Tuning GitHub for SPL development: branching models & repository operations for product engineers

Leticia Montalvillo
University of the Basque Country (UPV/EHU)
ONEKIN Research Group - Facultad de Informática - San Sebastián, Spain
leticia.montalvillo@ehu.es

Oscar Díaz
University of the Basque Country (UPV/EHU)
ONEKIN Research Group - Facultad de Informática - San Sebastián, Spain
oscar.diaz@ehu.es

ABSTRACT

SPLs distinguish between domain engineering (DE) and application engineering (AE). Though each realm has its own lifecycle, they might need to be regularly synchronized to avoid SPL erosion during evolution. This introduces two sync paths: update propagation (from DE to AE) and feedback propagation (from AE to DE). This work looks at how to support sync paths in Version Control Systems (VCSs) using traditional VCS constructs (i.e. merge, branch, fork and pull). In this way, synchronization mismatches can be resolved à la VCS, i.e. highlighting difference between distinct versions of the same artifact. However, this results in a conceptual gap between how propagations are conceived (i.e. update, feedback) and how propagation are realized (i.e. merge, branch, etc.). To close this gap, we propose to enhance existing VCSs with SPL sync paths as first-class operations. As a proof-of-concept, we use Web Augmentation techniques to extend GitHub’s Web pages with this extra functionality. Through a single click, product engineers can now (1) generate product repositories, (2) update propagating newer feature versions, or (3), feedback propagating product customizations amenable to be upgraded as core assets.

Keywords
SPL Evolution, VCS, branching model, Change Propagation

1. INTRODUCTION

Software Product Line (SPL) development is separated into two interrelated processes: (1) domain engineering (DE), where reusable assets and the scope and variability of the system are defined, and (2), application engineering (AE), where products are derived using these assets by resolving variability [16]. Although each realm has its own lifecycle, they need to be in sync to avoid SPL erosion due to evolution [6]. This introduces two sync paths: the update path (from DE to AE ) and the feedback path (from AE to DE) [12]. Update paths serve two scenarios: configuration repair (synchronize products configuration when variability model changes) [4] & product upgrade (where latest versions of reusable assets are propagated to products) [2, 20]. As for the feedback path, it is being proposed to extend the scope of the product line to emerging application engineering requirements [15]. However, synchronization is achieved not between artifacts but artifact versions. This requires propagations to act upon the right version of artifacts. This complexity calls for Version Control Systems (VCSs), not only to assist in managing the large number of artifacts, but also to help in synchronizing the AE and the DE realms. This paper addresses VCSs for SPLs. This involves: (1) a model of what a CoreAsset repository looks like (a.k.a. branching model), (2) a model of what a Product repository looks like, and (3), a set of operations to keep both models in sync.

Traditional VCSs are mainly thought for single-product development. State-of-the-art VCSs such as Git/GitHub, provide the basics but fall short in supporting sync paths between separated repositories (e.g. a CoreAsset repository and a Product repository). All GitHub offers is the fork link. However, forking (i.e. cloning) is not how products are derived. Indeed, products are built from a subset of core assets while forking would entail copying the entire CoreAsset repository. Likewise, GitHub’s pull request is also thought for synchronizing a whole repository, hence lacking a more piecemeal synchronization.

Different authors address VCS limitations for SPLs [2, 13, 14, 20]. Although the need for sync paths is recognized, no “algorithmically reproducible” details are given about how this is exactly achieved (e.g. no branching model is provided). This is unfortunate since the adequacy of branching models very much depend on the processes to be supported. This lack of formalization hinders the discussion about SPL-tuned branching models as well as the emergence of SPL sync good practices. In this setting, this work’s main contributions rest on

1. a repository architecture, which distinguishes between the CoreAsset repository, where domain engineering takes place, and Product repositories, where product engineering occurs. This provides the data structure (a.k.a. branching model) in which sync actions operate (Section 4).

2. the operational semantics for sync actions. Synchronization happens upon artifact versions. The previous branching model permits sync operations to be expressed in terms of the basic VCS constructs. This in turn implies that eventual mismatches that rise during
We start by motivating the problem.

2. PRODUCT DERIVATION: ILLUSTRATING THE CHALLENGE

We stick to the generic process for product derivation described in [7]. Deelstra et al. distinguish between the initial synchronization are resolved à la VCS, i.e. highlighting diff-erence between distinct versions of the same artifact (traditionally, using the diff option in VCSs). Therefore, we do not aim at automatic sync. Our aim is much more humble: tap into VCS popular mechanisms for SPL engineers to achieve sync in a way similar to what they do for single products (Section 5). However, this results in a conceptual gap between how sync paths are conceived, and how they are realized down into branching and merging. To close this gap, we propose leveraging VCSs with SPL sync operations.

