A Technology-Neutral Role-Based Collaboration Model for Software Ecosystems

Ştefan Stănciulescu[1], Daniela Rabiser[2], and Christoph Seidl[3]

[1] IT University of Copenhagen, Denmark
scas@itu.dk
[2] Johannes Kepler University Linz, CD Lab MEVSS, Austria
daniela.rabiser@jku.at
[3] Technische Universität Braunschweig, Germany
c.seidl@tu-braunschweig.de

Abstract In large-scale software ecosystems, many developers contribute extensions to a common software platform. Due to the independent development efforts and the lack of a central steering mechanism, similar functionality may be developed multiple times by different developers. We tackle this problem by contributing a role-based collaboration model for software ecosystems to make such implicit similarities explicit and to raise awareness among developers during their ongoing efforts. We extract this model based on realization artifacts in a specific programming language located in a particular source code repository and present it in a technology-neutral way. We capture five essential collaborations as independent role models that may be composed to present developer collaborations of a software ecosystem in their entirety, which fosters overview of the software ecosystem, analyses of duplicated development efforts and information of ongoing development efforts. Finally, using the collaborations defined in the formalism we model real artifacts from Marlin, a firmware for 3D printers, and we show that for the selected scenarios, the five collaborations were sufficient to raise awareness and make implicit information explicit.

Keywords: Software Ecosystem, Collaboration, Role Modeling, Marlin

1 Introduction

The software product line methodology allows to manage similar software systems by developing the core assets that are common to all systems and developing variant assets that are product specific [7]. Software ecosystems [5] (SECOs) provide a common platform to develop a family of closely related software systems having distinct characteristics. SECOs address development contexts which typically involve multiple organizations and product lines. However, both methodologies try to maximize reuse of software, reducing efforts in development as well as during maintenance of the products. With the advent of Github[4],

[4] http://www.github.com

projects share a common repository where changes can be pushed to from private repositories. This development process follows a similar path to the one of software ecosystems, where vendors contribute and maintain the platform without having a centralized mechanism.

Motivation: Products are frequently developed using a clone-and-own reuse approach by adapting existing solutions to create new customer-specific products. However, deviations from the reusable platform code (e.g., customer-specific features) remain largely undocumented. Empirical studies present this as an emerging challenge [33,22,21,12,11]. A lightweight solution for improving knowledge is to enhance awareness between developers. Lettner[5] et al. [23], propose the feature feed approach, which supports making specific implementations visible to interested users within the SECO via notifications.

Problem: Finding existing features or similar ones in the ecosystem is hard even for the community members. This is consistent with an observation of Berger et al. [4] that an excessive use of clone-and-own variants in industrial projects leads to loss of overview of the available functionality. Decentralization of information also leads to a loss of overview [12]. For example, in an open source 3D printer firmware project, 14% of pull requests were rejected because of concurrent development [33].

Contribution: The main contribution of this paper is a formalism based on role models to describe collaborations of contributors and artifacts in an ecosystem, with the main purpose of providing an overview of the ecosystem in terms of relations between users, repositories and features. For example, one developer that develops a similar feature to already existing one in another repository could be informed of the already existing code, and a link that describes the two features as being similar would be created. We describe the formalism and how it can be used to tackle existing challenges, i.e., raising developer awareness of concurrent feature development. Our long term goal is to build an automatic process of constructing collaboration models in a SECO. We use an exploratory case study to test and evaluate the feasibility and expressiveness of the formalism in the context of Marlin, a firmware for 3D printers developed and maintained as an open source project on Github.

The paper is structured as following: in Sec. 2 we introduce required background knowledge. In Sec. 3 we motivate this work using an example and extract requirements for the collaboration role model, and present the collaboration role model in Sec. 4. We present our evaluation on an exploratory case study in Sec. 5, related work in Sec. 6, and we conclude in Sec. 7.

2 Background

In this section, we introduce the basic notions of role models, features, and Github's forking mechanism.

Role models. A role model describes a (possibly infinite) set of object collaborations using role types [27]. Its focus lies on representing a single purpose of

[5] Daniela Rabiser's previous work was published under the name Daniela Lettner.

an object collaboration. Each role type specifies the behavior of one particular object with respect to the model's purpose. Role types relate to each other via relationships such as association and aggregation. To describe a concrete instance of a collaboration, roles may be mapped to various elements, e.g., source code fragments, users or repositories, depending on the respective collaboration. Role models may be complemented with formal ontologies providing standard terminologies and rich semantics to facilitate knowledge sharing and reuse [34,18].

Features. Commonalities and variability of a product portfolio are often captured using the abstract concept of features [3]. A feature has been defined as distinguishable characteristic of a concept (e.g., system, component) that is relevant to some stakeholder of the concept [8]. In the context of SECOs, a feature denotes a unit of configurable functionality. Software artifacts that implement specific program functionalities, i.e., features, can be discovered using feature location techniques [28]. However, there is no optimal feature location technique and the notion of feature varies widely in practice [3].

Github forking. Github[6] is a social coding platform that allows collaborative development. Github offers the *forking* mechanism to create a copy of a repository, together with a traceability link between the copied repository, the fork, and the original project. On Github, users can create pull requests which resemble traditional change requests. A pull request consists of a description, possible comments from users, and a set of commits. A pull request can be created either in the same repository, e.g., to allow a team to discuss the change, or from a fork to the original project. Forking in this sense is also known as the *pull-based development model*. Most often, forks are used to develop and test changes in isolation, and then those changes are integrated into the original repository using pull requests.

