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

SPLC 2015, July 20 - 24, 2015, Nashville, TN, USA
©2015 ACM. ISBN: 978-1-4503-3613-0/15/07 ...$15.00.
DOI: http://dx.doi.org/10.1145/2791060.2791083.

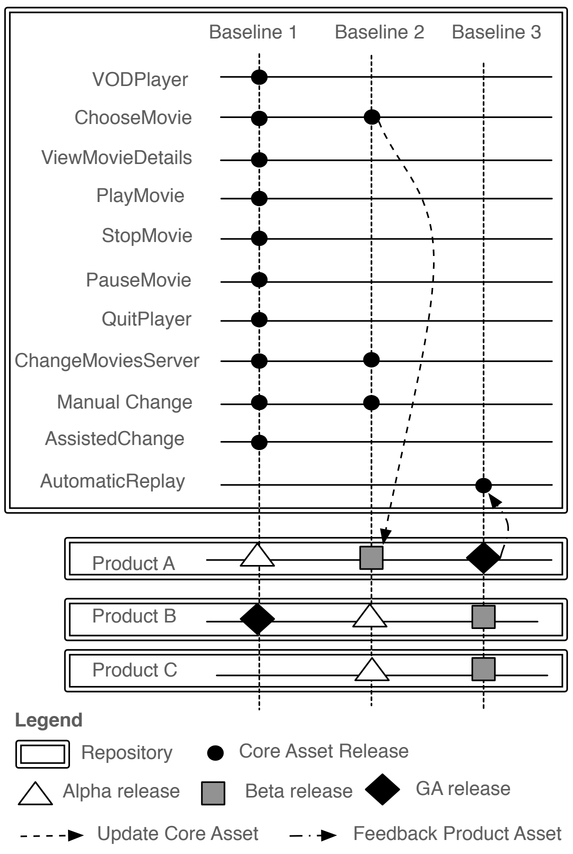


Figure 1: The SPL synchronization challenge (adapted from [11])

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. a browser extension for GitHub that accounts for the above-mentioned sync operations (subsections 5.1.1, 5.2.1 and 5.3.1). Through a single click, product engineers can now (1) generate product repositories along a certain configuration, (2) update propagations of newer core-asset versions, or (3), feedback propagation of product customizations.

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

Core Asset ID	Core Asset Name	Core Asset Description
CA1	VODPlayer	Provides the PL architecture and basic functionality to run the player
CA2	ChooseMovie	Allow users to view the list of available movie and select one
CA3	ViewMovie Details	Allow users to see a movie details: director, title and actors
CA4	PlayMovie	Allow users to start playing the movie they have already selected from the list
CA5	StopMovie	Allow users to stop the movie they are currently watching
CA6	PauseMovie	Allow users to pause the movie they are currently watching
CA7	Quit	Allow users to quit from the player
CA8	ChangeMovies Server	Allow users to change the server they are connected to
CA9	ManualChange	Provides a manual-like approach to change the server users are connected to
CA10	AssistedChange	Assists users on change server to connect to by loading a list of available servers

Table 1: *VODPlayer-PL* Core Assets

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 those 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.

3. PROPOSALS FOR VCS IN SPLS

VCSs are a cornerstone for distributed, collaborative 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*[1]. 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 taping 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.