3. PROPOSALS FOR VCS IN SPLS

VCSs are a cornerstone for distributed, collaborative development. And the iteration phase. In the initial phase, a first configuration is created from the core assets. In the iteration phase, the initial configuration is modified in a number of subsequent iterations until the product sufficiently implements the imposed requirements. Unlike Configurable Product Lines (CPLs) where product derivation is limited to the configuration expression, SPLs do not achieve such degree of reuse effectiveness, and require core assets to be customized during product derivation. This makes SPLs more difficult to manage that CPLs since they might potentially involve a larger number of artifacts (not just core assets, but product specific artifacts as well), handled by different teams, and following different life-cycles. This Section illustrates the complexities of product derivation through an example.

Consider VODPlayer-PL, a SPL for video playing software. VODPlayer-PL is implemented in Java using Feature-Oriented Programming [3] and includes ten core assets at its initial version (see Table 1). Products are derived from these core assets in accordance with a feature diagram (not included here). Both core assets and products are not standing still but evolve. And this introduces the challenge: synchronize the pace at which core assets and products are released, considering that those artifacts might well be governed by different teams with distinct priorities. Figure 1 depicts this matter. Core assets are arrayed down the left-hand side (e.g. VODPlayer, ChooseMovie). Each asset undergoes evolutionary change; its evolutionary trajectory extends to the right. The bottom shows the products in the SPL. Each product goes through various phases, such as alpha release, beta release, and General Available (GA) release. Across the top are several baselines. A baseline contains a set of assets, each at a given version, that work together and are used to build products. Besides re-use of core assets, Figure 1 also highlights possible sync paths (depicted as dotted lines): upgrades of ChooseMovie are percolated to ProductA whereas a customization conducted for ProductA is promoted as the core asset AutomaticReplay. The question is how to facilitate this process using existing VCSs.

Table 1: VODPlayer-PL Core Assets

<table>
<thead>
<tr>
<th>Core Asset ID</th>
<th>Core Asset Name</th>
<th>Core Asset Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CA1</td>
<td>VODPlayer</td>
<td>Provides the PL architec-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ture and basic functionality</td>
</tr>
<tr>
<td>CA2</td>
<td>ChooseMovie</td>
<td>Allow users to choose a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>server to connect to</td>
</tr>
<tr>
<td>CA3</td>
<td>ViewMovie</td>
<td>Allow users to pause the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>movie they have already</td>
</tr>
<tr>
<td></td>
<td></td>
<td>selected from the list</td>
</tr>
<tr>
<td>CA4</td>
<td>PlayMovie</td>
<td>Allow users to start the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>movie they have already</td>
</tr>
<tr>
<td></td>
<td></td>
<td>selected from the list</td>
</tr>
<tr>
<td>CA5</td>
<td>StopMovie</td>
<td>Allow users to stop the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>movie they are currently</td>
</tr>
<tr>
<td></td>
<td></td>
<td>watching</td>
</tr>
<tr>
<td>CA6</td>
<td>PauseMovie</td>
<td>Allow users to pause the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>movie they are currently</td>
</tr>
<tr>
<td></td>
<td></td>
<td>watching</td>
</tr>
<tr>
<td>CA7</td>
<td>Quit</td>
<td>Allow users to quit from</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the player</td>
</tr>
<tr>
<td>CA8</td>
<td>ChangeMoviesServer</td>
<td>Provide a manual-like</td>
</tr>
<tr>
<td></td>
<td></td>
<td>approach to change the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>server they are connected</td>
</tr>
<tr>
<td></td>
<td></td>
<td>to</td>
</tr>
<tr>
<td>CA9</td>
<td>ManualChange</td>
<td>Provides the PL architecture and basic functionality to run the player</td>
</tr>
<tr>
<td>CA10</td>
<td>AssistedChange</td>
<td>Assists users on change server to connect to by loading a list of available servers</td>
</tr>
</tbody>
</table>

Figure 1: The SPL synchronization challenge (adapted from [11])
development. SPLs promote collaborative development through reuse. Traditionally, collaborative development applies to different users working on the same piece of code. By contrast, SPLs set two realms (i.e., domain engineering & application engineering), where collaboration goes along the sync paths. The fact of being two separated realms makes it even more important to track who made which changes, and when they were made. Provenance of the contributions can turn key when, like in the SPL case, development might be distributed among different business units with their own budgets and responsibilities [5].

VCSs are specifically designed to keep track of who did what. Broadly, VCSs support “revisions”, i.e. a line of development (a.k.a baseline or trunk) with branches off of this. Disparate efforts are reunited by merging branches. In addition, repositories can be forked whereby a whole repository is cloned in a separated space. Unlike a branch, a fork is independent from the original repository. If the original repository is deleted, the fork remains. This space can be merged back through a pull request. The fork & pull model reduces the amount of friction for new contributors. This makes this model popular among open source projects because it allows people to work independently without upfront coordination. Notice that VCSs do not dictate the file structure nor when to branch or merge. This is part of the branching model. Approaches to branching models very much depend on the dependencies to be preserved through the VCS.