3 Challenges Arising in Software Ecosystems

The pull-based software development was investigated on its usage in practice by Gousios et al. [16], who found that around 14% of the repositories in GitHub used the pull request mechanism (data until 2013). The most common reasons for rejecting a pull request are concurrent modifications, the way a project handles distributed development (e.g., newer pull requests are chosen over older pull requests addressing the same problem), or changes not matching a project's road map. Duc et al. [12] also confirmed in an industrial setting that there is a loss of overview of which elements exist and who is working on which element, when clone-and-own is used as a reuse mechanism between teams. Berger et al. [4] found in their empirical investigation on industrial systems that too much cloning of variants obscures what functionality is available. The same issues were also observed by Stănciulescu et al. [33] in an open source project that has been heavily forked on GitHub. The authors identified that many pull requests were closed because of double development, overlapping or already existing code. This leads to time and effort wasted. Moreover, one

[6] https://github.com

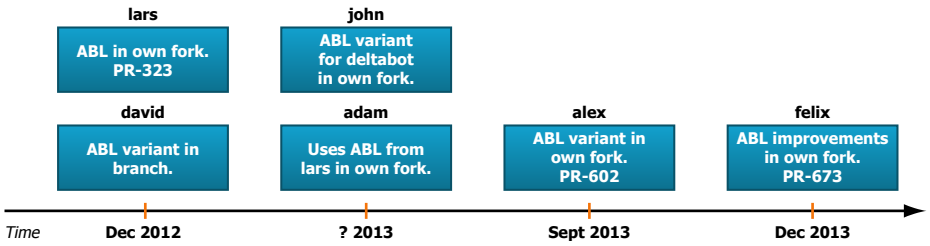

Figure 1. Timeline showing the evolution of the ABL feature.

of the maintainers explained that they are unaware of which elements exist in the SECO (e.g., forks, bug-fixes, improvements).

3.1 Motivating Example

Marlin[7] is a 3D printer firmware that is being developed as open source since 2011. Marlin has been forked by more than 2900 people on Github, with many of them contributing new features and testing the firmware. Due to the sheer amount of forks, it is difficult to get and maintain an accurate overview of what exists in the firmware's ecosystem, what features are being developed and what collaborations exist.

We use the feature *Auto Bed Leveling (ABL)* from Marlin to illustrate the challenges developers are faced with when they try to get an accurate overview of what exists related to a specific feature. The ABL feature is for computing bed tilt compensation for better results printing on a non-level bed. We extracted the example feature from Marlin's history. Fig. 1 shows the evolution of the ABL feature. Initially, it was developed experimentally in a fork by user *lars*[8] without being integrated in the main repository. From pull request PR-323[9], we can see that there was another version being developed by user *david* (one of the repository maintainers and developer of Marlin at that time) but not made public. Yet another form of this feature existed in another fork of Marlin that was a variant to support a different type of printers developed by user *john*. In September 2013, the ABL feature was ported and integrated into Marlin from the initial fork by user *alex* who had commit access to Marlin. It is unclear if there existed other variants of this feature at that time. Somewhere in between, user *adam* has integrated the ABL feature from *lars*' fork. Following the integration of ABL in September 2013, other users (e.g., user *felix*) started contributing with improvements and bug fixes of the ABL feature in their own fork. These users created pull requests (PR-673) to integrate their changes back into Marlin.

Even when having access to this data offhand, it is difficult to understand the evolution of the ABL feature and to reveal what existed at different points in time. Furthermore, since information related to the ABL feature is spread

[7] http://github.com/MarlinFirmware/
[8] We anonymized the names of the Github users to respect their privacy
[9] https://github.com/MarlinFirmware/Marlin/pull/323

across multiple repositories, it is hard to get an accurate overview. Moreover, the notion of a feature is not explicit in many cases but has to be made explicit from code annotations or developer documentation. When considering other SECOs, the problems in analyzing data are even amplified by different realization technologies, such as programming languages (e.g., Java, C++, Python) or repositories (e.g., Git, Mercurial, SVN, CVS).

3.2 Derived Requirements

A formalism is needed that lifts information on related (e.g., similar, improved) features being developed by different users in diverse repositories. Specifically, the formalism needs to address four requirements:

Requirement R1 – Analyze and present data technology-independent. The formalism needs to be agnostic of specific repository types and programming languages used. The main challenge is to be able to deal with different technologies using minimum effort.

Requirement R2 – Lift implicit knowledge. Implicit knowledge needs to be revealed and made explicit using repository analysis techniques. Furthermore, introducing a dedicated feature concept seems promising to link artifacts (e.g., source code, configuration options, documentation, tests) spread across repositories realizing a particular feature. Additionally, developers being responsible for specific feature implementations would be able to document their expert knowledge and, thus, make it available for other developers planning to reuse a feature or contributing to a feature.

Requirement R3 – Improve developer awareness. Developers should be aware of relevant features developed in other repositories of the SECO. The decision on whether a feature is relevant or not may be based on clone detection techniques or recommendation mechanisms and the features a developer has worked on or is currently working on.

Requirement R4 – Support modularization of strongly related elements. The formalism needs to support multiple levels of abstraction to facilitate zooming in to a specific feature, user or repository [26]. Such compartmentalization of strongly related entities (i.e., adhering to divide-and-conquer strategies and breaking down large-scale SECOs into smaller parts) further supports developers in understanding and reasoning about specific parts of a SECO. On the other hand, composing smaller parts into large ones could be effective in cases where an overview has to be included.

4 Collaboration Role Modeling

The main contribution of this paper is a *SECO collaboration role model* describing the relationships between repositories, users and features. Role modeling is not the same as programming language, even though concepts of relations, types and instances exist in role modeling as well. Role models are technology-neutral and can be used to guide collaboration between developers and to monitor

communications. Collaboration models not just lift explicit information, they also provide insights into implicit communication edges. Role modeling focuses mostly on collaborations and is more fine-grained than class modeling. Furthermore, roles can be attributed to artifacts other than source code, such as repositories or users.

