistence of a feature and an artifact, or a feature and an artifact. Thus, this approach, similar to ours for locating features from existing system variants, is independent of the artifacts types. However, *ECSEST* is able to locate features at different points in time. Regarding the composition of new variants, *ECSEST* can also use existing cloned variants to compose new products. In addition, *ECSEST* can easily retrieve the variant in an automated way by informing the set of feature revisions desired in the configuration. Our hints can help developers to determine if the variants generated in an

automated way have possible remaining or missing artifacts. The results from our feature revision location technique (Michelon et al. 2020d) show that our approach maps features to their artifacts with high precision and recall, which means that less effort is needed to tailor a variant, as fewer removals and additions are necessary when composing variants with new configurations.

But4Reuse (Martinez 2016) is an approach for migrating software variants into an SPL by constructing its feature model. Also, a unified, generic, and extensible framework is proposed to create benchmarks of feature location techniques and enable users, developers, and researchers to analyze and compare different techniques (Martinez et al. 2015). However, the approach does not permit an incremental evolution of the SPL, and the feature location approach is not able to locate feature revisions. Furthermore, although it is not the focus of our work, we also present a mining tool to generate ground truth variants. Then, ground truth variants can be generated with our mining tool and the ones already used in our work are available too. Also, future work on locating feature revisions and re-engineering software system variants with multiple revisions can benefit of our ground truth generator.

9 Conclusions and Future Work

Existing feature location techniques are limited to analyzing specific system snapshots at one certain point in time. To address this limitation, this paper demonstrated the importance of feature location in both space and time and introduced an automated approach for feature revision location, which allows to reason about features in different points of time and supports software systems evolving in space and time. The results show that our approach can locate the features' artifacts with a precision of 100% at file-level and $\geq 99\%$ at line-level granularity, as well as a recall of 97% at file-level and 99% at line-level granularity. The incorrect information retrieved is due to the different syntactic structures of the semantic of a source code. Regarding the performance of our feature revision location approach, we reported that it took on average in the worst case 250 seconds and in the best case 25 seconds to trace artifacts to feature revisions for each input variant.

For composing a new variant, our approach took around 18 seconds on average of all systems to compose a variant. Even if manual completion is necessary, it will not require extensive code additions or deletions by a developer. The hints provided by our approach make it easier to find possible artifacts to be added or removed based on the presenting missing and surplus clauses containing the feature revisions and traces with conflicts and/or that do not exist in the repository. Thus, our automated approach can aid developers to evolve and maintain software systems at the level of feature revisions, thereby saving time and effort. Hence, it facilitates the management of system variability in space and time by composing variants with feature revisions easily and in a reasonable time. It also supports combining feature revisions that never were combined previously. Therefore, *ECSEST* provides additional functionalities than commit messages in Git VCS, such as location of feature revisions and combination of feature revisions from different commits.

We hope that our results will inspire researchers and tool builders to work with feature revisions to treat feature evolution in space and time, and will also encourage them to address current VarCSs limitations and/or improve other existing variability tools combined with a strategy for dealing with the evolution in time. We encourage future work comparisons with our work to reuse *ECSEST* approach's strength, or fulfill remaining gaps, and

improve its weaknesses/limitations by the use of common metrics, such as precision and recall, and by the dataset available²⁷ containing the ground truth used.

As future work, we want to conduct more experiments with industrial systems and from different domains, considering other programming languages such as Java, and other different artifact types. We also want to evaluate *ECSEST* for managing clones in product line engineering with feature revisions, using operations such as a Git VCS pull and push, but using a distributed ECCO repository for feature revisions to aid the implementation of system variants with feature revisions (Hinterreiter et al. 2021). In addition, we plan to improve our approach for dealing with evolution of dependencies and interactions in the source code of feature revisions, similar to Feichtinger et al. (Feichtinger et al. 2021), to automatically check for inconsistencies between feature revisions and their implementation when composing new configurations. Concluding, our future biggest goal is to provide an independent mechanism for enabling the management of variants with any combination of feature revisions.

