Safe Evolution of Software Product Lines: Feature Extraction Scenarios

Fernando Benbassat
fcb@cin.ufpe.br

Paulo Borba
phmb@cin.ufpe.br

Leopoldo Teixeira
lmt@cin.ufpe.br

Informatics Center
Federal University of Pernambuco
Recife, Brazil

***Abstract*—Software Product Lines can improve productivity and product quality, but product line maintenance is not simple, since a single change can impact several products. In many situations, it is desirable to provide some assurance that we can safely change a SPL in the sense that the behaviour of existing products is preserved. Developers can rely on previously proposed safe evolution notions, by means of transformation templates to ensure safe evolution. However, the existing templates were only applied in scenarios where a product line expands, and have not been evaluated in the context of extracting features from existing code. Therefore, we conducted a study using an industrial system developed in Java with 400 KLOC. This study revealed the need for new templates to address feature extraction scenarios, as well as improving the existing templates notation to address more expressive mappings between features and code assets. As a result of this study, we successfully extracted a product line from this existing system using the proposed templates, and also found evidence that the new templates can help to prevent defects during product line evolution.**

***Keywords*—Software Product Lines, Product Line Evolution, Safe Evolution, Refinement**

I. Introdução

Linhas de produtos de software (LPS) permitem a geração de produtos relacionados a partir de artefatos de software reutilizáveis [1]. Os produtos gerados podem possuir funcionalidades ou características específicas que os diferenciam dos demais, porém a maior vantagem de utilizar LPS é o reuso de funcionalidades. Reutilizar de forma sistemática um conjunto específico de produtos melhora a qualidade e produtividade [1]. Tais benefícios se justificam quando a evolução, ou correção de um único defeito em uma determinada parte de código, é propagada para todos os produtos que a utilizam, não sendo necessário replicar esforços.

Por esse mesmo motivo que leva aos benefícios de LPS, a tarefa de evoluir uma LPS pode ser complexa, pois a introdução de um simples defeito no código pode afetar vários produtos. De fato, tal evolução pode ser considerada segura ou não com relação ao comportamento dos produtos antes e depois da mudança. Para a evolução ser considerada segura, o comportamento dos produtos existentes deve ser preservado, isto é, cada produto da LPS inicial deve ter comportamento compatível com pelo menos um produto da nova LPS [2], conforme ilustrado na Figura 1.

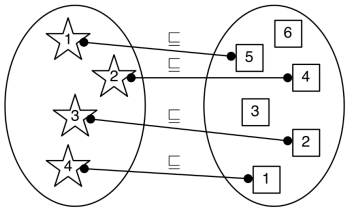

Figura 1. Exemplo ilustrativo do conceito de evolução segura. Elipses representam LPS e seus respectivos produtos.

Em geral, adicionar novas *features* opcionais, e alterar uma *feature* de obrigatória para opcional, são exemplos de evolução segura de LPS. Por exemplo, ao transformarmos uma *feature* obrigatória em opcional, a LPS passa a ter mais produtos, possivelmente o dobro de produtos. Os produtos onde a *feature* é selecionada são exatamente iguais aos produtos existentes antes da transformação. Ou seja, cada produto da LPS inicial tem comportamento compatível[1] com pelo menos um produto da nova LPS, o que está consistente com a noção ilustrada na Figura 1. Para os demais produtos que não incluem a *feature* modificada, basta que sejam bem-formados. De fato, terão comportamento distinto dos produtos existentes, porém, para estes, também não há correspondentes na LPS inicial.

Contudo, garantir que uma evolução é segura não é tarefa simples, pois uma mudança pode afetar uma variedade de produtos, além do fato de que é necessário alterar diferentes artefatos da LPS manualmente, o que pode resultar em erros. Para auxiliar os desenvolvedores durante a evolução de uma LPS, foram propostos *templates* de transformações de LPS [2], [3] que garantem a evolução segura. Os *templates* fornecem orientações para modificar os elementos da LPS e podem ajudar a evitar problemas decorrentes da evolução manual de uma LPS.

No entanto, os *templates* existentes foram aplicados apenas em cenários onde uma LPS cresce, expandindo funcionalidades. Mas é importante também avaliá-los no contexto de extra-

[1]No sentido de preservação de comportamento observável.

978-1-5090-5086-4/16 $31.00 © 2016 IEEE
DOI 10.1109/SBCARS.2016.21

ção de *features* a partir de código e funcionalidade existente, onde esse código existente é associado a novas *features*. Por exemplo, isto é particularmente útil no processo de transformar um produto em uma LPS. Para avaliar a aplicação desses *templates* no contexto de extração de *features*, e assim reforçar a validade externa dos resultados existentes [3], decidimos aplicá-los em um sistema industrial desenvolvido em Java, com todas as suas funcionalidades já implementadas, durante o processo de reestruturação desse sistema como uma LPS. Identificamos então que os *templates* existentes se aplicam apenas em situações de expansão das LPS.

Assim foi constatada a necessidade de criação de novos *templates* para casos onde trechos de código existentes são associados a *features*, obrigatórias ou opcionais. No caso de *feature* opcional, isto torna possível gerar produtos com e sem o comportamento associado àquele código. Além de propor novos *templates* com este foco, devido ao grande número de artefatos da LPS em questão, adaptamos *templates* existentes para usar uma notação de associação entre expressões de *features* e artefatos de software mais concisa; ao invés de listar cada artefato associado a uma expressão de *feature*, o desenvolvedor utiliza apenas um tipo de transformação que referencia um conjunto de artefatos da linha.

Depois de definidos os novos *templates* de extração de *features* e o escopo do sistema escolhido, aplicamos os *templates* e uma LPS foi derivada de forma sistemática por meio da aplicação dos *templates*. Ao todo, foram extraídas 6 *features*, a partir das quais geramos 10 produtos. Note que o código referente às 6 *features* já estava implementado, porém não distinguível (isolado do resto do código da aplicação) e apenas um único produto com todas as funcionalidades era gerado inicialmente. Para gerarmos os 10 produtos da nova LPS utilizamos a ferramenta *Hephaestus* [4], que avalia modelos de LPS e gera artefatos para instâncias específicas de uma LPS. Após a geração de cada produto da LPS extraída, foi executado um subconjunto de testes que era relacionado às funcionalidades presentes naquele produto, para verificar se o produto gerado apresentava alguma diferença de comportamento decorrente da evolução da LPS. Deste modo, colhemos evidências de que os novos *templates* de extração de *features* ajudam a prevenir defeitos e preservam o comportamento dos produtos existentes durante a evolução da uma LPS.

Em resumo, este trabalho tem como contribuições

- (i) proposta de novos *templates* para extração de *features*, incluindo melhorias na sintaxe utilizada na associação entre expressões de *features* e artefatos de implementação;
- (ii) avaliação da utilidade e segurança proporcionada pelo uso dos novos *templates* em um sistema com 400 KLOC.

O restante deste artigo está organizado como descrito a seguir. A Seção II apresenta um exemplo motivante de evolução segura usando *templates* existentes na literatura e os problemas encontrados. O detalhamento dos novos *templates* e o processo utilizado para criação dos mesmos é discutido na Seção III. Os resultados do estudo realizado estão na Seção IV e os trabalhos relacionados na Seção V. Finalmente, nós concluímos na Seção VI.