We use the collaboration role model as a way of understanding what entities and what kinds of collaborations between entities exist in a SECO. The mentioned entities—repositories, users and features—can be represented differently in concrete implementations (e.g., repositories: SVN, Git, CVS, Perforce, TeamServer; users: different account management systems; features: implicit or explicit, different realizations techniques such as annotations or feature-oriented programming). The collaboration role model is technology-neutral in the sense that it captures the relations and discards any information regarding the technologies used.

We present five distinct collaborations, each containing relevant roles and their relations for a specific concern of the SECO. These collaborations can be classified as either presenting inherent knowledge available implicitly in the repositories and making it explicit, such as the collaborations *development and usage* and *storage* do, or providing added information, such as, *subscription* and *recommendation* do. The collaboration *origin* provides both inherent (e.g., integrated feature) and added information (e.g., similar feature). In the following, we first describe each of the individual collaborations along with the concerns they address before elaborating on how to compose them to form a comprehensive SECO model.

4.1 Development and Usage Collaboration

The role model depicted in Fig. 2 defines roles related to feature development and feature usage. A user may either develop or implement a feature or re-use an existing feature implementation in her own context. Features are typically related to a dedicated feature location, e.g., a development branch in the repository. This role is needed to be able to create links between users and features. Knowing who uses the feature provides the basis for informing appropriate users of important fixes. Moreover, developers working on a feature can find other developers that have worked on the feature, or on similar features, and establish collaborations with them. It makes implicit information explicit, e.g., it introduces an explicit feature concept even if the underlying technology captures features only implicitly using different technologies (e.g., C preprocessor directives).

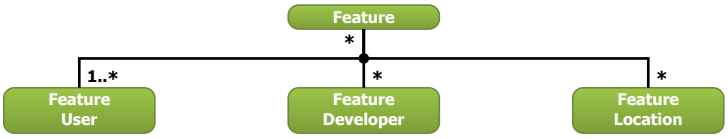

Figure 2. Development and usage collaboration

4.2 Storage Collaboration

Repositories are one of the most important elements in a SECO. They store all artifacts related to a concrete project. A repository is managed by at least one maintainer. Furthermore, contributors or users work with a repository, as depicted in Fig. 3. A maintainer is a user who is responsible for managing issues, incoming changes and other artifacts of the repository. The maintainer is allowed to accept and merge changes from external collaborators to the repository. A repository user benefits from a repository without contributing or having any maintenance tasks. A repository fork indicates that one repository is a clone of an existing repository. This collaboration model offers coarse grained information about the users of the repository and is particularly useful to document the origin of per forked repository.

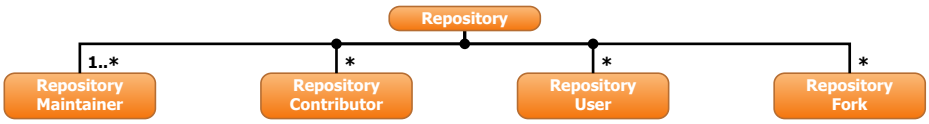

Figure 3. Storage collaboration

4.3 Origin Collaboration

The role model depicted in Fig. 4 shows the relations between several features that exist in the ecosystem. For example, a newly developed feature can be similar to another feature that already exists. This can be represented by using the relation between an original feature (the older one), and a similar feature (the newer one). This information could be (semi-)automatically extracted by a clone detection system [29] using the SECO and repository information. Furthermore, a feature may be enhanced by additional functionality to consider it as an improved feature of the original feature. Finally, an integrated feature represents a feature that is integrated in the main repository of the SECO.

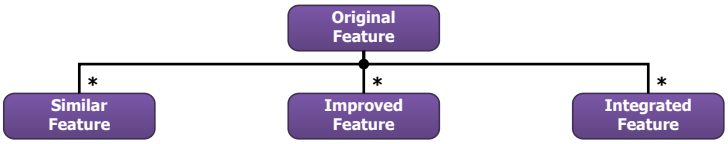

Figure 4. Origin collaboration

4.4 Subscription Collaboration

In addition to the roles depicted in Fig. 2, we define a subscription collaboration in Fig. 5 which follows the publish-subscribe pattern [13]. A user can subscribe to

a dedicated artifact. Based on a subscription, notifications are pushed to the subscribers. Such artifacts of interest can be of different granularity, from entire repositories (coarse) over features to lines of code in a file (fine-grained). This is similar to what exists currently in social coding platforms (e.g., Github or Bitbucket), but it is less restrictive allowing different levels of granularity. The role is designed specifically towards increasing developer awareness by using a notification system.

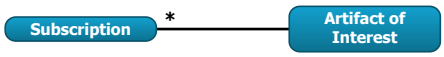

Figure 5. Subscription collaboration

4.5 Recommendation Collaboration

As in the case of the subscription collaboration, the main purpose of the recommendation collaboration depicted in Fig. 6 is to enhance visibility of existing features in the ecosystem, and provide useful recommendations of existing features. Feature recommendations could be automatically provided using a recommender system based on a database of preferences for items by users. For instance, a new user would be matched against the database to discover neighbors, which are other users who have historically had similar interests. Items that the neighbors like would then be recommended to the new user [30].