Acknowledgements This research was funded by the LIT Secure and Correct Systems Lab; the Austrian Science Fund (FWF), grant no. P31989; Pro2Future, a COMET K1-Centre of the Austrian Research Promotion Agency (FFG), grant no. 854184; the Brazilian National Council for Scientific and Technological Development (CNPq), grant no. 408356/2018-9; and Carlos Chagas Filho Foundation for Supporting Research in the State of Rio de Janeiro (FAPERJ), program PDR-10 Fellowship, grant no. 202073/2020. The support by the Austrian Federal Ministry for Digital and Economic Affairs and the National Foundation for Research, Technology and Development is gratefully acknowledged. This work was partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) grant RGPIN-2017-0542, and also has been supported by the competence centers program COMET of the Austrian Research Promotion Agency (FFG), grant no. 865891.

Funding Open access funding provided by Johannes Kepler University Linz.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

- Ananieva S, Greiner S, Kühn T, Krüger J, Linsbauer L, Grüner S, Kehrer T, Klare H, Kozioliek A, Lönn H, Krieter S, Seidl C, Ramesh S, Reussner RH, Westfechtel B (2020) A conceptual model for unifying variability in space and time. In: Lopez-herrejon RE (ed) 24th ACM international systems and software product line conference, volume - a, SPLC '20. ACM, pp 15:1–15:12. <https://doi.org/10.1145/3382025.3414955>
- Angerer F, Grimmer A, Prähofner H, Grünbacher P (2019) Change impact analysis for maintenance and evolution of variable software systems. *Autom Softw Eng* 26:417–461. <https://doi.org/10.1007/s10515-019-00253-7>
- Apel S, Batory D, Kstner C, Saake G (2013) *Feature-Oriented Software product lines: Concepts and implementation*. Springer Publishing Company, Incorporated, New York