[1]`https://help.github.com/articles/using-pull-requests/`

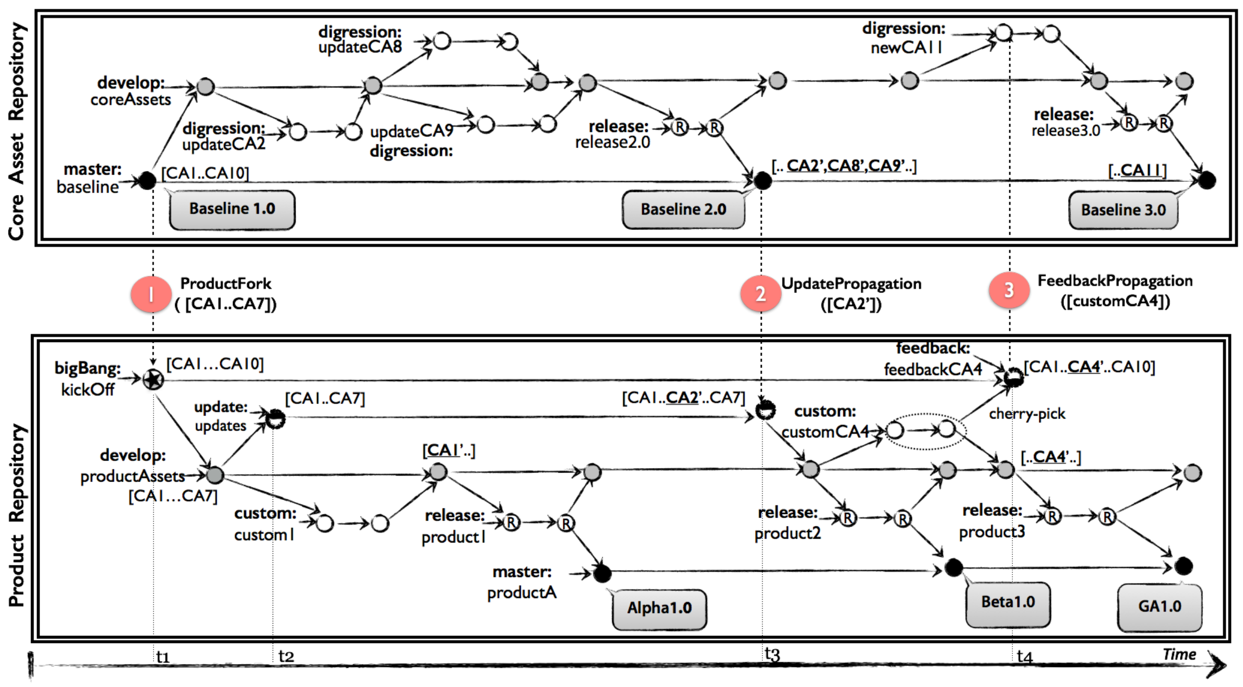


Figure 2: A closer look into the scenario described in Figure 1: branching impact due to (1) Product Fork, (2) Update Propagation and (3) Feedback Propagation. CA stands for the core assets of the sample SPL.

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).

[2]That 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.

- *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 subsection 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*-erence 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* evolve (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 *diff*-erentiate 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 replay 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 *ProductFork*, *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-

[3]VCS'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.

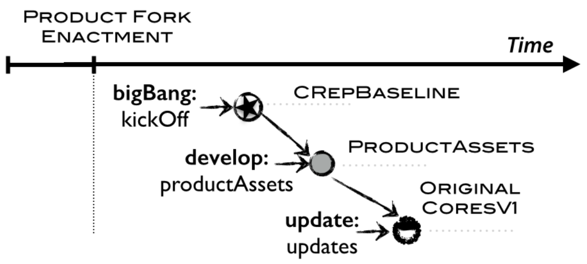


Figure 3: Product Fork involves 3 branches & 3 commits.

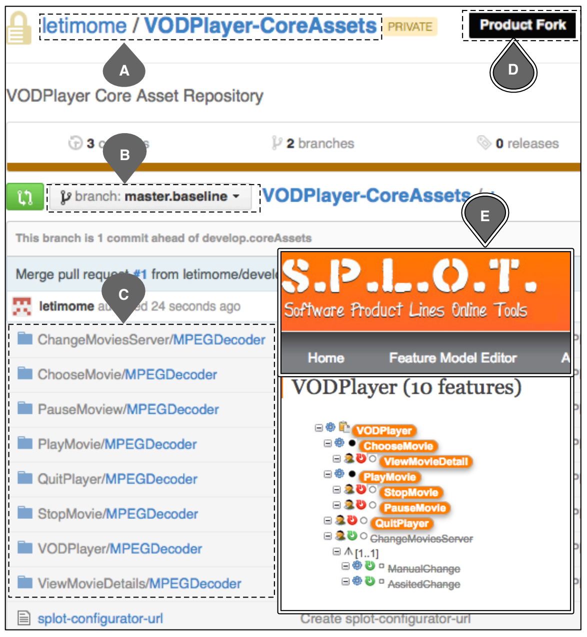


Figure 4: Leveraging *GitHub* with *ProductFork*

ing VCSs with these operations as first-class constructs. To this end, we need first to precisely indicate their operational semantics, and next, to integrate them into a VCS tool. As a proof-of-concept, we outline a *GitHub* implementation.