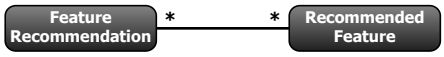

Figure 6. Recommendation collaboration

4.6 Collaboration Role Relations

The five collaborations provide a unified way to represent individual concerns of a SECO relevant to determine the relation of features, users and repositories. However, the collaborations are not completely isolated from one another. In fact, part of our notation connects selected roles of different collaborations implicitly. We define concrete *collaboration role relations* to specify how specific roles of individual or different collaborations are related to one another as depicted in Fig. 7. These relations further ensure flexibility regarding adding new collaboration types. For instance, if the model needs to be extended regarding the relation of product variants and features, one would integrate a parent-child relation into Fig. 7 indicating that a feature is the child of one or more product variant(s).

In many cases, the fact that an element plays one role automatically entails that it plays a certain other role as well. We describe this using the *role implication* (arrow with hollow arrow tip) as relation between roles of potentially different

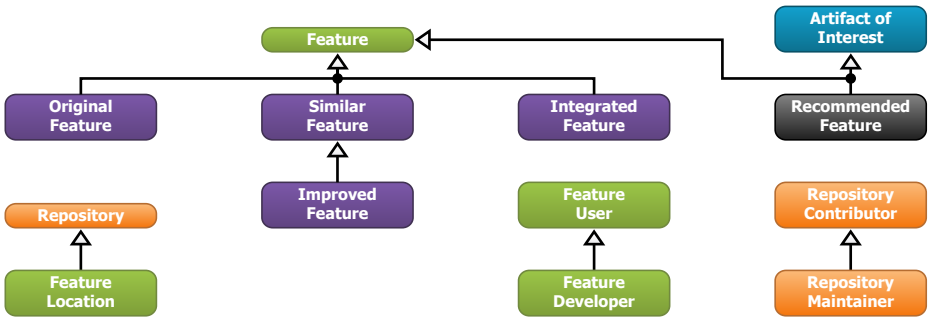

Figure 7. Collaboration role relations connecting roles of individual collaborations

collaborations. The role implication denotes that, whenever an element plays the role of the premise, it implicitly also plays the implied role in the conclusion. For example, the roles *original feature*, *similar feature*, *improved feature* and *integrated feature* from the *origin* collaboration all implicitly play the role *feature* from the *development and usage* collaboration. Furthermore, the enhancements in the role *improved feature* over those of the role *original feature* entail that each element playing the role *improved feature* also plays the role *similar feature*. Likewise, the role *recommended feature* from the *recommendation* collaboration entails the role *feature* and the role *artifact of interest* from the *subscription* collaboration. Furthermore, the *feature location* from the *development and usage* collaboration entails a *repository* role from the *storage* collaboration being mapped to the same element. Finally, being a *feature developer* also means being a *feature user* as well as being a *repository maintainer* means being a *repository contributor*.

The visual representations depicted in Fig. 8, Fig. 9 and Fig. 10 make the implied roles explicit for easier legibility but it would also be possible to determine the implied rules via a reasoning mechanism instead.

4.7 Instantiating Collaborations and Composing a SECO Model

To apply our notation to a concrete SECO, the collaborations are applied by mapping the individual roles to concrete elements to signal that element plays that role. This forms a comprehensive SECO model, which can be used for analyses, due to two reasons:

First, roles of different collaborations may be mapped to the same element to signal that one individual acts in multiple roles at once. For example, in Fig. 8, the roles *feature* and *original feature* are both mapped to the *AutoBedLeveling* functionality to signal both roles of the functionality. In consequence, it is possible to determine that user *lars* was the developer of the original feature for auto bed leveling by navigating from *feature developer* to *feature* and then *original feature*.

Second, the described collaboration role relations specify how relevant roles of individual or different collaborations are related to one another to allow navigation accross collaboration boundaries. For example, in Fig. 8, the fact that *lars/Marlin*

plays the role *feature location* of the *development and usage collaboration* automatically entails that it also plays the role *repository* of the *storage* collaboration. Furthermore, the fact that user *lars* plays the role *feature developer* automatically entails that he plays the role *feature user* as well. This allows navigation of the SECO model even if part of this information are only available implicitly.

It is worth noting that, in the mapping process, each collaboration may be applied multiple times with a different context, e.g., the *storage* collaboration could be instantiated multiple times if one person contributes to or maintains multiple different repositories. Hence, our notation of five individual collaborations and the collaboration role relations of Fig. 7 permits the creation of comprehensive SECO models.

5 Exploratory Case Study

We illustrate the introduced SECO collaboration role model in the context of our case study subject system Marlin [33]. Marlin exhibits the main challenges we want to address and we have access to a database containing commits, issues, pull requests and other meta-data[10]. Our research objective aims at studying the expressiveness and feasibility of the SECO collaboration role model. Specifically, we investigate two research questions:

- *RQ1: Is the expressiveness of the SECO collaboration role model sufficient to address the discussed challenges arising in software ecosystem environments?*
- *RQ2: Is the SECO collaboration role model useful for revealing redundant development efforts?*

We investigate three selected scenarios and discuss them in terms of the requirements derived in Sec. 3. Two of the three investigated scenarios deal with redundant development of features. We present how the introduced SECO collaboration role model can be used to inform developers about double developments and further recommend potentially interesting artifacts.

5.1 Data Collection Methods and Sources

Our main data source, a database containing information about repositories, users, forks, commits, pull requests, issues and other meta-data, has been created in earlier work [33]. Specifically, closed pull requests have been analyzed and the information regarding the reasons of closing them has also been stored in the database. As Marlin is a 3D printing firmware, the scenarios we inspect revolve around the functionality of 3D printers. In particular, we inspect three scenarios as part of the exploratory case study:

1. *AUTO_BED_LEVELING (ABL)*: Shows the evolution through different forks of a feature that computes a bed tilt compensation.