²⁷ <http://doi.org/10.5281/zenodo.4555199>

- Assunção KG, Vergilio SR (2014) Feature location for software product line migration: a mapping study. In: 18Th international software product line conference: Companion volume for workshops, demonstrations and tools - volume 2, SPLC 2014. ACM, New York, pp 52–59. <https://doi.org/10.1145/2647908.2655967>
- Bennett KH, Rajlich VT (2000) Software maintenance and evolution: a roadmap. In: Conference on the future of software engineering, ICSE '00. ACM, New York, pp 73–87. <https://doi.org/10.1145/336512.336534>
- Berger T, Chechik M, Kehrer T, Wimmer M (2019) Software evolution in time and space: Unifying version and variability management (dagstuhl seminar 19191). *Dagstuhl Rep* 9(5):1–30. <https://doi.org/10.4230/DagRep.9.5.1>
- Berger T, Lettner D, Rubin J, Grünbacher P, Silva A, Becker M, Chechik M, Czarnecki K (2015) What is a feature?: a qualitative study of features in industrial software product lines. In: 19Th international conference on software product line, SPLC 2015. ACM, New York, pp 16–25. <https://doi.org/10.1145/2791060.2791108>
- Berger T, She S, Lotufo R, Czarnecki K, Wasowski A (2010) Feature-to-code mapping in two large product lines. In: Bosch J, Lee J (eds) *Software product lines: Going beyond*. Springer, Berlin, pp 98–499
- Berger T, She S, Lotufo R, Wasowski A, Czarnecki K (2013) A study of variability models and languages in the systems software domain. *IEEE Trans Softw Eng* 39(12):1611–1640. <https://doi.org/10.1109/TSE.2013.34>
- Berger T, Steghöfer J, Ziadi T, Robin J, Martinez J (2020) The state of adoption and the challenges of systematic variability management in industry. *Empir Softw Eng* 25(3):1755–1797. <https://doi.org/10.1007/s10664-019-09787-6>
- Clements P, Northrop LM (2002) *Software product lines: Practices and patterns*. SEI series in software engineering. Addison-wesley, Boston
- Collins-Sussman B, Fitzpatrick BW, Pilato CM (2002) *Version Control with Subversion*. O'Reilly Media. <http://svnbook.red-bean.com/>
- Conradi R, Westfechtel B (1998) Version models for software configuration management. *ACM Comput Surv* 30(2):232–282. <https://doi.org/10.1145/280277.280280>
- Cruz D, Figueiredo E, Martinez J (2019) A literature review and comparison of three feature location techniques using argouml-spl. In: 13Th international workshop on variability modelling of software-intensive systems, VAMOS 2019. ACM, New York, pp 16:1–16:10. <https://doi.org/10.1145/3302333.3302343>
- Deorowicz S, Debudaj-Grabysz A, Gudyś A (2014) Kalign-LCS — a more accurate and faster variant of Kalign2 algorithm for the multiple sequence alignment problem. In: Gruca D. A., Czachórski T, Kozielski S. (eds) *Man-machine interactions 3*. Springer International Publishing, Cham, pp 495–502
- Dit B, Revelle M, Gethers M, Poshyanyk D (2013) Feature location in source code: a taxonomy and survey. *J Softw Evol Process* 25(1):53–95. <https://doi.org/10.1002/smr.567>
- Estublier J (2000) Software configuration management: a roadmap. In: Conference on the future of software engineering, ICSE '00. ACM, New York, pp 279–289. <https://doi.org/10.1145/336512.336576>
- Feichtinger K, Hinterreiter D, Linsbauer L, Prähofer H, Grünbacher P (2021) Guiding feature model evolution by lifting code-level dependencies. *J Comput Lang* 63:1–17. <https://doi.org/10.1016/j.cola.2021.101034>
- Fischer S, Linsbauer L, Lopez-Herrejon RE, Egyed A (2014) Enhancing clone-and-own with systematic reuse for developing software variants. In: 30Th IEEE international conference on software maintenance and evolution, ICSME 2014. IEEE, New York, pp 391–400. <https://doi.org/10.1109/ICSME.2014.61>
- Fischer S, Linsbauer L, Lopez-Herrejon RE, Egyed A (2015) The ecco tool: Extraction and composition for clone-and-own. In: 37Th IEEE international conference on software engineering, ICSE 2015, vol 2. IEEE, New York, pp 665–668. <https://doi.org/10.1109/ICSE.2015.218>
- Fischer S, Linsbauer L, Lopez-herrejon RE, Egyed A (2016) A source level empirical study of features and their interactions in variable software. In: 16Th international working conference on source code analysis and manipulation, SCAM 2016. IEEE, New York, pp 197–206
- Fischer S, Ramler R, Linsbauer L, Egyed A (2019) Automating test reuse for highly configurable software. In: 23Rd international systems and software product line conference, SPLC 2019. ACM, Paris, pp 1–11. <https://doi.org/10.1145/3336294.3336305>
- Gargantini A, Petke J, Radavelli M, Vavassori P (2016) Validation of constraints among configuration parameters using search-based combinatorial interaction testing. In: Sarro F, Deb K (eds) *Search based software engineering*. Springer International Publishing, New York, pp 49–63
- Ghabach E, Blay-fornarino M, Khoury FE, Baz B (2018) Clone-and-own software product derivation based on developer preferences and cost estimation. In: 12Th international conference on research challenges in information science. IEEE, pp 1–6. <https://doi.org/10.1109/RCIS.2018.8406682>
- Grünbacher P, Hanl R, Linsbauer L (2021) Using music features for managing revisions and variants in music notation software. In: Gottfried R, Hajdu G, Sello J, Anatrini A, MacCallum J (eds) *International conference on technologies for music notation and representation, TENOR'20/21*. Hamburg University for Music and Theater, Hamburg, pp 212–220

