Pós-Graduação em Ciência da Computação

Fernando Chaves Benbassat

Evolução Segura de Linhas de Produtos de *Software*: Cenários de Extração de *Features*

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
<http://www.cin.ufpe.br/~posgraduacao>

RECIFE

2017

Fernando Chaves Benbassat

Evolução Segura de Linhas de Produtos de *Software*: Cenários de Extração de *Features*

Este trabalho foi apresentado à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre Profissional em Ciência da Computação.

Orientador: Paulo Henrique Monteiro Borba

Co-orientador: Leopoldo Motta Teixeira

RECIFE

2017

Catalogação na fonte
Bibliotecária Monick Raquel Silvestre da S. Portes, CRB4-1217

B456e Benbassat, Fernando Chaves
Evolução segura de linhas de produtos de *software*: cenários de extração de *features* / Fernando Chaves Benbassat. – 2017.
66 f.: il., fig., tab.

Orientador: Paulo Henrique Monteiro Borba.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2017.
Inclui referências e apêndice.

1. Engenharia de software. 2. Linhas de produtos de software. I. Borba, Paulo Henrique Monteiro (orientador). II. Título.

005.1 CDD (23. ed.) UFPE- MEI 2017-85

Fernando Chaves Benbassat

Evolução Segura de Linhas de Produtos de Software: Cenários de Extração de Features

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Aprovado em: 20/02/2017.

BANCA EXAMINADORA

_____

Prof. Dr. Marcio Lopes Cornélio
Centro de Informática / UFPE

_____

Prof. Dr. Rodrigo Bonifácio de Almeida
Departamento de Ciência da Computação/UnB

_____

Prof. Dr. Paulo Henrique Monteiro Borba
Centro de Informática / UFPE
(**Orientador**)

Dedico esta dissertação à minha esposa Carolline, pelo apoio incondicional e constante incentivo.

Dedico também ao meu orientador Prof. Paulo Borba, pela confiança, paciência, incentivo e excelente orientação. E também oa meu co-orientador Prof. Leopoldo Teixeira por toda ajuda e ensinamentos.

Sem apoio de todos e da minha família, este trabalho não teria sido realizado. A eles, meu muito obrigado.

Esta dissertação é dedicada ao meu filho, Miguel.

Agradecimentos

A Deus por iluminar meu caminho e me dar forças para seguir sempre em frente. À Carolline Arruda, minha esposa, pelo carinho, paciência e suporte. Ao meu pequeno Miguel por simplesmente existir. Aos meus pais, Alberto e Acacia, sempre apoiando minha qualificação profissional.

Agradeço ao projeto intitulado "Projeto Workbench 2.0", patrocinado pela Samsung Eletrônica da Amazônia Ltda. nos termos da Lei Federal nº 8.248/91 em parceria com a Universidade Federal de Pernambuco/UFPE e o Centro de Informática/CIn pela oportunidade e apoio ao trabalho realizado.

Ao meu orientador Paulo Borba e co-orientador Leopoldo Teixeixa, obrigado pela oportunidade, confiança e pelo empenho durante o mestrado. Obrigado a Rodrigo Bonifácio pelo suporte na ferramenta *Hephaestus* [1], e a todos os membros do SPG (<http://www.cin.ufpe.br/spg>) que conviveram comigo durante o mestrado. Agradeço também o apoio do INES (<http://ines.org.br>), CNPq e FACEPE.

"Your work is going to fill a large part of your life, and the only way to be truly satisfied is to do what you believe is great work. And the only way to do great work is to love what you do. If you haven't found it yet, keep looking. Don't settle. As with all matters of the heart, you'll know when you find it."

(Steve Jobs)

Resumo

Linhas de Produtos de Software (LPS) são conjuntos de produtos de *software* reutilizáveis que compartilham funcionalidades ou comportamento. Reusar um conjunto específico de produtos pode melhorar a produtividade e qualidade dos produtos oferecidos por uma empresa, no sentido de que novos produtos podem ser criados combinando de forma sistemática os artefatos existentes. Porém manter LPS não é tão simples, uma vez que uma única mudança em um artefato pode afetar vários produtos. Em muitas situações, é desejável proporcionar algum tipo de garantia para alterar uma LPS de forma segura, no sentido de que o comportamento dos produtos existentes é preservado após a alteração. Os desenvolvedores podem contar com noções de evolução segura propostas anteriormente e por meio de *templates* de transformação para assegurar a evolução segura.

No entanto, os *templates* existentes focam apenas em situações em que LPS é expandida com o desenvolvimento de novas *features*, e não foram avaliadas no contexto da extração de *features* a partir do código existente. Por isso, para descobrir mais *templates* que se adequam à situações não previstas em estudos anteriores de evolução segura de Linhas de Produtos de Software (LPS), foi realizado um estudo utilizando um sistema industrial desenvolvido em Java, com aproximadamente 400 KLOC, com demanda para extração de *features* e transformação em LPS.