II. Motivação

Uma LPS é representada aqui como uma tripla de elementos: o *Feature Model* (FM), que possui as *features* e as dependências entre elas; o *Asset Mapping* (AM), que relaciona nomes de artefatos e os artefatos diretamente; e o *Configuration Knowledge* (CK), que mapeia expressão de *features* para nomes de artefatos [2]. Diversos cenários de evolução exigem a alteração de mais de um elemento. Por exemplo, para adicionar uma nova *feature* opcional, é necessário modificar e combinar corretamente estes 3 elementos conforme apresentado na Figura 2. Na figura a tripla (*F*,*A*,*K*) representa o FM, o AM e o CK, respectivamente. Uma simples mudança em um destes elementos pode impactar diretamente vários produtos gerados, e consequentemente, seu comportamento.

A Figura 2 detalha o cenário. Na parte de cima da imagem a tripla (*F*,*A*,*K*) denota a LPS na versão inicial, que gera apenas um produto.[2] Para transformar um produto em uma LPS, inicialmente assumimos o produto como sendo essa LPS simples, à qual adicionamos *features*, que resultam na capacidade de gerar mais produtos. Na parte de baixo, pintadas de azul, estão as modificações realizadas para adicionar a nova *feature* opcional *EventsSimulation*. No sistema, essa *feature* simula eventos para avaliação sintática de código fonte de aplicações.

Inicialmente o sistema não possui nenhuma *feature*, portanto o FM contém apenas *Root*. Para o AM, destacamos 3 arquivos em particular, que são alguns dos arquivos relacionados à nova *feature*. As reticências indicam que podem existir outros artefatos na lista. O CK relaciona *Root* a todos os arquivos do sistema, inclusive aos 3 arquivos mencionados. As alterações na LPS fazem com que o FM passe a ter duas *features*, *Root* e a opcional *EventsSimulation*, resultando na possibilidade de gerar dois produtos, incluindo ou não a nova *feature*. No CK, foi preciso adicionar uma nova linha com o lado esquerdo contendo uma expressão lógica denotando a ausência da nova *feature*. Do lado direito, associamos a ação de transformação *remove* à ausência da *feature*. Desta forma, o produto que inclui a nova *feature EventsSimulation* também inclui os arquivos *file1*, *file2* e *file3*. Isto resulta no mesmo comportamento do produto gerado antes da transformação, o que satisfaz o requisito de que todo produto da LPS inicial deve ter um correspondente na LPS resultante. Já no produto sem a nova *feature EventsSimulation*, os arquivos *file1*, *file2* e *file3* não estão presentes. Note que embora estejamos adicionando uma nova feature, o AM não foi alterado, visto que estamos extraindo esta feature a partir do código existente. Sendo assim, nenhum novo artefato foi adicionado a LPS. Além disso, para facilitar o entendimento, estamos mostrando apenas uma pequena parte dos artefatos relacionados à *feature EventsSimulation*.

Evoluir manualmente uma LPS é propenso a erros, como por exemplo, associar incorretamente expressões de *features* a nomes de artefatos no CK [3]. Além disso, os erros introduzidos durante a evolução manual da LPS podem ser difíceis de controlar porque eles estão presentes apenas em certas

[2]Todo produto pode ser representado como uma LPS desta forma.

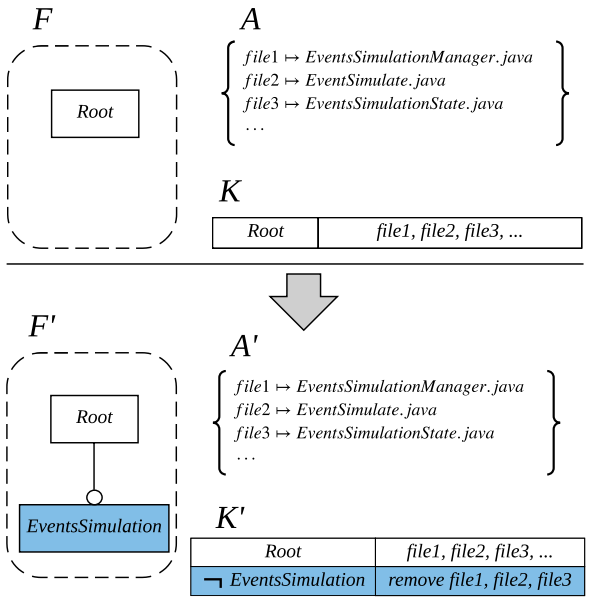

Figura 2. Exemplo: Extrair *feature* opcional a partir de código existente.

configurações de produtos. Para reduzir os problemas durante a evolução de uma LPS, desenvolvedores podem usar *templates* de transformação de LPS [2], [3], que descrevem mudanças que são consistentes com o conceito de evolução segura. Deste modo, com o objetivo de reforçar a validade externa dos resultados existentes, optamos por usar os *templates* propostos em um sistema industrial desenvolvido em Java e com todas as suas funcionalidades consolidadas. Para isso, escolhemos um primeiro cenário, mais simples, apenas para associar alguns artefatos de código-fonte do sistema a uma nova *feature* opcional, assim como ilustrado na Figura 2.

Dentre os *templates* existentes na literatura, o que mais se aproxima do cenário apresentado é o *template* para adicionar nova *feature* opcional exibido na Figura 3 [3]. *Templates* tem por padrão do lado esquerdo o estado inicial da LPS, e no lado direito o estado final. Com base na diferença entre lado esquerdo e lado direito percebemos o que foi alterado durante a evolução. O símbolo $\sqsubseteq$ indica refinamento de LPS [2], que é a formalização de evolução segura. Na parte de baixo da figura estão as pré-condições para aplicar o *template*. Com todas as condições satisfeitas, o desenvolvedor modifica os elementos da LPS de acordo com o *template* escolhido. Para evoluirmos uma LPS de forma segura, é preciso seguir cuidadosamente todas as regras e orientações apresentadas pelo *template*, caso contrário ele não pode ser usado.

Nesse contexto, tentamos utilizar o *template* da Figura 3 para adicionar a *feature* opcional *EventsSimulation* no sistema. Esse *template* é útil quando o desenvolvedor precisa adicionar uma *feature* opcional, sem alterar os artefatos existentes. De acordo com a Figura 3, só podemos introduzir uma nova *feature* opcional *O* juntamente com um novo artefato *a´*, uma

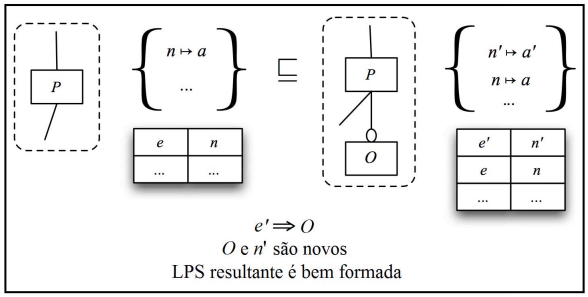

Figura 3. ADICIONAR NOVA FEATURE OPCIONAL [3]

vez que esse artefato só será incluído em produtos que incluem a *feature O*. Além disso, a primeira linha do CK associa *e* com *n*, para ilustrar que os demais itens existentes do CK permanecem inalterados após a transformação. Por outro lado, o exemplo da Figura 2 está incluindo uma nova *feature* partindo de artefatos já existentes. Como todo o código e artefatos de software relacionados à nova *feature EventsSimulation* já estão implementados (*file1*, *file2* e *file3*), o *template* da Figura 3 não pode ser usado. Deste modo, após analisar a lista de *templates* disponíveis na literatura [3],[3] verificamos que nenhum deles atendia o nosso cenário de evolução.