Back to SPLs, approaches broadly distinguish two main ways to face SPL development: extractive (departing from existing products) and pro-active (departing from scratch). Next paragraphs delve into proposed solution for these two scenarios.

**Extractive Scenario.** Here, a new product is obtained through clone&own from existing products. Branching model wise, two models are proposed:

- **branch-per-product-customer** [19]: a main branch holds the code shared by all products. Product variants come as branches off the main branch, one per customer, where customer-specific modifications are performed.
- **branch-per-product-functionality** [3]: there is one main branch per functionality that products may exhibit. Product variants are obtained by merging functionality branches.

As the authors themselves recognize, these approaches do not scale well, and do not account for effective reuse of code, since these branching model encourage the development of variants of products, and not reusable core assets. Along the clone & own practice, Fisher et al. [9] propose product variants to be obtained through cherry-picking feature snippets from distinct products. Closely related, Rubin et al. [17] considers the VCS implications to support propagation between product variants so that changes into a product can ripple to derived product variants.

**Pro-active Scenario.** Here, a distinction is made between core assets (thought for reuse) and products (thought for use). VCS wise, two approaches are proposed:

- **single repository.** Here, core assets and products are kept in the same repository. Traceability between core assets and derived products is achieved through branching [10]. On the downside, branches hold both core assets and products. Sharing the same space might be a problem if these different kinds of artifacts are handled by different teams along distinct life-cycles. Scalability might also be an issue. Here, Anastasopoulos [2] presents a tool on top of Subversion, which keeps SPL artifacts identified (where in the VCS). Engineers can perform activities related to evolution control including propagation of changes. Update propagation is performed by AE over a single core asset instance. Feedback propagation is conducted by DE over a single core asset. This operation, merges all the core assets instances into the core asset. This seems inconvenient since it assumes that all the instances have changes that need to be promoted to DE.

- **separate repositories.** Here, core assets and products are kept in different repositories, tied up through a derivation link [20]. Unlike Anastasopoulos, Thao et al. [20] do not consider reusing existing VCSs. Instead, they build a home-made one, which is capable of establishing dependencies between products and core assets. Whenever a product is derived, a new branch is automatically created in the core asset repository. This branch references the product repository main branch, and serves for change propagation (for both parties). If DE changes something on it, this is an update propagation. Hence, update propagation looks like permitting DE to override assets in Product repositories, which seems risky. Scalability might be an issue as well.

Our work follows Anastasopoulos in so far as tapping into existing VCS tools (in our case, GitHub). Like Thao et al., we also advocate for two types of repositories: CoreAsset repositories and Product repositories. Figure 1 depicts our sample SPL arranged along this repository architecture. Each repository is a separated installation, hence, managed by its own team. However, the SPL’s repositories are not isolated but conform an ecosystem tightened together through sync paths (depicted through dotted lines in Figure 1). Unfortunately, inter-repository operations are so far limited to fork & pull model: a fork clones a whole repository into a brand new one, which evolves independently until it might be merged back through a pull. This fits well for open software projects but fall shorts for SPLs. Here, reuse is not based on whole cloning but derivation: cherry-picking core-asset and next, customization. On this premise, we introduce the derive & update & feedback model which rests in the namesake operations. Unlike fork, derivation does not involve a whole clone but a cherry-picking selection of core assets. In the same vein, and in contrast with GitHub’s pull, update & feedback govern a piecemeal synchronization between Product repositories and its source CoreAsset repository. Next section delves into the branching model.

### 4. BRANCHING MODELS

VCSs support “revisions”, i.e. a line of development (the baseline or trunk) with branches off of this, forming a directed tree, visualized as one or more parallel lines of development (the “mainlines” of the branches) branching off a baseline (see Figure 2). The question is how to mimic the modus operandi of SPL development in terms of “parallel lines of development”, i.e. setting the branching model.
Since core assets and products are not born equal (i.e., products are derived from core assets while core assets might be obtained from scratch or extractively from existing products), we believe they can be better served by distinct branching models.

4.1 A Branching Model For Core Assets

For single-product development, a popular approach is branch-per-purpose [21]. This strategy recommends different branch types per task type. A popular Git branching model includes the following branching types [1]: *master*, *develop*, *digression* and *release*. For understanding sake, we stick to this terminology (see Figure 2 top):

- **Master** branch is a long-lived branch aimed at core assets release management. Each commit under master holds a stable release of core assets that work together (e.g., Baseline 1.0 holds core assets CA1 to CA10). This branch becomes essential for application engineers, and it is the cornerstone for product derivation.²

- **Develop** branch is a long-lived branch which serves as the mainline of development for core assets.