5.1 Product Fork

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

> ProductFork (UserAccount:userAccount, Repository:coreRepo, String[]: configuration):: 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* (line12). *BigBang:kickOff* keeps now all core assets from CoreRepo baseline (i.e, CRepBaseline). 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*[4] 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*

[4]"sha" 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

```
ProductFork(UserAccount:userAccount,Repository:coreRepo,String[]: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 do
  if (branch.name<>'master:baseline')
           DeleteBranchByName(userAccount,productRepo,branch.name)
Branch master=GetBranchByName(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)
DeleteBranchByName(userAccount,productRepo,'master:baseline')
Folder CRepBaseline=develop.commit.folders
for each coreAsset in CRepBaseline  do
  if (coreAsset.name not in configuration)
    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')
```

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: *letimome* 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* in setting 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 GitHub 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:

> UPDATEPROPAGATION(USERACCOUNT:USERACCOUNT, REPOSITORY:PRODUCTREPO, REPOSITORY:COREREPO) :: PULLREQUEST

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'S *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 reusing (i.e, ORIGINALCORES -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., ORIGINALCORES-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., ORIGINALCORESV2).
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 GETFILEBYNAME 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 this updates have

Algorithm 2 Update Propagation

```
 UpdatePropagation(UserAccount:userAccount,Repository:productRepo,Repository:coreRepo):PullRequest
Branch update=GetBranchByName(userAccount,productRepo,'update:updates')
Branch coreBaseline=GetBranchByName(userAccount,coreRepo,'master:baseline')
Folder originalCoresV1=update.commit.folders
Folder originalCoresV2= null
for each coreAsset in originalCoresV1 do{
  originalCoresV2= GetFolderByName(userAccount,coreRepo,coreBaseline,coreAsset.name)
  if (coreAsset.sha<>originalCoresV2.sha)
    CommitFolder(userAccount,productRepo,update,originalCoresV2,'new update for core asset:'+
    originalCoresV2.name)
}
File productConfig=GetFileByName(userAccount,productRepo,update,"product.config")
productConfig.content=coreBaseline.commit.sha
Commit(userAccount,productRepo,update,productConfig,'product synched to baseline'+coreBaseline.
    commit.sha)
Branch develop=GetBranchByName(userAccount,productRepo,'develop:productAssets')
CreatePullRequest(userAccount,productRepo,productRepo,develop,update,update.commit.comment)
```

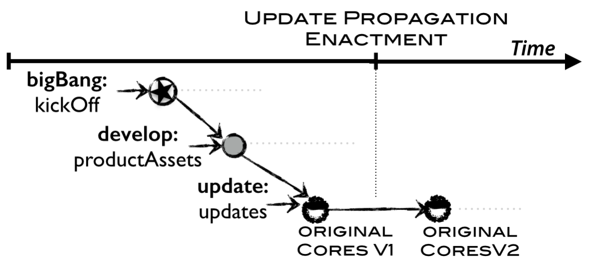


Figure 5: Update Propagation involves 1 commit for each core asset updated core asset & 1 pull_request

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 *Update_Propagation* 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:

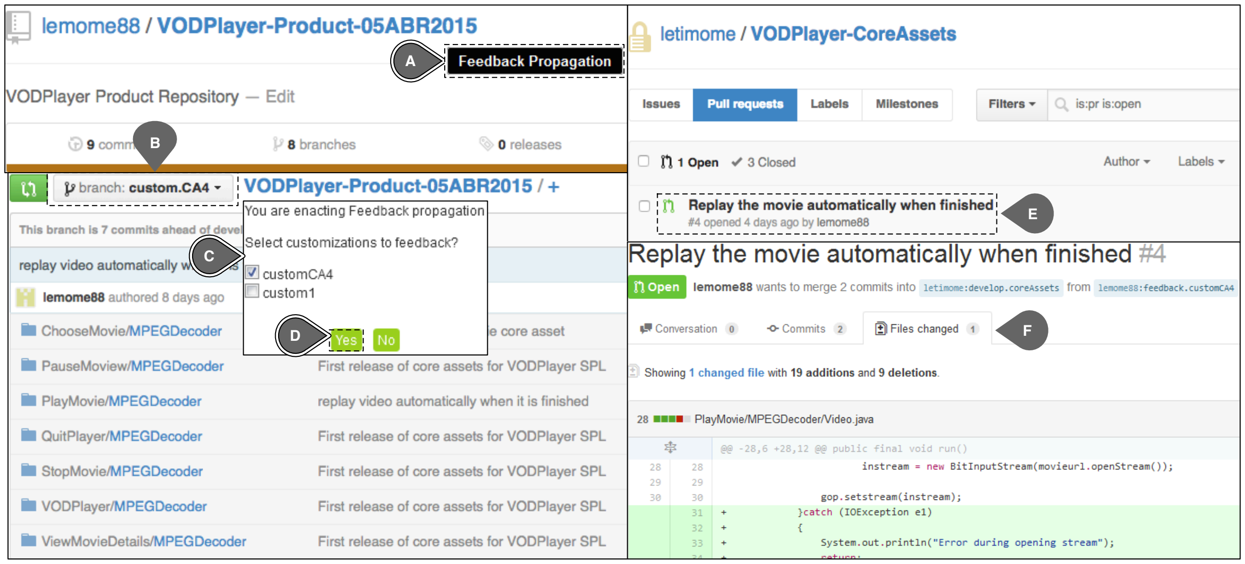
> FeedbackPropagation(UserAccount:userAccount, Repository:coreRepo, Repository:productRepo, Branch:kickOff, Branch[]:customizations, String: feedbackBranchName)::PullRequest

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.


lemome88 / VODPlayer-Product-05ABR2015
A
Feedback Propagation
VODPlayer Product Repository — Edit
9 commits
B
8 branches
0 releases
branch: custom.CA4
VODPlayer-Product-05ABR2015 / +
This branch is 7 commits ahead of develop
replay video automatically when it is finished
C
lemome88 authored 8 days ago
D
letimome / VODPlayer-CoreAssets
Issues
Pull requests
Labels
Milestones
Filters
is:pr is:open
1 Open
3 Closed
Author
Labels
E
F
Conversation 0
Commits 2
Files changed 1

You are enacting Feedback propagation

Select customizations to feedback?

- [x] customCA4
- [ ] custom1

Yes No

ChooseMovie/MPEGDecoder	core asset
PauseMoview/MPEGDecoder	First release of core assets for VODPlayer SPL
PlayMovie/MPEGDecoder	replay video automatically when it is finished
QuitPlayer/MPEGDecoder	First release of core assets for VODPlayer SPL
StopMovie/MPEGDecoder	First release of core assets for VODPlayer SPL
VODPlayer/MPEGDecoder	First release of core assets for VODPlayer SPL
ViewMovieDetails/MPEGDecoder	First release of core assets for VODPlayer SPL

Replay the movie automatically when finished
#4 opened 4 days ago by lemome88

Replay the movie automatically when finished #4

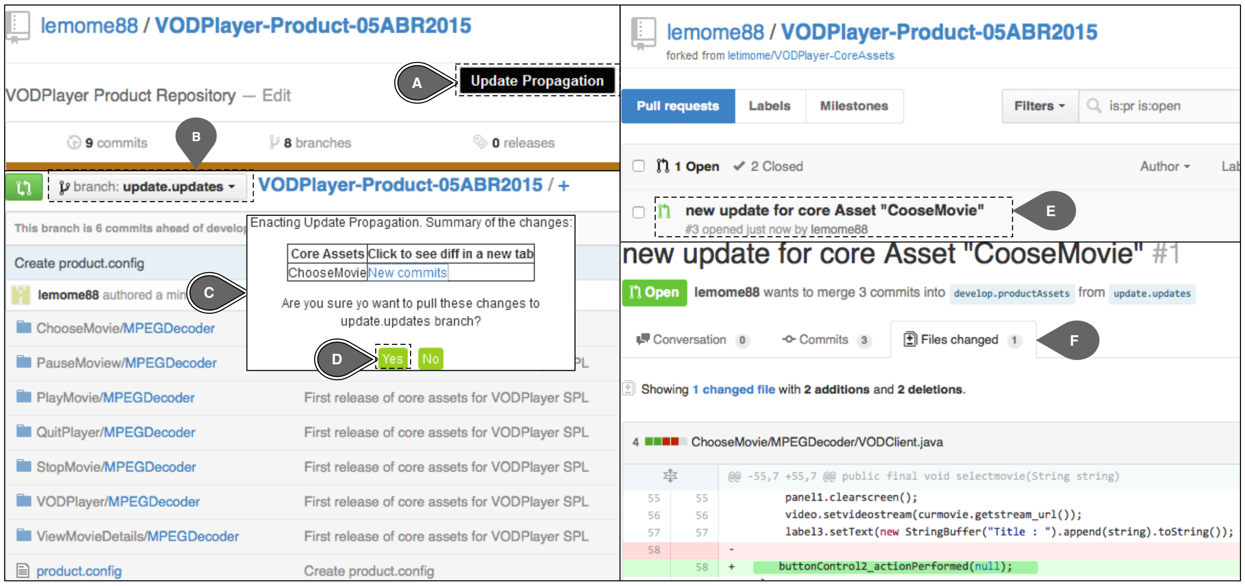
Open lemome88 wants to merge 2 commits into letimome:develop.coreAssets from lemome88:feedback.customCA4