- Ha H, Zhang H (2019) Performance-influence model for highly configurable software with fourier learning and lasso regression. In: 35Th international conference on software maintenance and evolution, ICSME 2019. IEEE, New York, pp 470–480. <https://doi.org/10.1109/ICSME.2019.00080>
- Herzig K, Just S, Zeller A (2016) The impact of tangled code changes on defect prediction models. *Empir Softw Eng* 21(2):303–336. <https://doi.org/10.1007/s10664-015-9376-6>
- Hinterreiter D, Linsbauer L, Feichtinger K, Prähofer H, Grünbacher P (2020) Supporting feature-oriented evolution in industrial automation product lines. *Concurr Eng Res Appl* 28:265–279. <https://doi.org/10.1177/1063293X20958930>
- Hinterreiter D, Linsbauer L, Grünbacher P., Prähofer H. (2021) Feature-oriented clone and pull for distributed development and evolution. In: 14Th international conference on the quality of information and communications technology, QUATIC '21
- Hinterreiter D, Nieke M, Linsbauer L, Seidl C, Prähofer H, Grünbacher P (2019) Harmonized temporal feature modeling to uniformly perform, track, analyze, and replay software product line evolution. In: 18Th international conference on generative programming: Concepts & experiences, GPCE 2019. ACM, New York, pp 115–128. <https://doi.org/10.1145/3357765.3359515>
- Krüger J, Mukelabai M, Gu W, Shen H, Hebig R, Berger T (2019) Where is my feature and what is it about? a case study on recovering feature facets. *J Syst Softw* 152:239–253. <https://doi.org/10.1016/j.jss.2019.01.057>
- Krüger J, Calikli G, Berger T, Leich T (2021) How explicit feature traces did not impact developers' memory. In: 28Th IEEE international conference on software analysis, evolution and reengineering, SANER '21. IEEE, pp 610–613. <https://doi.org/10.1109/SANER50967.2021.00075>
- Krüger J., Gu W, Shen H, Mukelabai M, Hebig R, Berger T (2018) Towards a better understanding of software features and their characteristics: a case study of marlin. In: 12Th international workshop on variability modelling of software-intensive systems, VAMOS 2018. ACM, New York, pp 105–112. <https://doi.org/10.1145/3168365.3168371>
- Liebig J, Apel S, Lengauer C, Kästner C, Schulze M (2010) An analysis of the variability in forty preprocessor-based software product lines. In: 32Nd ACM/IEEE international conference on software engineering - volume 1, ICSE 2010. ACM, New York, pp 105–114. <https://doi.org/10.1145/1806799.1806819>
- Linsbauer L, Berger T, Grünbacher P (2017) A classification of variation control systems. In: Flatt M, Erdweg S (eds) 16Th international conference on generative programming: Concepts and experiences, GPCE '17. ACM, New York, pp 49–62. <https://doi.org/10.1145/3136040.3136054>
- Linsbauer L, Egyed A, Lopez-herrejon RE (2016) A variability aware configuration management and revision control platform. In: Dillon LK, Visser W, Williams LA (eds) 38th International Conference on Software Engineering, ICSE '16. ACM, pp 803–806. <https://doi.org/10.1145/2889160.2889262>
- Linsbauer L, Fischer S, Lopez-Herrejon RE, Egyed A (2015) Using traceability for incremental construction and evolution of software product portfolios. In: 8Th international symposium on software and systems traceability, SST 2015. IEEE, New York, pp 57–60. <https://doi.org/10.1109/SST.2015.16>
- Linsbauer L, Lopez-Herrejon ER, Egyed A (2013) Recovering traceability between features and code in product variants. In: 17Th international software product line conference, SPLC 2013. ACM, New York, pp 131–140. <https://doi.org/10.1145/2491627.2491630>
- Linsbauer L, Lopez-herrejon RE, Egyed A (2017) Variability extraction and modeling for product variants. *Softw Syst Model* 16(4):1179–1199. <https://doi.org/10.1007/s10270-015-0512-y>
- Linsbauer L, Lopez-herrejon RE, Egyed A (2017) Variability extraction and modeling for product variants. *Softw Syst Model* 16(4):1179–1199. <https://doi.org/10.1007/s10270-015-0512-y>
- Linsbauer L, Schwägerl F, Berger T, Grünbacher P (2021) Concepts of variation control systems. *J Syst Softw* 171:110,796. <https://doi.org/10.1016/j.jss.2020.110796>
- Liu J, Batory D, Lengauer C (2006) Feature oriented refactoring of legacy applications. In: 28Th international conference on software engineering, ICSE 2006. ACM, New York, pp 112–121. <https://doi.org/10.1145/1134285.1134303>
- MacKay SA (1995) The state of the art in concurrent, distributed configuration management. In: Selected papers from the ICSE SCM-4 and SCM-5 workshops, on software configuration management. Springer, Berlin, pp 180–193
- Martinez J (2016) Mining software artefact variants for product line migration and analysis. Ph.D. thesis, Pierre and Marie Curie University, France. <http://orbilu.uni.lu/handle/10993/28675>
- Martinez J, Ziadi T, Bissyandé TF, Klein J, Le Traon Y (2015) Bottom-up adoption of software product lines: a generic and extensible approach. In: 19Th international conference on software product line, SPLC '15. ACM, New York, pp 101–110. <https://doi.org/10.1145/2791060.2791086>
- Martinez J, Ziadi T, Papadakis M, Bissyandé TF, Klein J, le Traon Y (2018) Feature location benchmark for extractive software product line adoption research using realistic and synthetic eclipse variants. *Inf Softw Technol* 104:46–59. <https://doi.org/10.1016/j.infsof.2018.07.005>