- **Digression** branches are short-lived branches that serve to assist on parallel development of core assets, to create new core assets or adapt existing ones (e.g., updateCA2 branch enhances CA2 core asset).

- **Release** branches are short-lived branches used to prepare the next release for the core asset baseline, before merging it to master (e.g., release2.0 branch).

This approach accounts for a parallel and consistent development of core assets under a single join development (by means of Develop and Digression branches). In addition, products can rely on a consistent release of core assets (baseline release in Master branch). This model embraces a release strategy whereby all core assets are made available all together on regular intervals. This may introduce a latency for application engineers. That is, even if a core asset implementation is ready for production, it cannot be released until other core assets are also ready to be in the next baseline release. This latency might lead product engineers to “clone and own” the best-fitting asset and adapt it to their needs [14]. Finding the right release pace is up to each SPL organization.

4.2 A Branching Model For Product Repositories

Unlike core assets, products are derived from other artifacts, i.e. the core assets. This states a dependency between products and core assets. Better said, between a product and the core assets used for its derivation. Notice, this dependency is not with all core assets but just with those assets that participate in the initial product configuration. This dependency might involve for product engineers, first, to be aware of upgrades for the core assets at hand (update propagation), and second, being able to communicate product customization which might be amenable to be turned into SPL’s core asset (feedback propagation). This subsec-

²That the core asset code is fully stable might be less an issue if development speed counts. Releasing not-fully tested features might make sense in these scenarios which we have not considered here.
tion introduces a branching model conceived for facilitating these propagations. By “facilitating” we mean to be able to express those propagations in terms of the basic VCS constructs (i.e. branch, merge, fork, pull). The final aim is to spot mismatches risen during synchronization à la VCS, i.e. highlighting diff-erences between distinct versions of the same artifact. In this way, SPL engineers handle sync in a very similar way to what they are used to for single products.

Our branching model for Product repositories rests on seven branch types to account for three purposes: development, delivery and propagation. For illustration purposes, we resort to our running example (see Figure 1) but now looking inside the repositories (see Figure 2).

For development: BigBang, Develop & Custom branches. BigBang is a long-lived branch, which keeps localized the baseline from which the product was derived. For instance, if a product wants to be derived from the CoreAsset Baseline 1.0, a BigBang branch would point to a commit exactly the same as baseline 1.0 (same commit object, although in different repositories). This branch remains untouched, during the repository life time. This is so, to enable feedback propagation process (see later). On the other hand, Develop and Custom branches embrace parallel development for product assets. Develop branch is a long-lived branch which holds the mainline of product asset development. Custom branches, obtained off Develop branches, are used for product specifics: core assets can be adapted while brand new assets can be introduced. When a customization is considered finished, Custom branches are merged back into Develop branch. Although good practices would advocate to delete Custom branches after merging them back to the mainline, our model maintains these branches alive for feedback purposes. Figure 2 (bottom) shows the case where a Product repository is derived from Baseline1.0, instantiating core assets CA1 to CA7. Additionally, CA1 is customized to CA1’, hence giving rise to a Custom branch.

For delivery: Release & Master branches. Upon a consistent set of product assets under a Develop branch, Release branches are created for obtaining an executable product with the help of assembly tools. When this product is ready for GA Release, it would be merged to the Master branch and tagged accordingly. Master is a long-lived branch containing product releases ready to be delivered to customers. Figure 2, shows the case where productA alpha release consists of the initially derived core assets plus CA1’ customization. The beta release includes an additional enhancement on CA2’. Finally, the GA Release also comprises a customization for CA4’.

For propagation: Update & FeedBack branches. Parallel development involves resolving eventual conflicts when acting upon the same artifact. VCSs offer diff tools that highlight differences in code lines to easily spot mismatches. For these tools to be effective, the artifacts to be compared should correspond to versions of the same artifact. However, when an artifact is composed with other artifacts, the result can no longer be qualified as “a version” of the composing artifacts. Hence, applying diff between a core asset and a product would be of limited use since the code of the core asset might be tangled and polluted with code that is not related with the core asset as such. This calls for Product repositories to keep an independent line of branching with untouched core assets. This is the goal of Update branches: holding the product’s core assets separated from the product mainline (i.e. develop branch). Upon a new baseline release in the CoreAsset repository, product engineers might request an update propagation and easily spot differences using diff (see later).