Visto que a maior parte dos *templates* existentes devem ser aplicados em cenários de expansão da LPS, com adição de novas funcionalidades e artefatos de software, não conseguimos com eles transformar, de forma segura, o sistema industrial em questão em uma LPS, como desejado. Então decidimos propor novos *templates* para contextos de extração de *features*, onde adicionamos *features* a partir de artefatos existentes. Entendemos que a necessidade de transformar código existente em *features* é bastante comum e outros cenários podem existir durante o desenvolvimento de uma LPS. Portanto, na próxima seção apresentamos em detalhe os *templates* propostos.

III. TEMPLATES PARA EXTRAÇÃO DE FEATURES

Para lidar com cenários de extração de *features*, como o ilustrado no exemplo de motivação, definimos 3 novos *templates* para evolução segura de LPS. Todos eles focam em transformar artefato existente em *feature*. No entanto a escolha de qual *template* utilizar depende do contexto envolvido e de que mecanismos devem ser usados para estruturar o código da *feature*: composicional, anotativo ou ambos [5]. No contexto composicional, a variabilidade é gerenciada por meio de artefatos completos. Ou seja, a implementação de uma *feature* está isolada em classes e módulos como um todo. Já no contexto anotativo, as *features* são implementadas por meio de trechos de código espalhados pelo sistema, geralmente entrelaçados com a implementação de outras *features* dentro de um mesmo artefato. Isto torna necessário o uso de algum mecanismo de pré-processamento, como compilação condicional e demarcação do código com anotações dentro dos artefatos

[3]Catálogo de *templates* para evolução segura https://github.com/spgroup/pl-refinement-templates-catalog.

[6]. Também podemos ter os dois contextos, composicional e anotativo, utilizados em combinação para implementar uma mesma *feature*.

Para descobrir novos *templates* de evolução segura, utilizamos um sistema industrial familiar, com todas as funcionalidades principais implementadas, e aproximadamente 400 KLOC. Inicialmente, escolhemos extrair uma *feature* que, pelo nosso conhecimento do domínio, era adequada de ser extraída como opcional. Nosso interesse era ter uma versão do sistema sem a funcionalidade que simula eventos para avaliação sintática de código fonte de aplicações (*feature EventsSimulation*). Logo, percebemos que a *feature* escolhida envolvia apenas classes completas, caracterizando um contexto composicional.

Antes de propor novos *templates*, tentamos usar uma sequência de *templates* existentes, aplicando pequenas mudanças e evoluindo a LPS por etapas até conseguir adicionar a *feature* desejada. Visto que nenhum dos *templates*, nem uma sequência destes, se aplica ao nosso cenário de evolução, definimos o primeiro *template* usando como base o previamente proposto para adicionar nova *feature* opcional, apresentado na Figura 3. Partimos do princípio que a estrutura da mudança no FM deve ser a mesma do *template* base. O AM não deve ser alterado do estado inicial para o final, já que artefatos não serão modificados nem incluídos no processo de extração da *feature*. Já o CK dos *templates* existentes permitia apenas associar expressões de *feature* a seleção de artefatos, ou, a transformações simples de pré-processamento dos artefatos e ativação de tags de pré-processamento. Para lidar com a grande quantidade de artefatos do sistema em questão, preferimos usar nos templates propostos aqui a notação mais expressiva de CK adotada pela ferramenta *Hephaestus*, que considera transformações adicionais, que permitem selecionar todos os artefatos existentes na LPS, bem como remover artefatos.

A partir desse momento, iniciamos a evolução segura da LPS seguindo o passo a passo do processo de extração de *features* detalhado na Figura 4. Antes de iniciar o processo de desenvolvimento definimos um conjunto com as possíveis *features* a serem extraídas, de acordo com nossa experiência e conhecimento do sistema e do domínio. No primeiro passo, escolhemos a *feature* a ser extraída e em seguida listamos todos os artefatos relacionados à ela (passo 2). Dentre os passos mais importantes do processo estão os de número 2, 4 e 6. O passo de número 2 é crítico, pois depende do conhecimento do desenvolvedor para saber quais arquivos e partes do código do sistema estão relacionados à *feature* escolhida. A seleção incorreta de qualquer arquivo relacionado à *feature* pode impactar no comportamento do produto final. Depois de listados todos os arquivos, fizemos uma validação com outro desenvolvedor com conhecimento do sistema (passo 3), para confirmar se a lista de artefatos estava correta e poderia ser usada para extrair a *feature* escolhida.

Com a lista de artefatos da *feature*, temos informações suficientes para seguir para o passo 4 e identificar qual o *template* mais apropriado para o cenário escolhido. Neste passo é preciso ter atenção para verificar se todas as pré-condições e regras existentes no *template* são aplicáveis ao

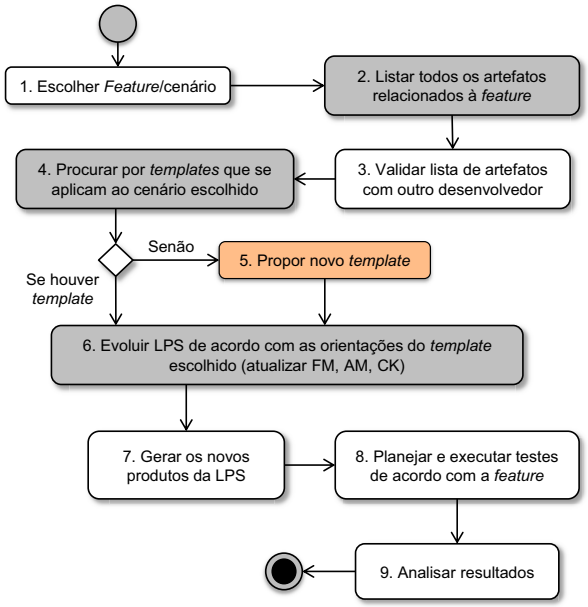

Figura 4. Passos do processo de extração de novas *features*.

cenário escolhido. Se o *template* não for apropriado para o cenário, não poderemos confiar que a evolução da LPS será segura. Caso nenhum dos *templates* existentes seja adequado, um novo *template* deverá ser proposto no passo 5. Este passo está destacado com uma cor diferente no processo, pois diz respeito à proposta de novos *templates*, ao invés do uso de existentes. No nosso estudo, utilizamos este passo para propor os *templates* discutidos a seguir. Na medida que mais *templates* forem propostos, este passo nem sempre se fará necessário. O próximo passo é modificar os elementos da LPS de acordo com as orientações apresentadas no *template* escolhido (passo 6). Após a modificação, geramos os novos produtos da LPS (passo 7), usando a ferramenta *Hephaestus* [4]. A geração dos produtos é útil para verificar boa formação da LPS e possibilitar os testes.