[10] `http://bitbucket.org/modelsteam/2015-marlin`

2. *FAN_CONTROL*: Shows the development of similar features in different forks using different ways to regulate ventilation.
3. *SWITCH*: Shows the development of similar features in different forks using the switching of an operation model of a hardware device.

We used information available in the database to conduct a pre-analysis to select the three scenarios. The analysis used keywords of known features, that the first author was aware of, due to previous experience with Marlin. For each of the selected scenarios, we further queryied the database to better understand the details of that scenario, and manually analyzed the results.

We then performed the following five steps to create the SECO role collaboration model per selected scenario.

S1. Inspect pull requests closed due to double development that are related to the scenario at hand
S2. Select interesting contributions (i.e., successfully merged pull request, pull request not merged due to double development)
S3. Break down contributions in terms of collaboration roles described in Sec. 4
S4. Instantiate the *development and usage*, *storage* and *origin* collaborations to represent inherent knowledge of the SECO
S5. Instantiate the *subscription* and *recommendation* collaborations to represent additional information of the SECO

5.2 Results

We instantiated the role-based collaboration model for each of the three inspected scenarios: *ABL* in Fig. 8, *FAN_CONTROL* in Fig. 9, and *SWITCH* in Fig. 10. Each diagram comprises information on (i) the feature under investigation (left side of the diagrams); (ii) the users[11] involved in the scenario (see diagram centers); (iii) the relations between repositories and users (see right side of the diagrams). For instance, in Fig. 8, the feature *ABL* was developed by the user *lars*, as indicated by the relation between the roles *original feature* and *feature developer*. The developer *lars* is both a maintainer and a contributor to the *lars/Marlin* repository.

There are two features with the same name. However, the one in the bottom left is an improved feature of the original feature developed in *lars/Marlin* fork. Furthermore, we can see that this improved feature was integrated in the *erik/Marlin* repository by user *alex*, who thus becomes a repository contributor to the *erik/Marlin* repository.

Both Fig. 9 and Fig. 10 show the development of similar features. The fan control collaboration depicted in Fig. 9 covers two related features which have been implemented by different developers in different forks. The feature *FAN_CONTROL* by *david* supports reducing the controller fan speed which can reduce unwanted airflow and noise. However, the described functionality is also supported by *robbie*'s feature *EXTRUDER_FAN_CONTROL*. This was discovered while discussing *davids*'s pull request with the repository maintainer of *erik/Marlin*. As

[11] We anonymized the names of the Github users to respect their privacy

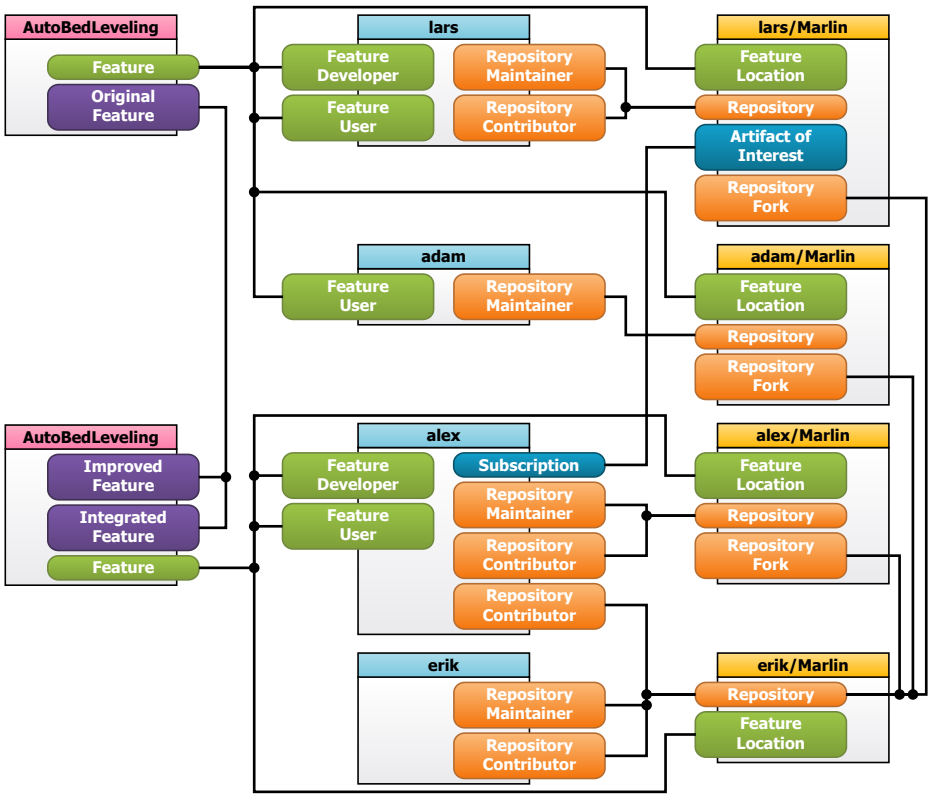

Figure 8. Instance of the collaboration model for the *AUTO_BED_LEVELING* (*ABL*) scenario

robbie's feature *EXTRUDER_FAN_CONTROL* provides more advanced fan control based on extruder hot-end temperatures, the developers decided to merge *robbie*'s feature to the main repository *erik/Marlin* and close *david*'s pull request.

Fig. 10 shows the duplication of the feature *M42* which has been developed by the user *erik* in his fork. The developer *anthony* submitted his new feature *M250* as a pull request. At a later time, the user realized that it was already implemented as *M42*. As the two features are almost identical, *anthony* is made a subscriber to the *M42* feature. Hence, the feature *M42* is recommended to *anthony* even though he was aware that it already existed.