Back to our example in Figure 2, domain engineers have been busy yielding Baseline 2.0 where CA2 is leveraged to automatically play a movie when the user selects it from a movie list (CA8 and CA9 have also been adapted). At time t3, application engineers conduct an UpdatePropagation upon Baseline 2.0. Should this upper version be integrated? The decision is twofold. First, product engineers diff-erentiate what’s new w.r.t. to previous version (i.e. diff(CA2, CA2’)). If satisfied, next they assess the impact of the new version of CA2 with respect to the product as such. This implies a merge with a Develop branch (see Figure 2). This accounts for a diff-driven stepwise decision that might help spotting potential mismatches between how CA2 evolves (in the domain realm) and how CA2 was customized (in the product realm).

Finally, Feedback branches support promotion of meaningful product customizations into core assets. By meaningful is meant a customization that makes sense as a unit. This might imply collecting code scattered throughout several Custom branches. The feedback process is twofold (see Figure 2). First, a Feedback branch is created to differentiate the customization code from the code in the original core assets. To isolate the customization code (i.e. avoiding mixing it up with other functionality), we cherry-pick those changes from the Custom branch at hand 3. Back to the example, CA4 was customized to automatically re-play a movie after finished. At time t4, application engineers conduct feedback propagation. First, they need to pinpoint the Custom branches at hand (e.g., customCA4 branch). Next, changes of customCA4 are cherry-picked and merged into a Feedback branch. Hence, feedbackCA4 branch only contains those changes for customCA4 (i.e., CA4’). When domain engineers handle this feedback request, a diff(develop:coreAssets, feedback:feedbackCA4) will highlight only changes for the new functionality (i.e., CA4). Domain engineers can now decide to stick with CA4 or rather, open a new core asset (i.e., CA11) where to generalize the product customization to the whole SPL.

5. SPL Sync Operations as First-Class Constructs in VCSS

Previous section introduces branching models for Product-Fork, UpdatePropagation and FeedbackPropagation to be expressed in terms of VCS primitive operations (i.e. fork, branch, merge). For instance, a productFork involves both a fork and a branch: a fork upon the CoreAsset repository which creates a BigBang branch; next, BigBang is branched into a Develop branch where only the required core artifacts are kept. Likewise, UpdatePropagation and FeedbackPropagation can also be expressed in terms of these VCS primitives. However, this introduces a gap between how operations are conceived, and how operations are realized, with the consequent costs associated. Our aim is to leverage exist-

---

3VCS’s cherry-pick operation takes the changes introduced in a commit, and tries to reapply it on the current branch. This is useful when there is a number of commits on a branch, and only one of them is to be integrated into another branch.
Algorithm 1 provides the details:

5.1 Product Fork

ProductFork takes a CoreRepository as input, and delivers a ProductRepository, along a given configuration. Namely:

\[
\text{PRODUCTFORK} \quad \text{(UserAccount:UserAccount, Repository:CoreRepo, String[]: configuration)} :: \text{Repository:ProductRepo}
\]

where UserAccount stands for the application engineer’s GitHub user account; CoreRepo stands for the CoreAssetRepository from which a Product repository will be derived; and configuration holds a list of core asset identifiers. ProductRepo stands for the newly initialized Product repository. Figure 3 describes the new Product repository. Algorithm 1 provides the details:

1. Perform a fork operation over CoreRepo (line 2). Now, UserAccount owns a copy of CoreRepo repository. At this point, ProductRepo and CoreRepo are identical (same branches, commits, tags, repository details, etc.), except for ProductRepo holds a fork link to CoreRepo.

2. Rename productRepo with pattern `<SPL_name> <product> <date>`, and change its description to state that productRepo is actually a product derived from a core repository (lines 3-4).

3. Adapt productRepo to the product branching model introduced in section 4.2 (lines 5-19), namely:

   (a) First, all branches that ProductRepo holds, are deleted (lines 5-7), except for master : baseline branch, which in ProductRepository turns into bigBang: kickOff. As there is no way to rename a branch in Git, the way to simulate this operation is to, first create a new branch for bigBang: kickOff (lines 8-9), and then delete master: baseline (line 12). BigBang: kickOff keeps now all core assets from CoreRepo baseline (i.e. CoreRepository). DELETEBRANCHBYNAME operation performs an HTTP request to delete branches of GitHub repositories.

   (b) Second, develop: productAssets branch is created off bigBang: kickOff (line 10). GETBRANCHBYNAME operation is accessed the GitHub API to obtain a branch by its name from a given repository. SETDEFAULTBRANCH operation performs a HTTP request to set as default branch of a GitHub repository.

   (c) Third, those core assets not referred in configuration are deleted (lines 13-16). DELETEFOLDER operation performs HTTP requests to delete all files from a given folder. At this point develop: productAssets branch only holds the core assets needed to exhibit by the product (Figure 3, PRODUCTASSETS).

   (d) Finally, update: updates branch is created off develop: productAssets (line 17), and initialized with the Product.config file. This file holds the sha identifier of the CoreRepo’s baseline version from which ProductRepo is derived (line 18-19). At this point, update: updates holds original reusable core assets versions (Figure 3, ORIGINALCORESV1).