- McGovern J, Ambler SW, Stevens ME, Linn J, Jo EK, Sharan V (2003) The practical guide to enterprise architecture. Prentice Hall, PTR
- Medeiros F, Kästner C, Ribeiro M, Nadi S, Gheyi R (2015) The love/hate relationship with the C preprocessor: an interview study (artifact). *Dagstuhl Artifacts Ser* 1(1):07:1–07:32. <https://doi.org/10.4230/DARTS.1.1.7>
- Medeiros F, Ribeiro M, Gheyi R, Apel S, Kästner C, Ferreira B, Carvalho L, Fonseca B (2018) Discipline matters: Refactoring of preprocessor directives in the #ifdef hell. *IEEE Trans Softw Eng* 44(5):453–469. <https://doi.org/10.1109/TSE.2017.2688333>
- Melo J, Brabrand C, Wasowski A (2016) How does the degree of variability affect bug finding? In: 38Th international conference on software engineering, ICSE '16. ACM, New York, pp 679–690. <https://doi.org/10.1145/2884781.2884831>
- Michelon GK, Assunção WKG, Obermann D, Linsbauer L, Grünbacher P, Egyed A (2021a) The life cycle of features in highly-configurable software systems evolving in space and time. In: 20Th international conference on generative programming: Concepts & experiences, GPCE 2021. ACM, New York, pp 1–14. <https://doi.org/10.1145/3486609.3487195>
- Michelon GK, Linsbauer L, Assunção WKG, Egyed A (2019) Comparison-based feature location in argouml variants. In: 23Rd international systems and software product line conference - Volume A, SPLC 2019. ACM, pp 17:1–17:5. <https://doi.org/10.1145/3336294.3342360>
- Michelon GK, Linsbauer L, Assunção WKG, Fischer S, Egyed A (2021b) A hybrid feature location technique for re-engineering single systems into software product lines. In: Grünbacher P, Seidl C, Dhungana D, Lovasz-Bukvova H (eds) 15th International Working Conference on Variability Modelling of Software-Intensive Systems, VaMoS '21. ACM, pp 11:1–11:9. <https://doi.org/10.1145/3442391.3442403>
- Michelon GK, Obermann D, Assunção WKG, Linsbauer L, Grünbacher P, Egyed A (2021c) Managing systems evolving in space and time: four challenges for maintenance, evolution and composition of variants. In: 25Th international systems and software product line conference - Volume A. ACM, New York, pp 75–80. <https://doi.org/10.1145/3461001.3461660>
- Michelon GK, Obermann D, Assunção WKG, Linsbauer L, Grünbacher P, Egyed A (2020a) Mining feature revisions in highly-configurable software systems. In: 24Th ACM international systems and software product line conference - Volume B, SPLC '20. ACM, New York, pp 74–78. <https://doi.org/10.1145/3382026.3425776>
- Michelon GK, Obermann D, Linsbauer L, Assunção WKG, Grünbacher P, Egyed A (2020d) Locating feature revisions in software systems evolving in space and time. In: Lopez-herrejon RE (ed) 24th ACM international systems and software product line conference, volume - a, SPLC '20. ACM, pp 14:1–14:11. <https://doi.org/10.1145/3382025.3414954>
- Michelon GK, Sotto-Mayor B, Martinez J, Arrieta A, Abreu R, Assunção W. K. G. (2021d) Spectrum-based Feature Localization: A Case Study Using argoUML, SPLC '21, ACM, New York. <https://doi.org/10.1145/3461001.3473065>
- Nassif M, Robillard MP (2017) Revisiting turnover-induced knowledge loss in software projects. In: 2017 IEEE International conference on software maintenance and evolution, ICSME '17. IEEE computer society, pp 261–272. <https://doi.org/10.1109/ICSME.2017.64>
- Passos L, Padilla J, Berger T, Apel S, Czarnecki K, Valente MT (2015) Feature scattering in the large: a longitudinal study of linux kernel device drivers. In: 14Th international conference on modularity, MODULARITY 2015. ACM, New York, pp 81–92. <https://doi.org/10.1145/2724525.2724575>
- Pereira JA, Acher M, Martin H, Jézèquel J (2020) Sampling effect on performance prediction of configurable systems: A case study. In: Amaral JN, Koziolok A, Trubiani C, Iosup A (eds) International Conference on Performance Engineering, ICPE '20. ACM, pp 277–288. <https://doi.org/10.1145/3358960.3379137>
- Pohl K, Böckle G., Linden FJvd (2005) Software Product Line Engineering: foundations, Principles and Techniques. Springer, Berlin
- Pohl K, Metzger A (2018) Software Product Lines. Springer International Publishing, Cham, pp 185–201. https://doi.org/10.1007/978-3-319-73897-0_11
- Rabiser R, Grünbacher P, Lehofer M (2012) A qualitative study on user guidance capabilities in product configuration tools. In: International conference on automated software engineering, ASE '12. ACM, pp 110–119. <https://doi.org/10.1145/2351676.2351693>
- Rubin J, Chechik M (2013) A survey of feature location techniques. In: Domain Engineering, Product Lines, Languages, and Conceptual Models. Springer, Berlin, pp 29–58. https://doi.org/10.1007/978-3-642-36654-3_2
- Schwägerl F (2018) Version control and product lines in model-driven software engineering. Ph.D. thesis, University of Bayreuth, Germany
- Schwägerl F, Westfechtel B (2016) Supermod: tool support for collaborative filtered model-driven software product line engineering. In: Lo D, Apel S, Khurshid S (eds) 31st International Conference on Automated Software Engineering, ASE '16. ACM, pp 822–827. <https://doi.org/10.1145/2970276.2970288>