The three scenarios show how the collaboration models use technology-dependent information and present it in a technology-neutral way. Implicit information that exists usually in the repository's issue tracker (if one exists) or developer's memory (e.g., if a feature was integrated) is made explicit by showing relations between features, repositories and users. This improves greatly the overview of existing features and their relation to raise developer awareness.

With respect to RQ1, in this scenario the five collaboration role concepts are sufficient to describe the relations between users, features and repositories

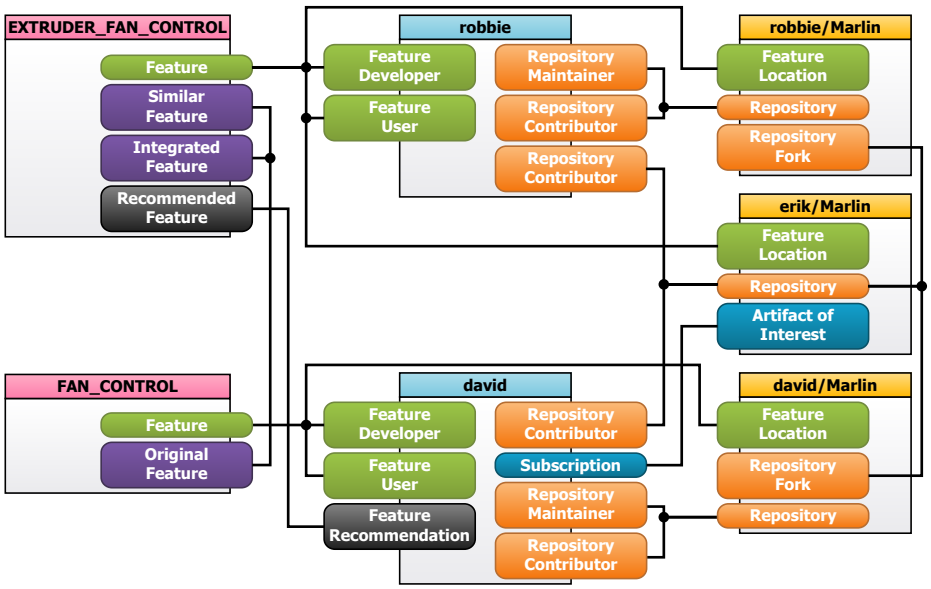

Figure 9. Instance of the collaboration model for the *FAN_CONTROL* scenario.

in a SECO. The formalism is technology independent as required by R1, transforms relevant implicit information in explicit information (R2) such as who is contributing to an artifact and where does that artifact reside, and increases the awareness of developers using the subscription and recommendation collaboration roles as needed by R3. We hypothesize that in other similar scenarios, the collaboration role model would be sufficient, but further studies need to be conducted to understand if the model is general enough.

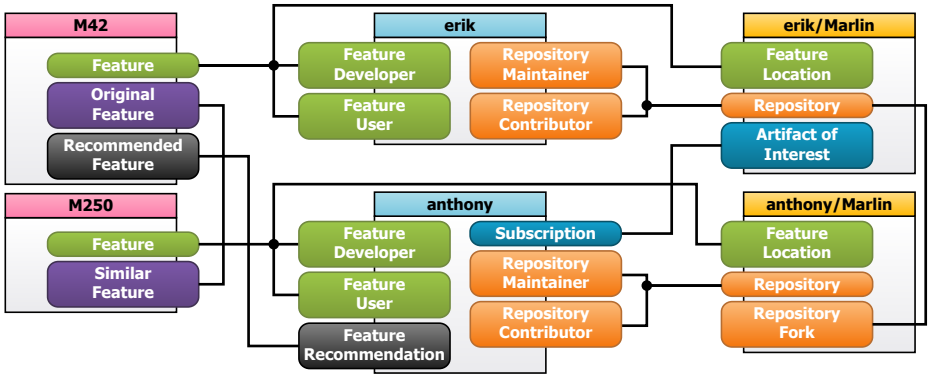

Figure 10. Instance of the collaboration model for the *SWITCH* scenario

Regarding RQ2, using the formalism, we can create models that offer a simple view of artifacts and their relations in the SECO, thus increasing awareness of current developments and limiting concurrent development of features or even of existing ones. Querying the database is more difficult than using the diagrams to inform a user of existing features. One reason is that a user would need to perform several queries to retrieve the same information that is presented in the diagram and to filter the queries. For example, querying for the string 'FAN' results in 194 commits that contain that string in the commit message. In such cases, a trivial task becomes harder to complete. Our formalism allows a more swift usage of the available information, decreasing the burden on developers.

5.3 Discussion

The main difficulty we encountered was to process a large quantity of data to retrieve only parts that we are interested in. The database used in this exploratory case study has a lot of information but it becomes less appealing to use when simple information needs to be retrieved. However, in the case of complex information–which code from what pull request of specific user was integrated in which repository–the database is extremely useful due to its querying capacities. Hence, in the future, our approach might be extended by a similar querying mechanism to limit the amount of information to a specific inquiry.

Regardless of the technology used, to create collaboration models in a SECO we need to mine and use available information from bug trackers, issue systems, version control systems and other meta-data. Collecting such information can be automated to a high degree, for example by using GHTorrent [15], that facilitates querying a MySQL database containing Github meta-data about repositories, users, commits, pull requests, issues and others. The difficulty lies in transforming the data into the collaboration format model, though some parts can be automated, e.g., extracting users, repositories, contributors. Creating roles, discovering features, similar features and integrated features is more laborious and difficult, and requires less trivial computations.