5.1.1 Leveraging GitHub with ProductFork

Product derivation is performed upon CoreAsset repositories. Figure 4 depicts VODPlayer-CoreAssets repository, available at https://github.com/letimome/VODPlayer-CoreAssets. Readers can click on this URL. However, this will only recover a plain GitHub HTML page. Enhancing GitHub pages with SPL-specific VCS operations is achieved through the GitLine browser extension. GitLine makes on-the-fly changes to GitHub pages to account for ProductFork, UpdatePropagation and FeedbackPropagation. Using Web Augmentation techniques [8], GitLine adds buttons to enact those operations, i.e. repositories are accessed through GitHub’s APIs, and extra iFrames are popped-up, should additional interactions with the user be needed. GitLine

ха” is GitHub name for unique hash identifier for an artifact, let this be a folder, a file or a commit object.
Algorithm 1 Product Fork

```java
ProductFork(userAccount, coreRepo, configuration): Repository
    repository productRepo = Fork(userAccount, coreRepo);
    productRepo.name = split(coreRepo.name, '-').(0)+'/Product'+currentDate();
    productRepo.description = 'A product derived from ' + coreRepo.name;
    for each branch in productRepo.branches:
      if (branch.name == 'master:baseline'):
        DeleteBranchByBranchName(userAccount, productRepo, branch.name)
    Branch master = GetBranchByBranchName(userAccount, productRepo, 'master:baseline')
    Branch bigBang = new Branch(userAccount, productRepo, master, 'bigBang:kickOff')
    Branch develop = new Branch(userAccount, productRepo, bigBang, 'develop:productAssets')
    SetDefaultBranch(userAccount, productRepo, develop)
    DeleteBranchByBranchName(userAccount, productRepo, 'master:baseline')
    File master:baseline = develop.commit.folders
    for each coreAsset in CRepBaseline:
      DeleteFolder(userAccount, productRepo, develop, coreAsset)
    Branch update = new Branch(userAccount, productRepo, develop, 'update:updates')
    File productConfig = new File(userAccount, productRepo, 'product.config', bigBang, commit.sha)
    Commit(userAccount, productRepo, update, productConfig, 'Create config file')
```

Algorithm 1 Product Fork

has been proven for Firefox 37.0, and its available for download at http://letimome.github.io/GitLine/. Note that GitLine needs to be locally installed in each browser from where the SPL repository is to be accessed. This subsection focuses on ProductFork. Drop-like icons are used to highlight certain facts. Double-lined drops denote GitLine layered content.

Figure 4 depicts VODPlayer-Pl CoreAsset repository. Drop A points to the owner and repository name: letsimome and VODPlayer-CoreAssets, respectively. Drop B points to the current branch. Drop C points to the core assets. On top of this rendering, GitLine layers additional content: a new button (drop D). On clicking, a panel shows up which delivers an iframe which holds the result of invoking a web-accessible feature configurator: S.P.L.O.T. [18] (drop E). The panel is automatically generated from the VODPlayer feature model which, in the current implementation, needs to be previously loaded at S.P.L.O.T. Users are now guided by S.P.L.O.T to set up the configuration (in the screenshot core assets CA1 to CA7 are selected). Once the configuration is over, the ProductFork algorithm resorts to GitHub’s APIs to automatically create a GitRepository repository. Its name follows the pattern: <SPL_name>./<product>./<date> (e.g. VODPlayer-Product-05ABR2015). This repository is already initialized with a BigBang branch, Update branch and a Develop branch (Figure 3). The latter holds the selected core assets. Now, application engineers are ready to start.

5.2 Update Propagation

UpdatePropagation takes a Product repository as input, and creates a new version for the Update branch. Namely:

```java
UpdatePropagation(UserAccount: userAccount, Repository:productRepo, Repository:coreRepo) :
```

where userAccount stands for the application engineer’s GitHub user account; productRepo denotes the hosting ProductRepository; and coreRepo corresponds to the CoreAsset repository from which productRepo was derived. The precondition to trigger the operation is: there is a new baseline version in coreRepo whose changes have not been yet propagated to productRepo. This is assessed by reading productRepo product.config file under update:updates branch, which holds the sha identifier to the coreRepo baseline to which productRepo is currently synchronized. If the sha at product.config differs from the one at coreRepo’s master:baseline, it means that productRepo is unsynchronized with coreRepo, and thus, update propagation can be enacted. Figure 5 describes Product repository branching structure before and after the operation. Algorithm 2 describes the operational semantics:

1. Get the latest baseline version available from coreRepo, and bring to productRepo’s update branch the newest versions of those core assets that productRepo is reusing (lines 2-10).
   (a) Specifically, for all those core assets versions productRepo is currently using (i.e., ORIGINAL CORES -V1), check if there is a newer reusable core asset version at coreRepo (lines 6-8).
   (b) If there is a newer version, get it and commit the new version of the asset (i.e., ORIGINAL CORES -V2) to update:updates branch (line 9). As GitHub web site only allows to commit a single file at a time, we developed CommitFolder HTTP operation which given a folder, all files contained inside are committed iteratively. At this point, productRepo holds new versions of reusable core assets under update:updates branch (i.e., ORIGINAL CORES -V2).