Os testes foram planejados de acordo com a *feature* extraída, a partir de um conjunto de testes já existente, e executados antes e após as mudanças na LPS (passo 8). No passo 9, comparamos os resultados dos testes antes e depois da transformação realizada na LPS e verificamos que os resultados foram os mesmos. Somando todos os testes executados na versão inicial do sistema e em todos os novos produtos gerados pela LPS extraída, ao total foram executados 5500 testes unitários (automáticos) e 92 testes funcionais (manuais). Isso nos dá certa evidência de que o comportamento dos produtos existentes foi preservado.

Não analisamos o processo de extração de *features* em si, para avaliar se os passos definidos são os melhores a serem seguidos, pois não era o foco do nosso estudo. Acreditamos que o resultado obtido durante o processo de desenvolvimento não teve impacto negativo devido ao processo de extração

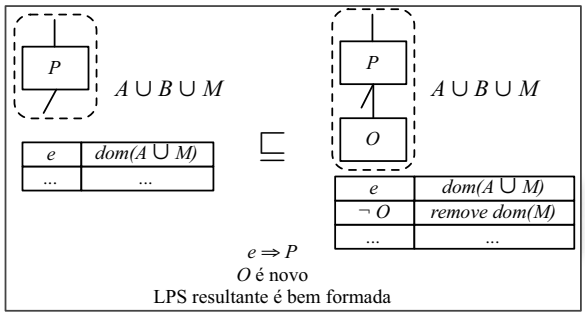

Figura 5. TRANSFORMAR ARTEFATO EXISTENTE EM NOVA FEATURE

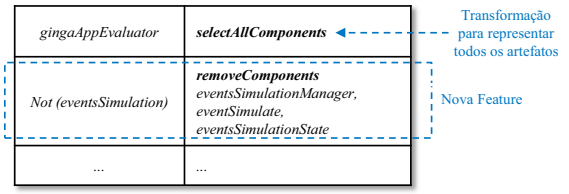

Figura 6. Exemplo de *Configuration Knowledge* (CK) mais expressivo.

de *features* adotado, que serviu como um guia prático de acompanhamento e planejamento das várias etapas do desenvolvimento. Este processo de desenvolvimento foi executado para cada nova extração de *feature* durante a evolução da LPS. A seguir apresentamos detalhadamente os novos *templates* propostos, bem como expomos suas principais diferenças.

A. TRANSFORMAR ARTEFATO EXISTENTE EM NOVA FEATURE

Este *template* é útil quando um desenvolvedor precisa associar artefatos existentes a uma nova *feature*, sem alteração ou adição de artefatos, como ilustra a Figura 5. O FM da linha resultante não especifica o tipo da nova *feature*, podendo ela ser opcional, obrigatória, alternativa ou *OR*. Note que, se a nova *feature* não for obrigatória, o conjunto de possíveis produtos é aumentado. Os produtos contendo a *feature* $\{P\}$ antes da transformação será equivalente ao produto contendo as *features* $\{P,O\}$ na nova LPS. O *template* representa o fato de que uma *feature* pode ter outras *features* relacionadas a ela usando linhas acima e abaixo dela.

Com relação ao AM, os artefatos da LPS original podem ser divididos em três conjuntos. O conjunto M representa os artefatos de nosso interesse, isto é, aqueles que serão associados com a nova *feature*. Os demais são artefatos associados com outras *features* ou obrigatoriamente incluídos nos produtos da LPS. A primeira linha do CK original associa *e* com *dom(A ∪ M)*, onde *e* é uma expressão de *feature* qualquer que implica na seleção da *feature P*, e a notação *dom(...)* representa a seleção de um conjunto de nomes de artefatos. No caso deste *template* em específico, utilizamos $A \cup M$, indicando que devem ser incluídos todos os artefatos nos conjuntos A e M. As reticências indicam que este CK pode inclusive ter outros mapeamentos, associando artefatos do conjunto B por exemplo, o que dá maior generalidade ao *template*.

Como mencionado, M representa os artefatos que serão associados à nova *feature* após a transformação. Na segunda linha do novo CK, quando a nova *feature O* não for selecionada, todos os artefatos relacionados à ela são removidos. Logo, os artefatos que antes eram selecionados quando a expressão *e* fosse satisfeita, na nova linha só serão incluídos no produto final quando a *feature O* também for satisfeita. A Figura 6 mostra o exemplo real de parte do CK mais expressivo onde uma única transformação pode representar todos os artefatos da LPS. Neste caso em específico, a transformação *selectAllComponents* inclui todos os artefatos da LPS. Daí a utilidade de termos a transformação *removeComponents*, associada com a ausência da *feature EventsSimulation* em um produto, o que remove apenas os artefatos relacionados. Isto evita que na primeira linha do CK tenhamos que listar todos os artefatos, exceto pelos três associados a *EventsSimulation*.

A pré-condição do *template*, além de afirmar que a expressão e deve implicar a *feature* P, também diz que não podemos ter uma outra *feature* chamada *O* no FM, pois isso deixaria o mesmo inválido. Finalmente, o *template* também requer que a LPS resultante seja bem formada, para certificar de que produtos gerados sem os artefatos associados com a nova *feature* sejam válidos. Podemos verificar a restrição de boa formação usando a abordagem de composição segura [7], [8]. Este *template* está consistente com a noção de evolução segura, pois para todo produto da LPS inicial, há um produto correspondente sintaticamente equivalente na LPS resultante — o produto onde a nova *feature* é selecionada.

Ao tentar extrair novas *features*, descobrimos a necessidade de criar outros dois *templates*, também para cenários de extração de *features*, um para o contexto anotativo, e outro para os dois contextos ao mesmo tempo, anotativo e composicional.

B. TRANSFORMAR ARTEFATO PRÉ-PROCESSADO EM NOVA FEATURE

O segundo *template* pode ser aplicado em casos onde a implementação da *feature* a ser extraída está espalhada em partes de arquivos que precisam ser pré-processados de acordo com diretivas de compilação condicional. Desta forma, utilizamos uma notação ligeiramente diferente do template anterior. A Figura 7 mostra o *template* para extrair *feature* a partir do contexto anotativo.

Como também estamos introduzindo uma nova *feature*, modificamos o FM de maneira similar ao que foi apresentado no *template* anterior, contemplando a possibilidade de adicionar qualquer tipo de *feature*. Por conta do contexto anotativo, o CK do estado inicial da LPS contém uma expressão de *feature e* associada com a transformação responsável por pré-processar artefatos da LPS, de acordo com as diretivas de compilação condicional. A meta-variável *n* representa o nome do artefato relacionado à nova *feature*, que após as mudanças na LPS, tem parte do seu código (c) envolvido por uma anotação $\#if\ x$. Ou seja, o código c agora pode ser incluso ou não no arquivo resultante, dependendo do pré-processamento do arquivo.