Esse estudo revelou a necessidade de novos *templates* para lidar com cenários de extração de *features*, bem como melhorar a notação de *templates* existentes para tratar mapeamentos (*Configuration Knowledge*) mais expressivos entre expressões de *features* e artefatos de código. Como resultado deste estudo, nós propomos *templates* novos e que não podem ser derivados dos existentes, extraímos com sucesso LPS a partir do sistema existente usando os *templates* propostos, e também encontramos evidência de que os novos *templates* podem ajudar a prevenir defeitos durante a evolução de uma LPS.

Palavras-chave: Linhas de Produtos de *Software*. Evolução de Linhas de Produtos. Evolução Segura. Refinamento.

Abstract

Software Product Lines (LPS) are set of reusable software products that share functionality or behavior. Reusing a specific set of products can improve productivity and product quality offered by a company in the sense that new products can be created by systematically combining existing artifacts. But SPL maintenance is not simple, since a single change on asset 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 after the change. Developers can rely on previously proposed safe evolution notions and by means of transformation templates to ensure safe evolution.

However, the existing templates focus only in scenarios where a SPL is expanded with the development of new features, and have not been evaluated in the context of extracting features from existing code. Therefore, to find out more templates that fit situations not foreseen in previous studies of the safe evolution of SPL, we conducted a study using an industrial system developed in Java, with roughly 400 KLOC, with demand for features extraction and transform into an SPL.

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 (Configuration Knowledge) between feature expressions and code assets. As a result of this study, we propose new templates that can not be derived from existing ones, we successfully extracted a SPL from this existing system using the proposed templates, and also found evidence that the new templates can help to prevent defects during SPL evolution.

Keywords: Software Product Lines. Product Line Evolution. Safe Evolution. Refinement.

Lista de figuras

Lista de tabelas

Lista de Abreviações e Siglas

AM	*Asset Mapping* pp. 17–24, 26–28, 30, 33, 37, 40, 41, 44, 45, 48, 52, 60, 62
CK	*Configuration Knowledge* pp. 9, 14, 17, 18, 20, 21, 20–24, 26, 28–30, 33, 34, 37, 39–41, 44, 45, 48, 51, 52, 60–62
FM	*Feature Model* pp. 14, 17–19, 21–24, 26, 28–30, 33, 38, 40, 41, 44, 45, 48–51, 60–62
LPS	Linhas de Produtos de Software pp. 7, 9, 14, 15, 14–24, 23, 24, 26–30, 32–45, 47–52, 54, 56–62
SBCARS	X Simpósio Brasileiro de Componentes Arquiteturas e Reúso de Software p. 16

Sumário

1 INTRODUÇÃO

Linhas de Produtos de Software (LPS) permitem a geração de produtos relacionados a partir de artefatos de software reutilizáveis [4]. 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 pode melhorar a produtividade e qualidade dos produtos oferecidos por uma empresa [4]. 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 indesejada de um simples defeito em um artefato de 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 em um artefato. Para a evolução ser considerada segura, o comportamento dos produtos existentes deve ser preservado após a alteração em um artefato, isto é, cada produto da LPS inicial deve ter comportamento compatível com pelo menos um produto da nova LPS [5], conforme ilustrado na Figura 1.

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 esses, também não precisa haver correspondentes na LPS inicial.

Contudo, garantir que uma evolução é segura não é tarefa simples, pois uma mudança em um artefato pode afetar uma variedade de produtos, além do fato de que é necessário alterar diferentes artefatos (como *Feature Model* (FM) e *Configuration Knowledge* (CK), além de código e modelos) 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 [5, 3] que garantem a evolução segura. Os *templates* fornecem orientações para modificar, de forma segura, os elementos da LPS e podem ajudar a evitar

[1] No sentido de preservação de comportamento observável.

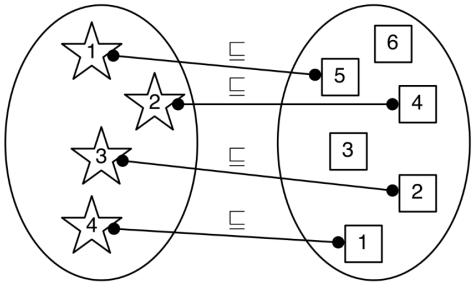

Fig. 1 – Exemplo ilustrativo do conceito de evolução segura. Elipses representam LPS e seus respectivos produtos. O símbolo ⊑ indica refinamento de programas [2].

problemas decorrentes da evolução manual de uma LPS.