Compositionality as demanded in R4 is principally supported by our formalism through the possibility to map collaborations multiple times (e.g., for different features) with some roles being mapped to the same element, which allows describing complex structures from basic constituents. Furthermore, the collaboration role relations connect cohesive collaborations. However, we still have to devise dedicated tool support, e.g., to semantically zoom in on a particular feature and only show relevant collaborations in order to make the sheer size of the collaboration model manageable.

5.4 Threats to Validity

As with any empirical evaluation, there are a number of threats to validity:

Internal Validity. For our evaluation, we considered only one repository with detailed information from an existing database. However, our method and formalism can be used for other types of repositories (Mercurial, CVS etc.), hosted on

different hosting services (Bitbucket, Github, Sourceforge etc.) as it is technology independent. To ensure correctness of our method and models, the first two authors each executed several scenarios independently of each other and recorded all the steps taken. We then compared the recorded steps, discussed them and agreed on steps S1-S5 presented in Section 5.1. Finally, we cross-validated the results by exchanging the realized scenario diagrams and verifying their correctness.

External Validity. There is a threat in concluding that the formalism is general enough to be applied to any SECO. While we have not run an extensive evaluation using other SECOs, from our experience, the formalism is general enough to model different complex relationships in a SECO. We plan on applying the formalism to Eclipse and also in an industrial SECO that exists within KEBA AG to verify our hypothesis.

6 Related Work

In the following, we describe related work grouped by its main application area.

Software Ecosystems (SECOs). Existing approaches supporting SECO modeling [6,31], do not specifically focus on distributing knowledge about collaborations of contributors and artifacts (e.g., a new feature developed by a specific contributor). For instance, although software supply networks (SSNs) [6] provide a business and management view, they can hardly cover technical development aspects of SECOs. In contrast, the TECMO meta-model [31] provides a technical viewpoint on a SECO and models the variability of a SECO and its evolution in terms of products. However, this rather coarse-grained and product-focused view may not sufficiently support developers in revealing duplicate development of features.

Developer Collaboration Networks. Joblin et al. [19] present an automated approach to capture a view on developer coordination. Their fine-grained approach is based on commit information and source-code structure, mined from version-control systems. Their main goal is to identify developer communities. Another study by Panichella et al. [25] investigates how collaboration links vary and complement each other when they are identified through analyzing data from different kinds of communication channels (i. e., mailing lists, issue trackers, and chat logs) and how revealed collaboration links overlap with relations mined from code changes. Begel et al. [2] conducted a survey on inter-team coordination and found that it is most challenging to find and keep track of activities among the engineers. We provide a formalism that could be used by future research to describe collaborations between developers in SECOs.

Forked Code Bases. An exploratory study by Gousios et al. [16] investigates pull-based software development practices. Specifically, reasons for not merging pull requests are inspected. Results show that 29% of unmerged pull requests are closed because the pull request is no longer relevant, or the feature is currently being implemented in another branch, a new pull request solves the problem better, or the pull request duplicates already available functionality. Another study by Gousios et al. [17] specifically examines managing and integrating contributions in a pull-based development environment. Pull requests are often rejected due

to code quality, but also because newer pull requests already solved the same issue. Duc et al. [12] conducted semi-structured interviews to understand multi-platform development practices. The results show that diverged code bases lead to redundant development effort and to a lack of knowledge of the whole system.

Role-based Feature Management. Muthig and Schroeter [24] present a feature management framework including a software product line information model. The proposed information model comprises a role model which is used to formally model individuals and their access on system resources and operations. The collaboration role model presented in this paper is not concerned with access control or permissions. We focus on describing relationships between features, developers and repositories.

Awareness. Researchers have recognized awareness as an essential aspect of successful collaborative software development. Awareness has been defined as *an understanding of the activities of others providing a context for your own activity* [10]. It has also been stressed that building mental models of others' activities is important for software engineering tasks [9]. A recent study indicates that code reviews can provide additional benefits such as knowledge transfer or increased team awareness [1]. Most of the related work on awareness support for software development has focused on collaborative coding rather than requirements management, project management or design [32]. However, there are attempts focusing on higher levels of abstraction. An example is IBM's Jazz software development environment, which aggregates data to improve awareness of higher-level as well as low-level aspects [14]. Kintab et al. present a framework for recommending experts to developers needing help with a specific code fragment or system component [20]. A ranked list of potential helpers is created based on code similarities and social relationships. In comparison, our work tries to lay the fundamental concepts that allow to use frameworks as this one for a specific goal.

7 Conclusion and Outlook

In this paper, we have introduced a formalism based on role modeling that describes collaborations between contributors and artifacts in a software ecosystem. The formalism is designed to tackle several challenges that currently exist in software ecosystems, such as generating a better overview of the software ecosystem, and transforming implicit knowledge into explicit knowledge to aid developers during the development phase of features. We have conducted an exploratory case study on the Marlin open source software ecosystem and selected three scenarios to apply our formalism. For the inspected scenarios, the five collaborations defined in the formalism were sufficient to model real artifacts, transform implicit knowledge into explicit knowledge, and improve awareness of existing artifacts.

This work is a step forward in understanding how to use existing data to improve the knowledge of developers in the SECO. We plan on applying the formalism to other open source and industrial projects to gain further experience and inspect any shortcomings. Furthermore, we would like to address requirement R4 in more detail and provide dedicated tool support for semantic

zoom on selected elements and their collaborations. Finally, it would also be interesting to explore how to incorporate changes to the SECO appearing as part of software evolution into our modeling notation.

Acknowledgments

This work was partially supported by the Christian Doppler Forschungsgesellschaft, Austria and KEBA AG, Austria. Further, this work was partially supported by the DFG (German Research Foundation) under grant SCHA1635/2-2 and by the European Commission within the project HyVar (grant agreement H2020-644298).