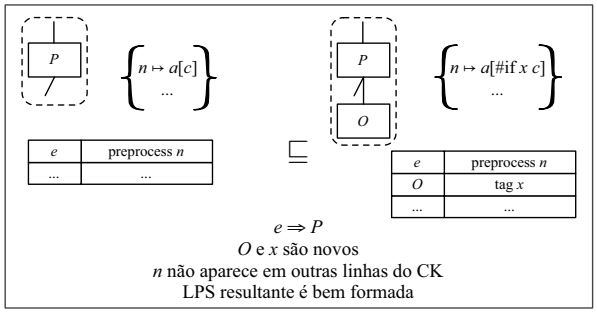

Figura 7. TRANSFORMAR ARTEFATO PRÉ-PROCESSADO EM NOVA FEATURE

Uma nova linha no CK é adicionada associando a nova *feature* (O) com a transformação *tag* x. Esta transformação é responsável por definir como verdadeira a *tag* x, ou seja, habilita a inclusão do código c envolvido pela diretiva $\#if$ x introduzida no arquivo. As pré-condições do novo *template* são similares às apresentadas no *template* anterior. A distinção neste caso é que além da *feature* O, não podemos ter outra *tag* chamada x no CK. Além disso, o artefato n relacionado à nova *feature* não pode aparecer em outras linhas do CK, para assegurar a evolução segura da LPS. No *template* em questão, representamos a inclusão de uma diretiva em um arquivo apenas, mas este *template* pode ser aplicado a diversos arquivos ao mesmo tempo, vide as reticências na representação do AM.

A Figura 8 ilustra um exemplo prático de compilação condicional usada em um método de classe, durante a evolução da nossa LPS. Nesse exemplo o a seria a classe como um todo, e o c seria a declaração do método *getNCLFunctionalAreas(NCLFileData nclData)*. Introduzimos a anotação $\#ifdef$ seguida da *tag* referente à nova *feature EvaluateNCLFunctionalAreas*, e no final, para delimitar o encerramento do bloco de código condicional, adicionamos $\#endif$. A LPS resultante é bem formada quando os produtos gerados sem a nova *feature*, ou seja, sem os trechos de código envolvidos com a *tag "tagEvalNCLFunctionalAreas"*, forem bem-formados. Os produtos que incluem a nova *feature* já eram bem formados anteriormente, uma vez que o código já existia, apenas habilitamos a possibilidade de remover trechos de código do programa. Assim como no *template* anterior, nenhum artefato foi adicionado ou removido, a única mudança realizada foi a introdução das diretivas de compilação condicional. Sendo assim, o *template* descreve uma evolução segura, pois assim como o *template* anterior, os produtos da LPS inicial tem correspondentes sintaticamente equivalentes na LPS resultante. Isto é, os produtos que incluem a *feature* O serão exatamente iguais aos produtos da LPS inicial.

C. TRANSFORMAR MÚLTIPLOS ARTEFATOS EXISTENTES EM NOVA FEATURE

O último *template* proposto foi derivado à partir da combinação dos dois já detalhados nas seções anteriores. A Figura 9 apresenta o *template* que pode ser usado quando uma *feature* é implementada de forma composicional e anotativa. Ou seja, parte da sua implementação está modularizada em classes, e parte entrelaçada e espalhada por meio dos artefatos, sendo necessário o uso de compilação condicional para demarcar o código.

```
//#ifdef tagEvalNCLFunctionalAreas
private Set<String> getNCLFuncionalAreas(NCLFileData nclData){
     allNclFileDataMap = this.getAllNclFileDatasFromApp(allNclFileDataMap,nclData);
     Set<String> areasTotal = new HashSet<String>();

     for (NCLFileData nclFileData : allNclFileDataMap.values()) {
          FunctionalAreaData fuctionAreaData = DataBaseFacade.getInstance().
                    getFunctionalAreaData(nclFileData.getFunctionalAreaData().getId());
          List<String> areas = fuctionAreaData.
                    getFuncionalAreasName(fuctionAreaData.getFunctionalAreas());
          areasTotal.addAll(areas);
     }
     return areasTotal;
}
//#endif
```

Figura 8. Trecho de código com anotação $\#ifdef$.

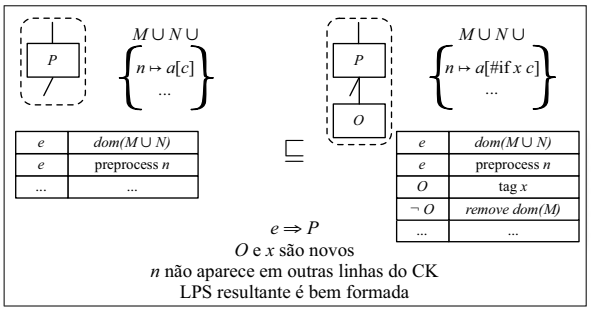

Figura 9. TRANSFORMAR MÚLTIPLOS ARTEFATOS EXISTENTES EM NOVA FEATURE

Assim como nos *templates* anteriores, introduzimos uma nova *feature* O no FM. O AM novamente tem três partes, onde uma parte (M) será associada inteiramente com a nova *feature*, como na Figura 5. Outra parte está associada com os trechos de código que serão envolvidos por diretivas, como na Figura 7. Finalmente, a terceira parte (N) está associada com outras *features* existentes.

O CK do estado inicial e final da LPS mescla as modificações dos dois *templates* anteriores. A nova *feature* O é associada à *tag* x e remove os artefatos de M por inteiro quando não é selecionada. Não é possível usar sequencialmente os outros dois *templates* já apresentados para alterar a LPS gradualmente, pois as pré-condições seriam quebradas. Por exemplo, ao usar qualquer um dos *templates* inicialmente, o segundo não poderia ser usado, pois já existiria uma *feature* O introduzida no FM. Da mesma forma que nos *templates* anteriores, os produtos da LPS inicial tem comportamento preservado pelos produtos que incluem a *feature* O na LPS resultante.

IV. AVALIAÇÃO

Os *templates* propostos foram avaliados de duas formas. Primeiro, formalizamos e provamos a corretude dos mesmos de acordo com a teoria de evolução segura de LPS [2], usando

a ferramenta *Prototype Verification System* (PVS) [9].[4] Além da verificação e prova formal, também avaliamos a utilidade e expressividade deles na prática. Nesta seção, detalhamos o estudo de caso realizado, os resultados obtidos e, finalmente, suas ameaças a validade. O objetivo deste estudo consiste em responder a seguinte questão:

Os novos *templates* para extração de *features* apoiam os desenvolvedores e preservam o comportamento da LPS durante a evolução segura?

A fim de responder esta pergunta, realizamos um estudo utilizando um sistema desenvolvido em Java e usamos os *templates* propostos durante sua transformação em uma LPS. Executamos testes antes e depois das modificações na LPS para checar a preservação de comportamento observável após aplicação dos *templates*, verificando se o resultado dos testes foram os mesmos. Ao introduzir *features*, para produtos da nova LPS sem a *feature*, é esperado que o teste relacionado falhe e os testes realizados nos produtos com a nova *feature* passem. Assim, identificamos que o comportamento do software foi preservado. Adicionalmente, com relação a preservar comportamento, reforçamos que os *templates* foram formalizados e provados de acordo com a teoria de evolução segura de LPS [2].