Assim sendo, para reforçar a validade externa dos resultados existentes [5, 3], decidimos aplicar os *templates* em um sistema industrial desenvolvido em Java, com todas as suas funcionalidades já implementadas, durante o processo de reestruturação desse sistema em uma LPS. Até então, os *templates* existentes só tinham sido aplicados em cenários onde uma LPS cresce, expandindo funcionalidades. À vista disso, optamos por avaliá-los também no contexto de extração 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, criando *features* a partir de código existente.

Dessa forma foi constatada a necessidade de criação de novos *templates* que se adequam a situações não previstas em estudos anteriores de evolução segura de LPS, 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 esse 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 e adotada por ferramentas de gerenciamento de variabilidade como o *Hephaestus* [1]. 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 9 *features*, a partir das quais geramos 15 produtos. Note que o código referente às 9 *features* já estava implementado, porém não distinguível (isolado do resto do código da aplicação) e apenas um único produto com todas as funcionalidades estava disponível inicialmente. Para gerarmos os 15 produtos da nova LPS utilizamos a ferramenta *Hephaestus* [1], que avalia modelos de LPS e gera artefatos para instâncias específicas de uma LPS. A fim de evidenciar a evolução segura da

LPS, executamos o mesmo subconjunto de testes no produto original (sem as mudanças da LPS) e no produto completo, gerado após todas as mudanças nos artefatos e na LPS. Após verificar os resultados dos testes em ambos os produtos (original e com todas as 15 *features* presentes), constatamos que os resultados foram os mesmos, portanto o comportamento do novo produto, com todas as 15 *features*, foi preservado. Além disso, ao extrair cada *feature* executamos um subconjunto diferente de testes relacionados à *feature* extraída que, nesse caso, serviu para evidenciar que os produtos gerados que contém a *feature* apresentam o mesmo comportamento do produto original, pois os resultados dos testes executados foram equivalentes. Deste modo, colhemos evidência 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 evolução segura de LPS, 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 aproximadamente 400 KLOC, evoluindo de forma sistemática um produto em uma LPS.

Versões iniciais da definição dos novos *templates*, bem como uma parte menor do estudo e sua avaliação foram publicadas [6] no X Simpósio Brasileiro de Componentes Arquiteturas e Reúso de Software (SBCARS).[2]

O restante deste trabalho está organizado como descrito a seguir. O Capítulo 2 resume os principais conceitos utilizados para a compreensão deste trabalho, e apresenta uma visão geral da teoria de refinamento de LPS. O Capítulo 3 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 no Capítulo 4. Os resultados do estudo realizado estão no Capítulo 5, e finalmente as considerações finais no Capítulo 6, onde apresentamos os trabalhos relacionados, os trabalhos futuros e a conclusão.

[2] <http://cbsoft.org/sbcars2016>

2 EVOLUÇÃO SEGURA DE LINHAS DE PRODUTOS DE SOFTWARE

Nesse capítulo revisamos os conceitos principais utilizados neste trabalho. Primeiro, discutiremos as LPS na Seção 2.1, em seguida detalhamos os 3 elementos da LPS responsáveis pela geração automática dos produtos, o FM na Seção 2.1.1, o *Asset Mapping* (AM) na Seção 2.1.2 e o CK na Seção 2.1.3. Na Seção 2.2 discutimos o conceito de evolução segura que será fundamental para compreensão do trabalho como um todo e por fim a ferramenta *Hephaestus* na Seção 2.3, usada para gerar novos produtos da LPS.

2.1 LINHAS DE PRODUTOS DE SOFTWARE

Uma linha de produtos de *software* é definida como um conjunto de artefatos reusáveis que quando combinados geram produtos que compartilham características em comum, e são suficientemente distintos entre si. A estratégia de desenvolvimento de LPS possibilita um aumento significativo na produtividade [4]. Além disso, ao estender uma LPS com um novo produto, o tempo de mercado é reduzido, porque os artefatos de *software* existentes provavelmente correspondem a uma grande parte do novo produto. Ao manter produtos existentes, o esforço também é reduzido, pois as mudanças em um determinado artefato irá refletir sobre vários produtos. Outro ponto que deve ser considerado, é que como os artefatos de *software* são usados e testados em diferentes produtos e contextos, a qualidade do mesmo também pode ser melhorada [7].

Com os benefícios de produtividade fornecidos pelo uso de LPS é possível gerar milhares de produtos, com características e comportamentos diferentes. Contudo, os erros cometidos ao modificar um artefato de *software* podem ser difíceis de detectar manualmente, já que esse artefato é usado por diversos produtos e em contextos diferentes. Talvez a correção em um determinado artefato de *software* ou produto não sirva para todos os casos onde ele é usado, pois as características de ambiente de *software* são distintas.