2. Update product.config file to indicate that productRepo is now in sync with productRepo (lines 11-13). First the file is obtained by means of getFileByBranchName operation, which is a HTTP request to get a file from a GitHub repository (line 11). Afterwards, file content is updated with the sha identifier of coreRepo last baseline version (line 12), and committed to update:updates branch (line 13).

3. Finally, a pull request is enacted to notify application engineers about the new changes pulled from the CoreAsset repository (lines 14-15). The pull request requests to merge update: updates branch into develop: productAssets branch. At this point application engineers can reason about the impact of these updates have
Algorithm 2 Update Propagation

1. Branch update=GetBranchByName(userAccount, productRepo, ‘update:updates’)
2. Branch coreBaseline=GetBranchByName(userAccount, coreRepo, ‘master:baseline’)
3. Folder originalCoresV1=update.commit.folders
4. Folder originalCoresV2= null
5. for each coreAsset in originalCoresV1 do{
6. originalCoresV2= GetFolderByName(userAccount, coreRepo, coreBaseline, coreAsset.name)
7. if (coreAsset.sha<>originalCoresV2.sha)
8. CommitFolder(userAccount, productRepo, update, originalCoresV2, ‘new update for core asset:’+originalCoresV2.name)
9. }
10. File productConfig=GetFileByName(userAccount, productRepo, update,"product.config")
11. productConfig.content=coreBaseline.commit.sha
12. Commit(userAccount, productRepo, update, productConfig, ‘product synched to baseline’+coreBaseline.commit.sha)
13. Branch develop=GetBranchByName(userAccount, productRepo, ‘develop:productAssets’)
14. CreatePullRequest(userAccount, productRepo, productRepo, develop, update, commit.comment)

Figure 5: Update Propagation involves 1 commit for each core asset updated core asset & 1 pull request into the product assets by popping up the diff panel (see later).

5.2.1 Leveraging GitHub with UpdatePropagation

Update propagation is performed by application engineers upon a Product repository. Figure 7 depicts VODPlayer-Product-05ABR2015, i.e. the Product repository obtained in the previous sub-section, available at https://github.com/lemome88/VODPlayer-Product-05ABR2015. Let’s assume that core assets evolve until Baseline 2.0 (time frame t1-t3) where a new version of CA2 (i.e. ChooseMovie) is available. During the same timeframe, product engineers customized CA1 into CA1’. At this time, application engineers perform updatePropagation. Figure 7(left) depicts this scenario. Drop B points to the current branch. Drop A points to the UpdatePropagation button. On clicking, a pop-up displays the summary of changes to be pulled (drop C): a list of rows with the name of the updated core asset (e.g. ChooseMovie), and a link to the Core-Asset-repository’s commits describing those changes (“New commits”). Following these links brings product engineers to the Core Asset realm by opening a new browser tab, where the ChooseMovie asset evolution is shown in a diff panel (not shown in the Figure), so that product engineers can make an informed decision about whether to pull these changes back to the Product repository. If so decided, developers go back to the Product repository (Figure 7(left), and click the Yes button (drop D). The ChooseMovie newer version is pushed to the Update branch (e.g. update:updates). Application engineers are notified through a new pull request (drop E) to merge update:updates into develop:productAssets. Developers can now open the pull request to retrieve the changes (drop F). A new page shows up with the diff-erences: diff(‘develop:productAssets, update:updates’). If changes are accepted, application engineers merge the branches. Otherwise, the pull request is closed, and the Product repository sticks with the old asset versions.

5.3 Leveraging GitHub with FeedbackPropagation

FeedbackPropagation takes a Product repository as input, and creates a new version for the Feedback branch. Namely:

FeedbackPropagation(UserAccount:userAccount, Repository:coreRepo, Repository:productRepo, Branch:kickOff, Branch[]:customizations, String: feedbackBranchName)::PullRequest

where userAccount stands for the application engineer’s GitHub user account; coreRepo stands for the CoreAsset repository; customizations correspond to the set of branches that keep the product specific changes that want to be propagated back to the CoreAsset repository; finally, feedbackBranchName refers to the name for the feedback branch to be created. Figure 8 describes Product repository branching structure before and after the operation. Algorithm 3 provides the details:

1. Create a Feedback branch, labeled feedbackBranchName (i.e., newFeedback), off bigBang:kickOff (lines 2-3).
2. Build the meaningful customization based on existing customizations branches (lines 4-8). This requires, for each custom branch in customizations (Figure 8, C1 and C3), to cherry-pick the changes that each of the custom branch introduces (i.e., r1,r2 for C1) and to commit them into newFeedback branch (i.e., r5). As GitHub does not provide cherry picking operation, we needed to develop it for GitHub repositories.