A. Estudo de Caso

Utilizamos um sistema familiar para os autores, com acesso ao repositório. Este sistema pertence à uma empresa da indústria de software e por questões de privacidade não pode ter seu código-fonte publicado. Sua implementação está consolidada, com todas as funcionalidades principais implementadas, com aproximadamente 400 KLOC. O sistema foi desenvolvido durante 5 anos, por uma equipe formada inicialmente por 7 desenvolvedores com experiência variando entre 1 a 4 anos. A equipe de desenvolvimento chegou a ter 12 desenvolvedores no decorrer do projeto, no entanto, o primeiro autor trabalhou na implementação desde o início do projeto.

Como o sistema escolhido não estava implementado como UMA LPS, tivemos que criar inicialmente o FM, AM e CK com apenas uma *feature* (*Root*) para gerar a primeira LPS. Neste caso, a LPS consistia de apenas um produto com todas as funcionalidades do sistema, como ilustrado nos exemplos anteriores. A Figura 10 ilustra a sequência de *features* extraídas, o *template* de transformação usado para extrair cada *feature*, e os pontos de execução de testes e verificação de resultados após cada extração. Para extração de *features*, primeiramente escolhemos 2 *features* de simples implementação para facilitar o estudo e definição dos novos *templates*. Em seguida, selecionamos algumas *features* onde a identificação e extração do código associado a essas *features* era mais fácil por experiencia do desenvolvedor com o sistema. A escolha da *feature* a ser extraída foi por familiaridade com as *features*, primeiro foram extraídas as *features* que se tinha maior certeza de onde estavam localizadas. A exceção foi a extração das *features* 3 e 4, em que os testes só

[4] https://github.com/spgroup/theory-pl-refinement/

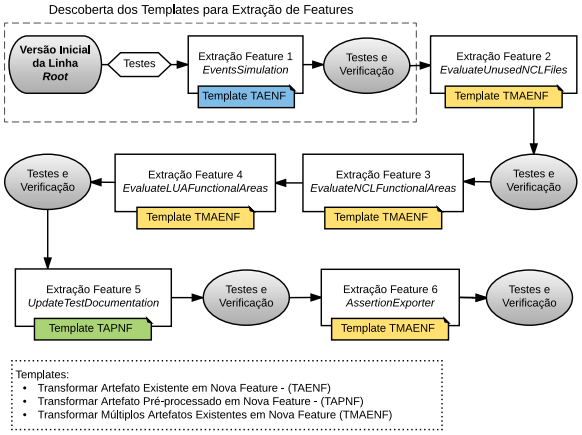

Figura 10. Evolução da nova LPS.

foram realizados após a extração de ambas as *features*. As duas *features* tinham vários artefatos em comum, portanto acreditamos que seria mais produtivo proceder com a extração de ambas, para em seguida realizar os testes e verificações apenas uma vez, considerando os produtos da LPS com e sem as *features* extraídas.

A descoberta e definição dos novos *templates* foi realizada durante a extração da primeira *feature*. Porém, foram feitas várias melhorias na especificação dos *templates* até chegar à versão final. No estudo, todos os *templates* propostos foram utilizados, como podemos ver na Figura 10. No entanto, o mais usado foi o *template* TRANSFORMAR MÚLTIPLOS ARTEFATOS EXISTENTES EM NOVA FEATURE, que adiciona *features* implementadas com contexto composicional e anotativo.

Durante a evolução da LPS foram gerados 10 novos produtos a partir das 6 *features* extraídas, todas elas *features* opcionais. Para o primeiro e o segundo produto, consideramos a ausência das *features* 1 (*EventsSimulation*) e 2 (*EvaluateUnusedNCLFiles*) respectivamente, e a presença das demais funcionalidades do sistema. A configuração dos produtos #3, #4 e #5 foi uma combinação dos possíveis produtos envolvendo as *features* 3 (*EvaluateNCLFunctionalAreas*) e 4 (*EvaluateLUAFunctionalAreas*), que tem artefatos em comum. O produto #6 foi gerado sem a *feature* 5 (*UpdateTestDocumentation*) e o produto #7 sem a *feature* 6 (*AssertionExporter*). Para gerar o produto #8 fizemos uma escolha aleatória de *features*, que resultou na seleção das *features* 2, 4 e 6, ou seja, 1, 3 e 5 estiveram ausentes deste produto. Os produtos #9 e #10 só foram gerados após realizar todas as alterações na LPS para extrair as 6 *features*. O produto #9 inclui todas as 6 *features* e o produto #10 tem todas as *features* ausentes. Todos esses produtos, bem como a quantidade de testes realizados em cada um deles e os resultados obtidos nesse estudo são apresentados na próxima seção.

Tabela I
QUANTIDADE DE TESTES EXECUTADOS POR PRODUTO

Produto	*Descrição*	*Features*	*Testes Unitários (Automáticos)*	*Testes Funcionais (Manuais)*
#0	Versão inicial da linha	Sistema original, sem alterações	500	12
#1	Sem a *feature* 1	f1( )	500	6 *
#2	Sem a *feature* 2	f1(**x**) f2( )	500	7 *
#3	Sem as *features* 3 e 4	f1(**x**) f2(**x**) f3( ) f4( )	500	6 *
#4	Sem a *feature* 3	f1(**x**) f2(**x**) f3( ) f4(**x**)	500	6 *
#5	Sem a *feature* 4	f1(**x**) f2(**x**) f3(**x**) f4( )	500	6 *
#6	Sem a *feature* 5	f1(**x**) f2(**x**) f3(**x**) f4(**x**) f5( )	500	6 *
#7	Sem a *feature* 6	f1(**x**) f2(**x**) f3(**x**) f4(**x**) f5(**x**) f6( )	500	7 *
#8	Sem as *features* 1, 3 e 5	f1( ) f2(**x**) f3( ) f4(**x**) f5( ) f6(**x**)	500	12
#9	Com todas as 6 *features*	f1(**x**) f2(**x**) f3(**x**) f4(**x**) f5(**x**) f6(**x**)	500	12
#10	Sem todas as 6 *features*	f1( ) f2( ) f3( ) f4( ) f5( ) f6( )	500	12
			5500	92

B. Resultados

Para validar o comportamento dos produtos gerados, executamos testes automáticos e manuais. Todos eles já existiam e foram usados durante a implementação das funcionalidades. Escolhemos de forma *ad-hoc*, baseado nas *features* extraídas, um conjunto de 500 testes automáticos a partir de um total de aproximadamente 9,5K testes existentes, além de alguns testes funcionais para serem executados manualmente. O mesmo conjunto de testes foi executado e analisado em todos os produtos da LPS. Os testes automáticos são considerados testes unitários e não testam todas as partes do sistema, portanto foi preciso executar outros tipos de testes a fim de coletar maior evidência de que o comportamento dos produtos existentes é preservado.

Sendo assim, a fim de verificar outras partes do sistema, executamos manualmente testes funcionais para cada produto. Neste caso, as quantidades de testes são diferentes, pois em alguns casos os testes não são necessários. Pelo menos um dos testes funcionais verifica especificamente a existência ou não da *feature* extraída de acordo com o produto. Nestes casos, pode ser que o comportamento esperado do testes seja verificar que uma determinada *feature* não esteja disponível no produto.