LPS tem se tornado um importante paradigma de desenvolvimento de *software*, pois permite às empresas otimizar os seus processos de engenharia. A ideia de criar *software* a partir de partes reutilizáveis é discutida há bastante tempo, inclusive na engenharia de *software* convencional, onde as organizações armazenam em seus repositórios, arquivos e estruturas de *software* para serem reutilizados como bibliotecas em vários sistemas de diferentes segmentos. Contudo, esses repositórios não são planejados, sistematizados, estruturados e organizados para facilitar uma abordagem orientada ao reuso em larga escala. Assim, os desenvolvedores ficam procurando neles fragmentos de *software* ou algoritmos

para reusar e isto demanda tempo. Além disso, no desenvolvimento de *software* convencional o reuso contempla apenas pequenas partes de código, onde o mesmo é reutilizado em poucas partes do sistema. Por outro lado, na abordagem de LPS, a reutilização é planejada e executada de forma sistemática, todo o código implementado é concebido tendo em conta a possibilidade de reutilização nos diversos produtos que serão gerados pelas LPS.

Ao usar a estratégia de LPS é possível variar alguns ou até todos os requisitos de *software* dependendo dos pontos de variação definidos na fase inicial do ciclo de vida da LPS. Ao longo de todo o processo de desenvolvimento os vários pontos de variação são analisados, e são escolhidos os comportamentos necessários para a concretização do produto final de *software*. Em resumo, esses benefícios podem ser atribuídos à reutilização estratégica do *software*. Assim o único real esforço de engenharia necessário para a concepção de cada produto da LPS está relacionado às variações que são realmente únicas, e à devida seleção de comportamentos característicos de cada ponto de variação.

Para caracterizar os produtos de uma LPS utilizamos as *features*, que representam a variação em uma LPS. Elas podem ser definidas como requisitos reutilizáveis ou características de uma LPS [8]. Uma LPS é representada aqui como uma tripla de elementos: o FM, que descreve as *features* e as dependências entre elas; o AM, que relaciona nomes de artefatos e os artefatos diretamente; e o CK, que mapeia expressões de *features* para nomes de artefatos. Para explicar os conceitos apresentados a seguir, parte dos textos foram adaptados de Borba, Teixeira e Gheyi[5].

2.1.1 FEATURE MODEL

Um FM é geralmente representado como uma árvore, composta de *features* e organizadas em modelos, sendo a representação mais comum baseada na abordagem *Feature-Oriented Domain Analysis* (FODA) [8]. Esse modelo consiste em um diagrama detalhando uma decomposição hierárquica de *features* e informações sobre como elas se relacionam entre si. As *features* têm nomes distintos e grupos abstratos de requisitos associados, com uma notação específica para cada tipo de relação entre elas:

- **Obrigatória**, presente em todas as configurações de produto onde a *feature* pai estiver selecionada (círculo cheio);
- **Opcional**, pode ou não estar presente em um produto (círculo vazio);
- **Alternativa**, grupo de *features* mutuamente exclusivas, pelo menos uma *feature* e apenas uma *feature* pode ser selecionada (arco vazio);
- ***Or***, grupo de *features* onde é possível selecionar uma ou mais *features* (arco cheio).

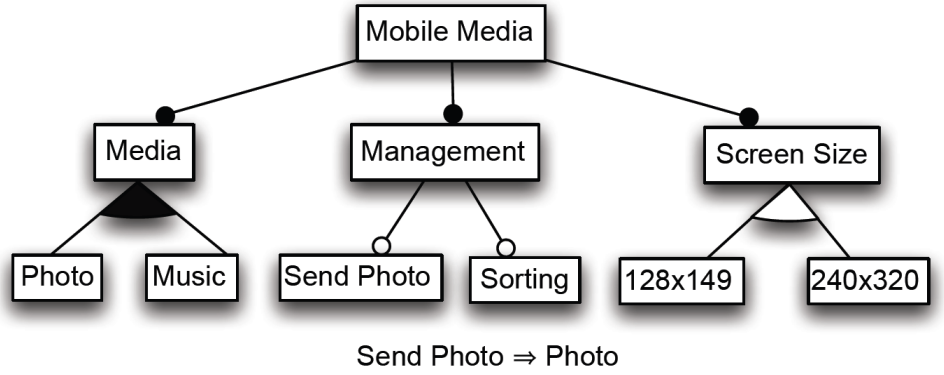

Fig. 2 – Exemplo de *Feature Model* [3]

O produtos gerados pela LPS serão combinações de *features* possíveis de acordo com o FM, onde a configuração de um produto será formado pela seleção da *feature*. As configurações de produtos podem ser válidas ou não. Uma configuração de produto válida é quando as *features* selecionadas satisfazem todas as restrições do FM, especificadas graficamente e por meio de fórmulas. A Figura 2 mostra um exemplo do FM *Mobile Media*, apresentado por Neves et al.[3], onde as *features* **Sorting** e **Send Photo** são opcionais, as *features* **Media**, **Management** e **Screen Size** são obrigatórias, **Photo** e **Music** são *features-Or* e as duas *features* filhas da *feature* **Screen Size** são alternativas. A expressão logo abaixo da árvore do FM, afirma que a *feature* **Photo** deve estar sempre presente no produto em que a *feature* **Send Photo** for selecionada, caso contrário, o produto gerado será inválido. Um exemplo de produto inválido é a seleção das *features*