- Schwägerl F, Westfechtel B (2019) Integrated revision and variation control for evolving model-driven software product lines. *Softw Syst Model* 18(6):3373–3420. <https://doi.org/10.1007/s10270-019-00722-3>
- Seidl C, Schaefer I, Aßmann U (2013) Capturing variability in space and time with hyper feature models. In: 8Th international workshop on variability modelling of software-intensive systems, VAMOS 2014. ACM, New York, pp 6:1–6:8. <https://doi.org/10.1145/2556624.2556625>
- Seidl C, Schaefer I, Aßmann U (2014) Deltaecore – A model-based delta language generation framework. In: Fill H, Karagiannis D, Reimer U (eds) *Modellierung 2014*, LNI, vol P-225, pp 81–96
- Sincero J, Schirmeier H, Schröder-Preikschat W, Spinczyk O (2007) Is The Linux Kernel a Software Product Line?. In: van der Linden, F, Lundell B (eds) *International Workshop on Open Source Software and Product Lines, SPLC-OSSPL '07*, Kyoto
- Strüder D, Mukelabai M, Krüger J, Fischer S, Linsbauer L, Martinez J, Berger T (2019) Facing the truth: Benchmarking the techniques for the evolution of variant-rich systems. In: 23Rd international systems and software product line conference - Volume A, SPLC '19. ACM, New York, pp 177–188. <https://doi.org/10.1145/3336294.3336302>
- Ting KM (2010) *Precision and Recall*. Springer US, Boston. https://doi.org/10.1007/978-0-387-30164-8_652
- Vale T, Almeida ES (2019) Experimenting with information retrieval methods in the recovery of feature-code SPL traces. *Empir Softw Eng* 24(3):1328–1368. <https://doi.org/10.1007/s10664-018-9652-3>
- Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A (2000) *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers, USA. <https://doi.org/10.1007/978-1-4615-4625-2>

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.



Gabriela Karoline Michelson is currently a PhD student at the Institute for Software Systems Engineering (ISSE) and LIT Secure and Correct Systems Lab at the Johannes Kepler University Linz (JKU) - Austria. Gabriela received her M.Sc. in Informatics (2018) from the Federal Technological University of Paran (UTFPR) - Brazil with a period of three months in University of California (UC Davis) - United States. Her areas of interest are variability management, software systems evolution, highly configurable software systems, software product lines, and version control systems. Website: <https://gabrielamichelson.github.io/>



David Obermann is currently a master's computer science student at the Johannes Kepler University (JKU) in Linz, Austria. Until 2020 he worked as a research assistant at the Institute for Software Systems Engineering (ISSE) at JKU Linz where he assisted in the areas of software systems evolution, variability management and highly configurable systems.