References

1. A. Bacchelli and C. Bird. Expectations, outcomes, and challenges of modern code review. In *Proc. of ICSE'13*, 2013.
2. A. Begel, Y. P. Khoo, and T. Zimmermann. Codebook: Discovering and exploiting relationships in software repositories. In *Proc. of ICSE'10*, 2010.
3. T. Berger, D. Lettner, J. Rubin, P. Grünbacher, A. Silva, M. Becker, M. Chechik, and K. Czarnecki. What is a feature?: a qualitative study of features in industrial software product lines. In *Proc of SPLC'15*, 2015.
4. T. Berger, D. Nair, R. Rublack, J. Atlee, K. Czarnecki, and A. Wąsowski. Three cases of feature-based variability modeling in industry. In J. Dingel, W. Schulte, I. Ramos, S. Abrahão, and E. Insfran, editors, *Model-Driven Engineering Languages and Systems*, volume 8767 of *Lecture Notes in Computer Science*, pages 302–319. Springer International Publishing, 2014.
5. J. Bosch. From software product lines to software ecosystems. In *Proc. of SPLC'09*, 2009.
6. V. Boucharas, S. Jansen, and S. Brinkkemper. Formalizing software ecosystem modeling. In *Proc. of 1st Int'l Workshop on Open Component Ecosystems*, 2009.
7. P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. SEI Series in Software Engineering, Addison-Wesley, 2001.
8. K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Boston, MA, 2000.
9. L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb. Social coding in github: Transparency and collaboration in an open software repository. In *Proc. of CSCW'12*, 2012.
10. P. Dourish and V. Bellotti. Awareness and coordination in shared workspaces. In *Proc. of CSCW'92*, 1992.
11. Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki. An exploratory study of cloning in industrial software product lines. In *Proc. of CSMR'13*, 2013.
12. A. N. Duc, A. Mockus, R. Hackbarth, and J. Palframan. Forking and coordination in multi-platform development: A case study. In *Proc. of ESEM'14*, 2014.
13. P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.
14. R. Frost. Jazz and the eclipse way of collaboration. *Software, IEEE*, 24(6):114–117, 2007.

15. G. Gousios. The ghtorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 233–236, Piscataway, NJ, USA, 2013. IEEE Press.
16. G. Gousios, M. Pinzger, and A. v. Deursen. An exploratory study of the pull-based software development model. In *Proc. of ICSE'14*, 2014.
17. G. Gousios, A. Zaidman, M.-A. Storey, and A. van Deursen. Work practices and challenges in pull-based development: The integrator's perspective. In *Proc. of ICSE'15*, 2015.
18. T. R. Gruber. Toward principles for the design of ontologies used for knowledge sharing. *Int. J. Hum.-Comput. Stud.*, 43(5-6):907–928, 1995.
19. M. Joblin, W. Mauerer, S. Apel, J. Siegmund, and D. Riehle. From developer networks to verified communities: A fine-grained approach. In *Proc. of ICSE'15*, 2015.
20. G. A. Kintab, C. K. Roy, and G. I. McCalla. Recommending software experts using code similarity and social heuristics. In *Proc. of CASCON'14*, 2014.
21. D. Lettner, F. Angerer, P. Grünbacher, and H. Prähofer. Software Evolution in an Industrial Automation Ecosystem: An Exploratory Study. In *Proc. of SEAA'14*, 2014.
22. D. Lettner, F. Angerer, H. Prähofer, and P. Grünbacher. A Case Study on Software Ecosystem Characteristics in Industrial Automation Software. In *Proc. of ICSSP'14*, 2014.
23. D. Lettner and P. Grünbacher. Using feature feeds to improve developer awareness in software ecosystem evolution. In *Proc. of VaMoS'15*, 2015.
24. D. Muthig and J. Schroeter. A framework for role-based feature management in software product line organizations. In *Proc. of SPLC'13*, 2013.
25. S. Panichella, G. Bavota, M. Di Penta, G. Canfora, and G. Antoniol. How developers' collaborations identified from different sources tell us about code changes. In *Proc. of ICSME'14*, 2014.
26. M.-O. Reiser and M. Weber. Multi-level feature trees. *Requirements Engineering*, 12(2):57–75, 2007.
27. D. Riehle and T. R. Gross. Role model based framework design and integration. In *Proc. of OOPSLA '98*, 1998.
28. J. Rubin and M. Chechik. A survey of feature location techniques. In *Domain Engineering, Product Lines, Languages, and Conceptual Models*, pages 29–58. 2013.
29. H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes. Sourcerercc: Scaling code clone detection to big code. *CoRR*, abs/1512.06448, 2015.
30. B. Sarwar, G. Karypis, J. Konstan, and J. Riedl. Item-based collaborative filtering recommendation algorithms. In *Proceedings of the 10th International Conference on World Wide Web*, WWW '01, pages 285–295, New York, NY, USA, 2001. ACM.
31. C. Seidl and U. Aßmann. Towards modeling and analyzing variability in evolving software ecosystems. In *VaMoS*, 2013.
32. B. Sengupta, S. Chandra, and V. Sinha. A research agenda for distributed software development. In *Proc. of ICSE'06*, 2006.
33. S. Stănciulescu, S. Schulze, and A. Wąsowski. Forked and Integrated Variants In An Open-Source Firmware Project. In *Proc. of ICSME'15*, 2015.
34. R. Studer, V. R. Benjamins, and D. Fensel. Knowledge engineering: Principles and methods. *Data Knowl. Eng.*, 25(1-2):161–197, 1998.