{*Music*, *Send Photo*, 240x320},

pois essa seleção de *features* invalida a condição exibida abaixo da árvore do FM (**Send Photo** ⇒ **Photo**). Para tornar válida essa seleção de *features*, é necessário adicionar também a *feature* **Photo**, tendo como resultado a configuração válida

{*Music*, *Photo*, *Send Photo*, 240x320}.

Outro exemplo de configuração válida seria

{*Music*, *Photo*, 240x320},

juntamente com as *features* obrigatórias **Media**, **Management** e **Screen Size**.

Em resumo, uma configuração de produto é uma seleção de *features* válida que satisfaz todas as restrições do FM, especificadas graficamente e por meio de fórmulas. Cada configuração de produto corresponde a um produto na LPS, expresso em termos das *features* que suporta. Isto captura a intuição de que um FM denota o conjunto de produtos de uma LPS [9].

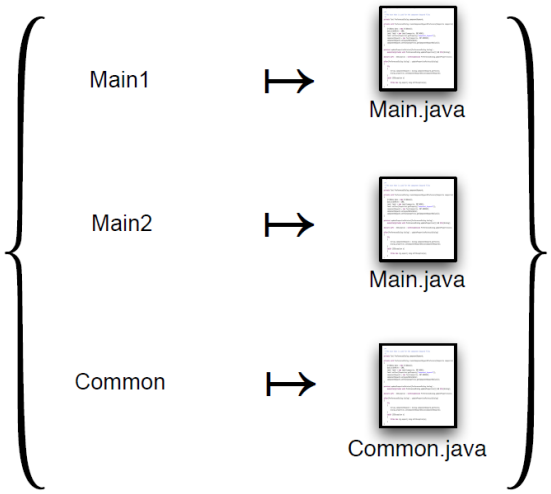

Fig. 3 – Exemplo de *Asset Mapping* [3]

2.1.2 ASSET MAPPING

Em uma LPS, especificamos e implementamos *features* com artefatos reutilizáveis, tais artefatos de software são listados no AM, onde associamos o artefato real a um nome de artefato. Esse mapeamento de artefatos basicamente lista todos os artefatos da LPS, que especificam e implementam as *features*, como documentos de requisitos, modelos de design, código, modelos, testes, arquivos de imagem, arquivos XML e assim por diante. Portanto, o AM é independente de linguagem de programação e qualquer tipo de arquivo pode ser mapeado.

Esse mapeamento de artefatos (AM) basicamente corresponde a um ambiente de declarações de artefatos, onde apenas um nome de recurso pode ser mapeado para um artefato real. No exemplo da Figura 3, temos as duas classes *Main.java* com o mesmo nome real do artefato, porém são arquivos diferentes, localizados em pastas distintas e associados a diferentes nomes de recurso. Isso serve para impedir referências ambíguas a artefatos reais.

2.1.3 CONFIGURATION KNOWLEDGE

O CK compreende o mapeamento entre as expressões de *features* e sua implementação. Logo, todos os artefatos de software que constituem uma determinada *feature* deverão, de alguma maneira, constar no CK. Os nomes das *features* são usados, nas fórmulas proposicionais, como átomos para simbolizar a condição de presença [10]. Assim, a negação de uma *feature* indica que ela não deve ser selecionada, caso contrário ela será selecionada e deverá fazer parte do produto final.

A Figura 4 apresenta uma parte do CK da LPS do *Mobile Media*, onde prescreve que, quando as *features* **Photo** e **Music** forem selecionadas, o aspecto *AppMenu.aj* deverá estar presente, bem como os arquivos *Photo.java*, *Music.java*, *Common.aj*, entre outros

Mobile Media	MM.java, ...
Photo	Photo.java, ...
Music	Music.java, ...
Photo ∨ Music	Common.aj, ...
Photo ∧ Music	AppMenu.aj, ...
⋮	⋮

Fig. 4 – Exemplo de CK [3]

não exibidos, deverão fazer parte do produto final. Nesse exemplo da Figura 4, o aspecto *AppMenu* não deve estar presente em produtos que possuem apenas uma das *features* **Media**. Isto é precisamente o que especifica a quinta linha do CK simplificado da LPS do *Mobile Media*. Da mesma forma, as implementações **Photo** e **Music** compartilham alguns recursos, portanto, a quarta linha evita repetir os nomes dos artefatos na segunda e terceira linhas. Dada uma configuração de produto válida, por exemplo a configuração

{*Photo*, 240x320},

a avaliação do CK produz os artefatos que constituem um produto correspondente com os seguintes artefatos reais associados

{*MM.java,. . . , Photo.java,. . . , Common.aj,. . .*}.