Showing 1 changed file with 19 additions and 9 deletions.

28 PlayMovie/MPEGDecoder/Video.java

```
@@ -28,6 +28,12 @@ public final void run()
28  28                        instream = new BitInputStream(movieurl.openStream());
29  29
30  30                  gop.setstream(instream);
    31  +           }catch (IOException e1)
    32  +           {
    33  +               System.out.println("Error during opening stream");
```

Figure 6: Leveraging *GitHub* with *FeedBackPropagation:* enacting (left) and outcome (right).


lemome88 / VODPlayer-Product-05ABR2015
A
Update Propagation
VODPlayer Product Repository — Edit
9 commits
B
8 branches
0 releases
branch: update.updates
VODPlayer-Product-05ABR2015 / +
This branch is 6 commits ahead of develop
Create product.config
C
lemome88 authored a minute ago
D
lemome88 / VODPlayer-Product-05ABR2015
forked from letimome/VODPlayer-CoreAssets
Pull requests
Labels
Milestones
Filters
is:pr is:open
1 Open
2 Closed
Author
Labels
E
F
Conversation 0
Commits 3
Files changed 1

Enacting Update Propagation. Summary of the changes:

Core Assets	Click to see diff in a new tab
ChooseMovie	New commits

Are you sure yo want to pull these changes to update.updates branch?

Yes No

ChooseMovie/MPEGDecoder	
PauseMoview/MPEGDecoder	
PlayMovie/MPEGDecoder	First release of core assets for VODPlayer SPL
QuitPlayer/MPEGDecoder	First release of core assets for VODPlayer SPL
StopMovie/MPEGDecoder	First release of core assets for VODPlayer SPL
VODPlayer/MPEGDecoder	First release of core assets for VODPlayer SPL
ViewMovieDetails/MPEGDecoder	First release of core assets for VODPlayer SPL
product.config	Create product.config

new update for core Asset "CooseMovie"
#3 opened just now by lemome88

new update for core Asset "CooseMovie" #1

Open lemome88 wants to merge 3 commits into develop.productAssets from update.updates

Showing 1 changed file with 2 additions and 2 deletions.

4 ChooseMovie/MPEGDecoder/VODClient.java

```
@@ -55,7 +55,7 @@ public final void selectmovie(String string)
55  55          panel1.clearscreen();
56  56          video.setvideostream(curmovie.getstream_url());
57  57          label3.setText(new StringBuffer("Title : ").append(string).toString());
58      -
    58  +      buttonControl2_actionPerformed(null);
```

Figure 7: Leveraging *GitHub* with *UpdatePropagation:* enacting (left) and outcome (right).

Algorithm 3 FeedBackPropagation

```
FeedbackPropagation(UserAccount:userAccount,Repository:coreRepo,Repository:productRepo,Branch[]:
    customizations,String:feedbackBranchName)
Branch bigBang=GetBranchByName(userAccount,productRepo,"bigBang:kickOff")
Branch newFeedback=new Branch(userAccount,productRepo,bigBang,feedbackBranchName)
for each custom in customizations do {
  Folder customizedAssets=GetChangesFromBranch(userAccount,productRepo,custom)
  for each  custAsset in customizedAssets do
    CommitFolder(userAccount,productRepo,newFeedback,custAsset,'customized asset:'+custAsset.name)
}
Branch develop=GetBranchByName(userAccount,coreRepo,'develop:coreAssets')
CreatePullRequest(userAccount,coreRepo,productRepo,develop,newFeedback,newFeedback.commit.comment)
```