(a) This, requires first to identify the assets that a given branch (e.g. custom branch) has changed. GetChangesFromBranch is a HTTP operation which returns all the artifacts that a given branch has changed, arranged in a tree structure.
Figure 6: Leveraging GitHub with \textit{FeedbackPropagation}: enacting (left) and outcome (right).

Figure 7: Leveraging GitHub with \textit{UpdatePropagation}: enacting (left) and outcome (right).

\begin{algorithm}
\caption{FeedbackPropagation}
\begin{algorithmic}
\State \textbf{FeedbackPropagation}(\textbf{UserAccount:} userAccount, \textbf{Repository:} coreRepo, \textbf{Repository:} productRepo, \textbf{Branch:} customizations, \textbf{String:} feedbackBranchName)
\State \textbf{Branch} bigBang = \textbf{GetBranchByName}(userAccount, productRepo, "bigBang:kickOff")
\State \textbf{Branch} newFeedback = \textbf{new Branch}(userAccount, productRepo, bigBang, feedbackBranchName)
\For {each \textbf{custom} in customizations}
\State \textbf{Folder} customizedAssets = \textbf{GetChangesFromBranch}(userAccount, productRepo, custom)
\For {each \textbf{custAsset} in customizedAssets}
\State \textbf{CommitFolder}(userAccount, productRepo, newFeedback, custAsset, 'customized asset:'+custAsset.name)
\EndFor
\EndFor
\State \textbf{Branch} develop = \textbf{GetBranchByName}(userAccount, coreRepo, 'develop:coreAssets')
\State \textbf{CreatePullRequest}(userAccount, coreRepo, productRepo, develop, newFeedback, newFeedback.commit.comment)
\end{algorithmic}
\end{algorithm}
Figure 8: Feedback Propagation involves 1 branch & 1 commit for each Custom branch involved & 1 pull_request

(b) Then, all the identified assets are committed into newFeedback branch.

3. When all customizations have been merged into newFeedback branch, a pull request is created in coreRepo, requesting to merge ProductRepo’s newFeedback branch into coreRepo develop branch (lines 9-10).

5.3.1 Leveraging GitHub with FeedbackPropagation Feedback propagation is performed over a Product repository. Figure 6 (left) depicts VODPlayer-Product-05ABR2015 repository at time t3: a custom branch (i.e., customCA4) was created for CA4 (i.e. PlayMovie). Meanwhile, VODPlayer-CoreAsset repository also committed some changes. At this point, application engineers want to promote changes done in customCA4 (i.e. new version for CA4). Figure 6(left) depicts this scenario. Drop B points to the current branch. Drop A points to the new FeedbackPropagation button. On clicking, a pop-up lists all Custom branches that the Product repository holds (drop C). Users can now select the desired customization (e.g. customCA4 branch), and press the Yes button (drop D). This triggers the feedback propagation algorithm. The scenes, a new Feedback branch is created (i.e. feedbackCA4), and the CoreAsset repository receives a pull request coming from the Product repository (drop E in Figure 6(right)). When this request is opened, domain engineers are invited to merge the newly created VODPlayer-Product’s Feedback branch (i.e., feedback:customCA4) into VODPlayer-CoreAssets’ Develop branch (i.e., develop: coreAssets). Drop F points to the diff view of the changes proposed by this pull request. At this point, domain engineers should decide whether the customization is useful to the whole product line. If so, domain engineers would need to refactor the customized core assets. This might require to create a new Digression branch (e.g. newCA11 branch in Figure 2).

6. CONCLUSION

This work considers a SPL scenario where core assets and products evolve along different life-cycles but get synchronized through propagation events. We introduce a branching model that permits to capture these sync paths in terms of VCS standard operations. Next, so-described processes are delivered as first-class constructs on top of an existing VCS, i.e. Git/GitHub. This permits reducing “the accidental complexity” that goes with supporting sync paths while freeing up developers for focusing on “the essential complexity”, i.e. attuning and refactoring code coming from different developers. Tested for a FOP composer, the approach is valid as long as dedicated core assets for dedicated functionalities are involved. Next work includes to extend composition options as well as facing scalability issues (i.e. SPL with large number of features and products). Our hope is that by delivering GitLine to the community, sync-path good practices emerge. This work is an attempt to make these practices explicit ... and available.

7. REFERENCES