Finalmente, para assegurar que o CK está bem formado, é preciso que os nomes das *features* estejam presentes no FM e os nomes de artefatos presentes no AM. Dado que não existem problemas nos 3 elementos da LPS, o processo de geração de um produto com notação de CK composicional consiste em:

- (i) Escolher uma configuração de produto válida no FM;
- (ii) Avaliar o CK, ou seja, validar as expressões de *features* e assim verificar quais nomes de artefatos serão processados;
- (iii) Separar os artefatos reais de software do AM de acordo com a avaliação do item anterior;
- (iv) Gerar o novo produto com as *features* escolhidas.

2.2 EVOLUÇÃO SEGURA

Nosso estudo é baseado na evolução segura de LPS e para nos ajudar a identificar os cenários de evolução segura, devemos considerar a noção de refinamento de LPS [5].

Esse conceito é fundamentado na noção de refinamento de programa [2], onde é possível comparar os artefatos do programa em relação à preservação do seu comportamento. Por exemplo, ao comparar duas versões distintas de um programa: na versão mais recente novas funcionalidades foram adicionadas, porém o comportamento do programa para as funcionalidades que existiam na versão mais antiga permanece o mesmo, logo podemos assegurar que o comportamento do programa foi preservado.

Como já mencionamos na seção anterior, nesse trabalho consideramos que uma LPS consiste em um FM, um CK, e um mapeamento de artefatos (AM) que ao serem combinados, geram produtos, ou seja, conjuntos de artefatos bem formados. A noção de boa formação pode ser instanciada pela conhecida noção de verificação de tipos existente na maioria das linguagens de programação.

Semelhante ao refinamento do programa [2], o refinamento de LPS também preserva o comportamento. No entanto, ele vai além do código-fonte e outros tipos de artefatos reutilizáveis, e pode também transformar o FM e o CK. Em um refinamento de LPS, a LPS resultante deve ser capaz de gerar produtos que correspondem comportamentalmente aos produtos das LPS originais. Assim, os usuários de um produto original não podem observar diferenças comportamentais ao usar as mesmas funcionalidades do produto correspondente na nova LPS. Isso é exatamente o que garante a preservação do comportamento do produto ao melhorar um *design* na LPS, por exemplo.

Após as alterações na LPS, se as mesmas forem feitas considerando as noções de refinamento citadas, todos os produtos da LPS original devem ter pelo menos um produto correspondente na LPS resultante. Isso não significa que a quantidade de produtos antes e depois deve ser a mesma, podemos ter menos produtos na LPS resultante, desde que todos os produtos da LPS sejam refinados. Uma situação seria ter dois produtos A e B na LPS original que são refinados por um produto C na LPS resultante. Nesse caso, temos menos produtos na LPS resultante e ainda assim é considerado refinamento, pois o produto C contempla o comportamento dos dois produtos A e B, caso contrário a evolução é declarada não-segura. Quando estendemos a LPS sem impactar os usuários existentes, evidenciamos que a transformação é segura. Basicamente, cada produto (conjunto de artefatos válidos) gerado pela LPS original deve ser refinado por algum produto da LPS recentemente melhorada.

Representamos o conjunto de todas as configurações de produtos válidas correspondente à semântica de um *Feature Model* como $[[F]]$, a função semântica de um *Configuration Knowledge* como K, um *Asset Mapping* como A e uma configuração de produto c como $[\![K]\!]_c^A$. Essa é a função que gera produtos, usando uma configuração, o CK e o AM.

Definição 1: Linha de Produtos
Para um *Feature Model* F, um *Asset Mapping* A e um *Configuration Knowledge* K, dizemos que a tupla

$$(F, A, K)$$

é uma linha de produtos quando, para todo $c \in [\![F]\!]$, desde que

$$wf([\![K]\!]_c^A)$$

Essa definição estabelece que, para uma configuração c, *Configuration Knowledge* K e *Asset Mapping* A relacionado a uma determinada LPS, $[\![K]\!]_c^A$ é um conjunto bem-formado de artefatos. A restrição de boa formação (wf) é necessária pois caso falte uma entrada no CK, por exemplo, um conjunto de artefatos poderá ser impactado, algumas partes necessárias do produto não estarão presentes e, portanto, os produtos gerados serão inválidos. Do mesmo modo, um erro ao escrever uma entrada no CK ou AM pode gerar um conjunto de artefatos inválidos devido a conflitos em artefatos. Logo, é exigido que os elementos da LPS sejam coerentes e válidos.