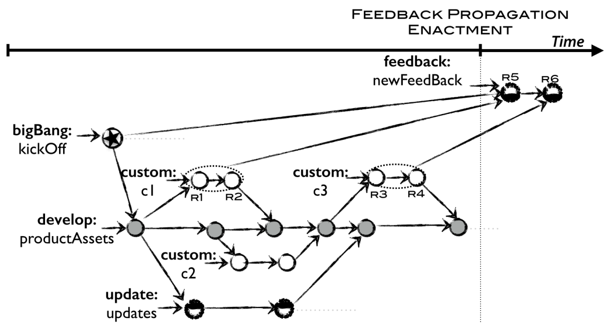


Figure 8: Feedback Propagation involves 1 branch & 1 commit for each Custom branch involved & 1 pull_request

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

3. When all CUSTOMIZATIONS have been merged into NEW FEEDBACK 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 *FeedBack_Propagation* 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. Behind 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 those 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 extent 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.

Acknowledgment

This work is co-supported by the Spanish Ministry of Education, and the European Social Fund under contract TIN2011-23839 (Scripting). Montalvillo enjoys a doctoral grant from the University of the Basque Country.

7. REFERENCES

[1] Git branching model. http://nvie.com/posts/a-successful-git-branching-model.
[2] ANASTASOPOULOS, M. *Evolution Control for Software Product Lines: An Automation Layer over Configuration Management.* PhD thesis, Fraunhofer IESE, 2013.
[3] APEL, S., BATORY, D., KÄSTNER, C., AND SAAKE, G. *Feature-Oriented Software Product Lines.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.
[4] BARREIROS, J., AND MOREIRA, A. A cover-based approach for configuration repair. *SPLC* (2014).
[5] BOSCH, J. Software product lines: organizational alternatives. *ICSE* (2001).
[6] BOSCH, J. Maturity and evolution in software product lines-approaches, artefacts and organization. *SPLC* (2006).
[7] DEELSTRA, S., SINNEMA, M., AND BOSCH, J. Product derivation in software product families: A case study. *Journal of Systems and Software* (2005).
[8] DÍAZ, O., AND ARELLANO, C. The augmented web: Rationales, opportunities and challenges on browser-side transcoding. *To appear at ACM Transactions on the Web* (2015).
[9] FISCHER, S., LINSBAUER, L., LOPEZ-HERREJON, R. E., AND EGYED, A. Enhancing clone-and-own with systematic reuse for developing software variants. In *ICSME* (2014).
[10] GURP, J. V., AND PREHOFER, C. Version management tools as a basis for integrating Product Derivation and Software Product Families. *SPLC* (2006).
[11] KRUEGER, C., AND CLEMENTS, P. Systems and Software Product Line Engineering. *Encyclopedia of Software Engineering* (2013).
[12] KRUEGER, C. W. Towards a Taxonomy for Software Product Lines. *PFE* (2004).
[13] KRUEGER, C. W., AND COVE, L. H. Variation management for software production lines. *SPLC* (2002).
[14] MCGREGOR, J. D. The Evolution of Product Line Assets. *Technical Report* (2003).
[15] NUNES, C., GARCIA, A., LUCENA, C., AND LEE, J. History-sensitive heuristics for recovery of features in code of evolving program families. *SPLC* (2012).
[16] POHL, K., BÖCKLE, G., AND LINDEN, F. J. V. D. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer-Verlag New York, Inc., 2005.
[17] RUBIN, J., CZARNECKI, K., AND CHECHIK, M. Managing Cloned Variants : A Framework and Experience. *SPLC* (2013).
[18] S.P.L.O.T. http://www.splot-research.org.
[19] STAPLES, M., AND HILL, D. Experiences adopting software product line development without a product line architecture. *APSEC* (2004).
[20] THAO, C. T. C., MUNSON, E., AND NGUYEN, T. Software Configuration Management for Product Derivation in Software Product Families. *ECBS* (2008).
[21] WALRAD, C., AND STROM, D. The importance of branching models in SCM. *Computing Practices* (2002).