Wesley K. G. Assunção is currently a University Assistant at Johannes Kepler University Linz (JKU) - Austria, Post-Doctoral researcher at Pontifical Catholic University of Rio de Janeiro (PUC-Rio) - Brazil, and visiting professor at the Graduate Program in Computer Science at Western Paran State University (Unioeste) - Brazil. Wesley received his M.Sc. in Informatics (2012) and Ph.D. in Computer Science (2017) both from Federal University of Paran (UFPR) - Brazil. His areas of interest are Software Modernization, Variability Management, Collaborative Engineering of Complex Systems, Software Testing, and Search Based Software Engineering. He published research papers, in collaboration with international researchers, in conferences like ICSME, SANER, MSR, EASE, SPLC, SSBSE, GECCO, to cite some, as well as in journals such as EMSE, IST, and JSS. Wesley has also been serving as reviewers for many conferences and journal, and as organizer of conference, symposiums, workshops, competitions, and meetings. Website: <https://wesleyklewerton.github.io/>



Lukas Linsbauer is currently a postdoctoral researcher at the Institute of Software Engineering and Automotive Informatics at the Technische Universität Braunschweig in Germany. He received his Doctorate in 2016 from the Institute for Software Systems Engineering at the Johannes Kepler University Linz in Austria under the supervision of Prof. Alexander Egyed and Dr. Roberto Erick Lopez-Herrejon. His research interests include highly variable and configurable systems, software product lines, feature-oriented software and systems development, clone detection, and version control systems.



Paul Grünbacher is an Associate Professor at the Institute of Software Systems Engineering at Johannes Kepler University Linz (Austria). His research interests include software product lines, model-based development and evolution, requirements engineering, and software monitoring. From 2013-2021 Paul headed the Christian Doppler Laboratory for Monitoring and Evolution of Very-Large-Scale Software Systems. He has published more than 150 papers in international peer-reviewed journals, conferences, and workshops. Paul is an Editorial Board Member of the Empirical Software Engineering Journal (Springer) and the Information and Software Technology Journal (Elsevier). He is regularly serving as a reviewer for international journals and conferences. He is member of ACM, ACM SIGSOFT, the IEEE CS, the Austrian Computer Society, and Euromicro. In 2021 he was designated as a Fellow of Automated Software Engineering by the Steering Committee of the IEEE/ACM International Conferences on Automated Software Engineering.

Stefan Fischer received the M.Sc. and Doctoral degree in software engineering and computer science from Johannes Kepler University Linz, Linz, Austria. He is a Senior Researcher in the Software Competence Center Hagenberg, Hagenberg, Austria. He has several years of experience in software engineering research and technology transfer. His main research interests include configuration-aware software testing, and software analytics.




Roberto E. Lopez-Herrejon is an Associate Professor at the Department of Software Engineering and Information Technology of the Ecole de Technologie Supérieure of the University of Quebec in Montreal, Canada. Prior he was a senior postdoctoral researcher at the Johannes Kepler University in Linz, Austria. He was an Austrian Science Fund (FWF) Lise Meitner Fellow (2012-2014) at the same institution. From 2008 to 2014 he was an External Lecturer at the Software Engineering Masters Programme of the University of Oxford, England. From 2010 to 2012 he held an FP7 Intra-European Marie Curie Fellowship sponsored by the European Commission. He obtained his Ph.D. from the University of Texas at Austin in 2006, funded in part by a Fulbright Fellowship sponsored by the U.S. State Department. From 2005 to 2008, he was a Career Development Fellow at the Software Engineering Centre of the University of Oxford sponsored by Higher Education Funding Council of England (HEFCE). His main expertise is in software customization, software product lines, and search based software engineering.



Alexander Egyed is Professor for Software-Intensive Systems at the Johannes Kepler University, Austria. He received his Doctorate from the University of Southern California, USA and worked in industry for many years. He is most recognized for his work on software and systems design - particularly on variability, consistency, and traceability.

Affiliations

Gabriela Karoline Michelin¹  · **David Obermann**² · **Wesley K. G. Assunção**^{2,3} · **Lukas Linsbauer**⁴ · **Paul Grünbacher**² · **Stefan Fischer**⁵ · **Roberto E. Lopez-Herrejon**⁶ · **Alexander Egyed**²

¹ Institute for Software Systems Engineering, LIT Secure and Correct Systems Lab, Johannes Kepler University Linz, Linz, Austria

² Institute for Software Systems Engineering, Johannes Kepler University Linz, Linz, Austria

³ Pontifical Catholic University of Rio de Janeiro, Rio de Janeiro, Brazil

⁴ Institute of Software Engineering and Automotive Informatics, Technische Universität Braunschweig, Braunschweig, Germany

⁵ Software Competence Center Hagenberg GmbH, Hagenberg, Austria

⁶ École de Technologie Supérieure (ÉTS), University of Quebec, H3C 1K3, Montreal, Quebec, Canada