Na maioria dos cenários de refinamento de LPS, muitas mudanças precisam ser aplicadas aos artefatos de código, FM e CK, que muitas vezes levam a LPS refinada a gerar mais produtos do que antes. Enquanto são gerados produtos que refinam todos os produtos da LPS original, os usuários não têm razão para reclamar, pois o comportamento dos produtos existente são preservados. Para estender de forma segura a LPS foram formalizadas essas restrições em termos de refinamento de programas (conjuntos de artefatos). Basicamente, cada produto (conjunto de artefatos válidos) gerado pela LPS original deve ser refinado por algum produto da LPS resultante, conforme definido abaixo.

Definição 2: Refinamento de Linha de Produtos
Para as linhas de produtos (F, A, K) e (F', A', K'), essa última refina a primeira, representada por

$$(F, A, K) \sqsubseteq (F', A', K')$$

sempre que

$$\forall c \in [\![F]\!] \cdot \exists c' \in [\![F']\!] \cdot [\![K]\!]_c^A \sqsubseteq [\![K']\!]_{c'}^{A'}$$

Lembrando que, para uma configuração c, o *Configuration Knowledge* K, e o *Asset Mapping* A relacionado a uma determinada LPS, $[\![K]\!]_c^A$ é um conjunto bem-formado de artefatos. Portanto, $[\![K]\!]_c^A \sqsubseteq [\![K']\!]_{c'}^{A'}$ refere-se ao refinamento do conjunto de artefatos. A definição que acabamos de mencionar, que representamos usando o símbolo $\sqsubseteq$ (também usado para representar refinamento de programas), refina a LPS, portanto, todos os produtos na nova LPS também devem ser bem formados. A relação de refinamento de LPS é uma pré-ordem: é reflexiva e transitiva. Isto é o que nos permite compor diferentes *templates* de evolução segura e também ter uma LPS bem-formada e válida. A definição também é composicional, no sentido de que refinar um FM, AM ou CK que fazem parte de uma LPS válida produz uma LPS refinada e válida. A propriedade de composicionalidade

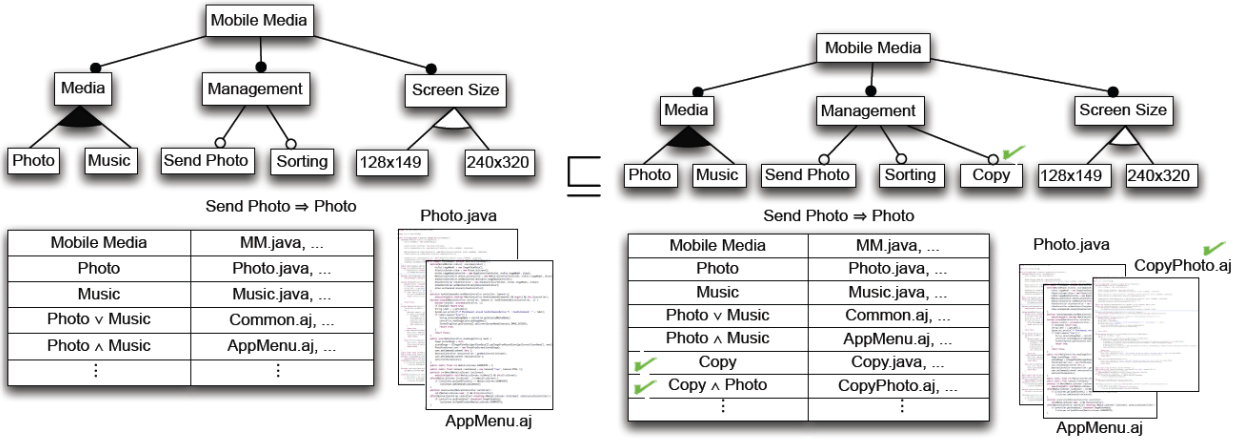

Fig. 5 – Exemplo: Adicionando uma *feature* opcional na LPS *Mobile Media* [3]

é essencial para garantir o desenvolvimento independente destes artefatos em uma LPS.

Essa definição de refinamento de LPS pode ser melhor entendida com o exemplo simplificado da LPS *Mobile Media*, ilustrado na Figura 5. Nesse exemplo, estamos adicionando a *feature* opcional **Copy**, logo a nova LPS gera duas vezes mais produtos do que a LPS original. Nesse caso, a metade dos novos produtos gerados - os que não têm a *feature* **Copy** - se comportam exatamente como os produtos originais porque os artefatos existentes na LPS permanecem inalterados. Isso garante que a transformação é segura; Nós estendemos a LPS sem impactar usuários existentes.