A Tabela I ilustra a quantidade de testes executados, bem como a configuração de cada produto gerado durante a evolução segura da LPS. Os testes executados nos produtos #0, #8, #9 e #10 são os mesmos. Para os outros produtos, selecionamos um conjunto de testes de acordo com as *features* existentes. Alguns dos produtos possuem a mesma quantidade de testes funcionais, identificados com asterisco na tabela. Embora isto possa sugerir que são os mesmos testes, isto é apenas uma coincidência, pois os testes e os resultados esperados para cada um deles são diferentes. Em alguns casos, foi necessário verificar que a *feature* ou o comportamento relacionado à *feature* não estava presente no produto gerado. Os testes foram executados em diferentes momentos da evolução da LPS, com exceção dos testes realizados nos produtos #7, #8, #9 e #10, que foram realizados quando todas as 6 *features* já haviam sido extraídas. Os resultados dos testes executados antes e depois da transformação na LPS foram comparados, e verificamos que os resultados foram compatíveis. Desse modo, a comparação de resultados nos dá certa evidência de que o comportamento dos produtos existentes foi preservado.

Durante a geração dos novos produtos e verificação da condição “LPS resultante é bem formada” existente em nossos *templates*, identificamos 2 falhas decorrentes da associação incorreta entre *feature* e artefatos. Mesmo com a validação de outro desenvolvedor, as falhas foram identificadas e corrigidas durante extração da *feature* 2 (*EvaluateUnusedNCLFiles*) e a segunda falha na extração das *features* 3 (*EvaluateNCLFunctionalAreas*) e 4 (*EvaluateLUAFunctionalAreas*). Essas falhas encontradas reforçam a necessidade de atenção às pré-condições e modificações dos elementos da LPS listadas pelos *templates*, pois produtos mal formados podem produzir comportamento não esperado.

Com exceção da verificação da condição "LPS resultante é bem formada", todos os outros testes executados foram bem sucedidos e nenhum outro defeito foi encontrado. Todos os produtos gerados preservam o comportamento dos produtos existentes, portanto podemos concluir que os nossos *templates* podem ser usados para extração de *features* durante a evolução segura de uma LPS.

C. Ameaças à Validade

Com relação à validade **interna**, são duas as principais ameaças identificadas: escolha de quais *templates* utilizar, e a associação manual entre *features* e código existente. A nossa abordagem baseia-se no fato de que evoluir manualmente uma LPS é propenso à erros. A escolha de quais *templates* utilizar depende do desenvolvedor conhecer todos os *templates* disponíveis e escolhê-los corretamente de acordo com o cenário de evolução. Uma escolha mal feita, ou uma pré-condição não verificada adequadamente, poderá introduzir erros na LPS e consequentemente alterar o comportamento dos produtos

gerados. Para reduzir essa possibilidade de erro, os três autores em conjunto revisaram a validade da aplicação dos templates.

A associação manual entre *features* e código existente também depende da experiência e conhecimento do desenvolvedor em relação ao domínio. Para extrair uma *feature* é necessário saber quais artefatos implementam determinado comportamento do sistema. No nosso estudo, para minimizar esse impacto fizemos uma validação com outro desenvolvedor, além do primeiro autor, que tinha 3 anos de experiência trabalhando no sistema. Tal solução, de fato, apesar de geral, depende da presença de um segundo especialista no domínio. Portanto, não poderá ser aplicada em outros contextos se não houver mais de um especialista. A extração de *features* em outros domínios demanda, além da presença de um especialista no domínio, um especialista em LPS. Os *templates* diminuem a necessidade do especialista em LPS. No estudo, a tarefa foi feita por um dos autores que possui conhecimento no sistema e conhecimento em LPS.

Durante a execução do estudo, ocorreram duas falhas provenientes da associação incorreta entre *feature* e artefatos, ao fazer a aplicação incorreta de um dos *templates*. Conseguimos identificar e corrigir as duas falhas na fase de verificação da restrição de boa formação presente nos *templates*. Essas falhas não comprometeram os resultados do estudo, pois foram decorrentes da aplicação manual incorreta do *template*, além de terem sido identificadas e corrigidas antes da execução dos testes. Ferramentas de suporte à aplicação dos *templates*, caso existentes, poderiam facilitar a aplicação dos mesmos, evitando erros no processo de evolução. Da mesma forma, entendemos que ferramentas podem auxiliar também no processo de localização de *features*. No entanto, como já tinhamos a presença de especialistas no sistema, não utilizamos tais ferramentas. Por outro lado, como são vários produtos analisados e comportamentos diferentes, também é possível que tenhamos negligenciado erros introduzidos pelas alterações manuais durante a evolução da LPS, e que o número de erros seja maior.

Sobre validade **externa**, devido à pequena quantidade de *features* extraídas, os resultados quantitativos não podem ser generalizado com confiança para outros projetos. Os projetos podem ter práticas de desenvolvimento e *features* com características diferentes, e como consequência, outros *templates* poderiam ter sido criados. Embora não tenhamos incluído outros projetos no nosso estudo, consideramos o sistema analisado significativo devido ao seu tamanho e complexidade, além de se tratar de um sistema real.

Finalmente, a ameaça de validade de **construto** está relacionada à escolha do subconjunto de testes, que foi o critério utilizado para afirmar que os *templates* apoiam os desenvolvedores e preservam comportamento da LPS. A escolha foi feita de forma *ad-hoc*, e por isso outros testes poderiam ter sido escolhidos. Porém, os testes selecionados foram baseados nas *features* extraídas, além do fato de que selecionamos uma grande quantidade de testes, o que nos proporcionou uma boa cobertura de cada *feature*. Os resultados qualitativos são uma evidência inicial de que o nosso conjunto de *templates* de evolução segura é expressivo para lidar com cenários de extração de LPS a partir de um produto.

V. Trabalhos Relacionados

Alves e colaboradores [10] propõem uma noção de refatoração informal para LPS, e apresentam um catálogo de refatoração para FM. Gheyi e colaboradores [11] estendem esse trabalho, propondo um catálogo completo e mínimo de *templates* de transformação de FM que preservam o conjunto de configurações válidas do FM. O nosso trabalho vai além desses, pois a noção de refinamento de LPS que usamos, e consequentemente nossos *templates*, suportam outros elementos da LPS como CK e artefatos, além do FM.

A noção de refinamento de LPS discutida aqui apareceu pela primeira vez com um foco em refatoração [6], ilustrando diferentes tipos de transformações que podem ser úteis para derivar e evoluir LPS. Borba e colaboradores [2] formalizaram esta proposta inicial, generalizando uma teoria de refinamento de LPS que pode ser utilizada com várias linguagens usadas para descrever os elementos da LPS. Eles também instanciam a teoria com a formalização de linguagens específicas para FM e CK, e introduzem e provam a solidez de vários *templates* de transformação de refinamento. Neves e colaboradores [3] propuseram novos *templates* para uma linguagem de CK diferente, com base na análise empírica da evolução de duas LPS reais, avaliando a expressividade dos *templates* antigos e novos em cinco LPS. Os nossos *templates* são diferentes dos propostos anteriormente, pois são aplicáveis em situações de extração de *features* a partir de código e funcionalidade existentes, o que não é contemplado pelos *templates* propostos anteriormente. Além disso, utilizamos uma notação de CK ligeiramente diferente.

Passos e colaboradores [12] analisam a evolução do kernel do Linux, propondo padrões de evolução semelhantes aos nossos *templates*, tendo em conta as alterações no FM (KConfig, em seu contexto), CK (Kbuild), e os artefatos com as diretivas de pré-processamento (cpp). Essas mudanças não se restringem à cenários de evolução segura, pois eles também consideram transformações potencialmente inseguras. No entanto, vemos os estudos complementares, como alguns de seus padrões relatados são semelhantes aos nossos *templates*, como Transformar artefato existente em nova feature. A principal diferença é que a nossa intenção é de fornecer garantias e apoio quando a intenção é evoluir com segurança uma LPS, enquanto eles estão interessados em relatar os cenários de evolução que acontecem em um período de tempo específico.

Thaker e colaboradores [8] definem que a composição segura está relacionada com a geração segura e a verificação das propriedades dos elementos da LPS. Eles mostraram como as propriedades de segurança da LPS podem ser verificadas usando FMs e soluções SAT. Teixeira e colaboradores [7] apresentam uma abordagem para verificar a composição segura de LPS composicional. Como os nossos *templates* requerem que a LPS resultante seja bem formada, podemos usar a

abordagem de composição segura para verificar esse tipo de condição dos nossos *templates*.

Várias abordagens [13]–[16] focam na refatoração de um produto em uma LPS, não explorando a evolução da LPS em geral, como fazemos aqui com os nossos *templates*. Kolb e colaboradores [14] discutem um estudo de caso na refatoração de componentes de código legado em uma implementação de LPS. Eles definem um processo sistemático de refatoração de produtos com o objetivo de obter os artefatos da LPS. Não há discussão sobre FM e CK. Da mesma forma, Kastner e colaboradores [13] focam apenas em transformar artefatos de código, implicitamente utilizando noções de refinamento de programas orientado à aspectos [17]. Como discutido aqui e em outros lugares [6] estes não são suficientes para justificar o refinamento de LPS, pois temos de considerar não apenas código como FM e CK, por exemplo. Trujillo e colaboradores [16] vão além de artefatos de código, mas não consideram explicitamente transformações para FM e CK como os nossos *templates* fazem. Eles também não consideram preservar o comportamento; eles de fato usam o termo "refinamento", mas no sentido bem diferente de substituição ou adição de comportamento extra para os artefatos.

VI. Conclusão

Neste trabalho, nós definimos novos *templates* para evolução segura de LPS, com o objetivo de suportar cenários de extração de *features* não suportados anteriormente. Também foi melhorada a notação existente no CK, ao invés de listar todos os artefatos associados a uma expressão de *feature*, o desenvolvedor utiliza apenas um tipo de transformação que referencia todos os artefatos da linha. Nós utilizamos os novos *templates* na prática durante a transformação de um sistema real em uma LPS, preservando o comportamento dos produtos existentes. Ao todo foram extraídas 6 *features* e, a partir delas, foram gerados 10 novos produtos. Os novos *templates* complementam a lista de *templates* existentes e podem orientar os desenvolvedores durante a evolução segura de uma LPS.

Como trabalho futuro, pretendemos usar os novos *templates* para extrair mais *features* e ampliar a quantidade de produtos do sistema utilizado nesse estudo. Também gostaríamos de estender o uso dos *templates* sugeridos em outros sistemas com características e ambientes de desenvolvimento diferentes. Um outro interesse é a construção de ferramentas de apoio à aplicação dos *templates*. Tais ferramentas poderiam inclusive definir critérios de recomendação de refatoramento para uso dos *templates* propostos.

Agradecimentos

Agradecemos a Rodrigo Bonifácio pela ajuda na ferramenta *Hephaestus* [4], e a Gabriela Sampaio e outros membros do SPG[5] pelos comentários. Agradecemos também o apoio do INES,[6] CNPq e FACEPE.

[5]http://www.cin.ufpe.br/spg
[6]http://ines.org.br

Referências

[1] K. Pohl, G. Böckle, and F. J. van Der Linden, *Software product line engineering: foundations, principles and techniques*. Springer Science & Business Media, 2005.
[2] P. Borba, L. Teixeira, and R. Gheyi, "A theory of software product line refinement," *Theoretical Computer Science*, pp. 2–30, 2012.
[3] L. Neves, P. Borba, V. Alves, L. Turnes, L. Teixeira, D. Sena, and U. Kulesza, "Safe evolution templates for software product lines," *Journal of Systems and Software*, pp. 42–58, 2015.
[4] R. Bonifácio, L. Teixeira, and P. Borba, "Hephaestus: A tool for managing product line variabilities," *III SBCARS*, pp. 26–34, 2009.
[5] S. Apel, D. Batory, C. Kästner, and G. Saake, *Feature-Oriented Software Product Lines*. Springer, 2013.
[6] P. Borba, "An introduction to software product line refactoring," in *Proceedings of the 3rd International Summer School Conference on Generative and Transformational Techniques in Software Engineering III*, ser. GTTSE'09, 2011, pp. 1–26.
[7] L. Teixeira, P. Borba, and R. Gheyi, "Safe composition of configuration knowledge-based software product lines," *J. Syst. Softw.*, pp. 1038–1053, 2013.
[8] S. Thaker, D. Batory, D. Kitchin, and W. Cook, "Safe composition of product lines," in *Proceedings of the 6th International Conference on Generative Programming and Component Engineering*, ser. GPCE '07, 2007, pp. 95–104.
[9] S. Owre, N. Shankar, J. M. Rushby, and D. W. Stringer-Calvert, "PVS Language Reference," *SRI International*, 2001. Version 2.4.
[10] V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, and C. Lucena, "Refactoring product lines," in *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, ser. GPCE '06, 2006, pp. 201–210.
[11] R. Gheyi, T. Massoni, and P. Borba, "Algebraic laws for feature models." *J. UCS*, pp. 3573–3591, 2008.
[12] L. Passos, J. Guo, L. Teixeira, K. Czarnecki, A. Wąsowski, and P. Borba, "Coevolution of variability models and related artifacts: a case study from the linux kernel," in *Proceedings of the 17th International Software Product Line Conference*. ACM, 2013, pp. 91–100.
[13] C. Kastner, S. Apel, and D. Batory, "A case study implementing features using aspectj," in *Software Product Line Conference, 2007. SPLC 2007. 11th International*. IEEE, 2007, pp. 223–232.
[14] R. Kolb, D. Muthig, T. Patzke, and K. Yamauchi, "A case study in refactoring a legacy component for reuse in a product line," in *21st IEEE International Conference on Software Maintenance (ICSM'05)*. IEEE, 2005, pp. 369–378.
[15] J. Liu, D. Batory, and C. Lengauer, "Feature oriented refactoring of legacy applications," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 112–121.
[16] S. Trujillo, D. Batory, and O. Diaz, "Feature refactoring a multi-representation program into a product line," in *Proceedings of the 5th international conference on Generative programming and component engineering*. ACM, 2006, pp. 191–200.
[17] L. Cole and P. Borba, "Deriving refactorings for AspectJ," in *Proceedings of the 4th international conference on Aspect-oriented software development*. ACM, 2005, pp. 123–134.