2.3 HEPHAESTUS

A ferramenta *Hephaestus* [1] tem como principal funcionalidade gerar artefatos de instâncias específicas de LPS. Em sua versão atual, ela apoia o gerenciamento de variabilidade em LPS com diferentes tipos de artefatos, como cenários de caso de uso, código fonte e processos de negócio. O *Hephaestus* permite aos usuários selecionarem os artefatos e elementos da LPS (FM, AM e CK) de forma simples para o processo de derivação de produtos. A ferramenta avalia esses dados de entrada e então gera os produtos da LPS.

Existem diferentes representações de CK, das quais a mais simples basicamente mapeia uma *feature* a um artefato da LPS. O processo de derivação de produtos do *Hephaestus* avalia cada expressão de *feature* do CK a partir das *features* selecionadas do FM. No CK, todas as transformações associadas às expressões de *features* avaliadas como verdadeiras são aplicadas aos artefatos da LPS selecionada para o processo; essas transformações selecionam, removem ou modificam partes dos artefatos para o produto que está sendo gerado.

O *Hephaestus* também implementa a transformação *selectAllComponents* que inclui

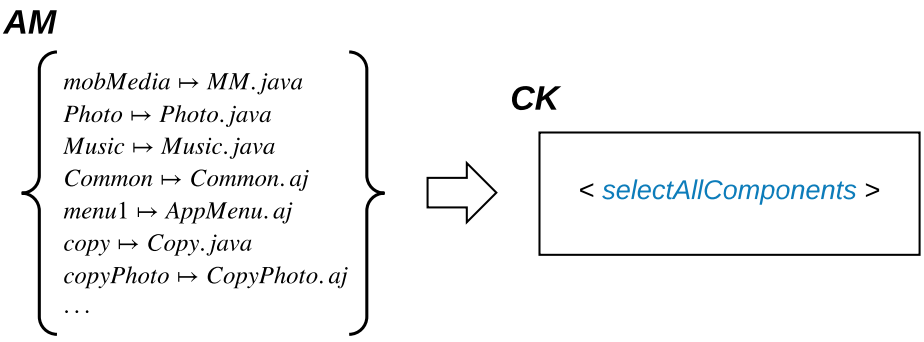

Fig. 6 – Exemplo da transformação *selectAllComponents* da ferramenta *Hephaestus* [1]

todos os artefatos das LPS existentes em um diretório raiz indicado em um arquivo de propriedades da ferramenta, sem a necessidade de listar todos os artefatos no AM. Essa transformação facilita bastante a manutenção de LPS, pois minimiza o trabalho manual de informar todos os artefatos no CK, como também reduz a possibilidade de erros durante a atualização manual de LPS. A Figura 6 exemplifica a transformação *selectAllComponents* para a LPS do *Mobile Media*, substituindo todos os artefatos existentes no AM. Vale ressaltar que a transformação *selectAllComponents* não elimina a necessidade de listar no AM os artefatos associados à outras transformações.

3 MOTIVAÇÃO

Diversos cenários de evolução exigem a alteração de mais de um elemento da LPS. Por exemplo, para adicionar uma nova *feature* opcional é necessário modificar e combinar corretamente os 3 elementos da LPS conforme apresentado na Figura 7, onde consideramos um sistema industrial desenvolvido em Java que possui duas funcionalidades principais com foco para o sistema de TV Digital brasileira, a geração de aplicações a partir de regras do modelo e a análise sintática e semântica de aplicações. Na figura, a tripla (F, A, K) representa o FM, o AM e o CK, respectivamente. Uma simples mudança em um desses elementos pode impactar diretamente vários produtos gerados, e consequentemente, seu comportamento.

A Figura 7 detalha o cenário. Na parte de cima da imagem a tripla (F, A, K) denota a LPS na versão inicial, que gera um produto contendo apenas a *feature Root* e formado por todos os arquivos do AM.[1] 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*, a partir de código já existente. 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 alguns arquivos da LPS, que são alguns dos arquivos existentes que serão relacionados à nova *feature* e outros não. As reticências indicam que podem existir outros artefatos na lista. O CK relaciona *Root* a todos os arquivos do sistema, inclusive aos 5 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*. Ao analisar a *feature EventsSimulation*, identificamos que parte da sua implementação estava modularizada em classes, e outra parte, responsável por chamar o processo de simulação de eventos, estava entrelaçada em outro artefato não relacionado à nova *feature*, sendo necessário o uso de compilação condicional para demarcar o código. Adicionamos a marcação *#ifdef* para compilação condicional no artefato *LuaBehavioralSimulator.java* com a *tag tagEvtSimulation*. No CK, foi preciso adicionar 3 novas linhas:

- A primeira linha adicionada (em azul) com o lado esquerdo contendo a *feature Root* e do lado direito o mecanismo de pré-processamento (*preprocess*) para compilação condicional do trecho de código existente no artefato *file5*;

[1] Todo programa pode ser representado como uma LPS desta forma.

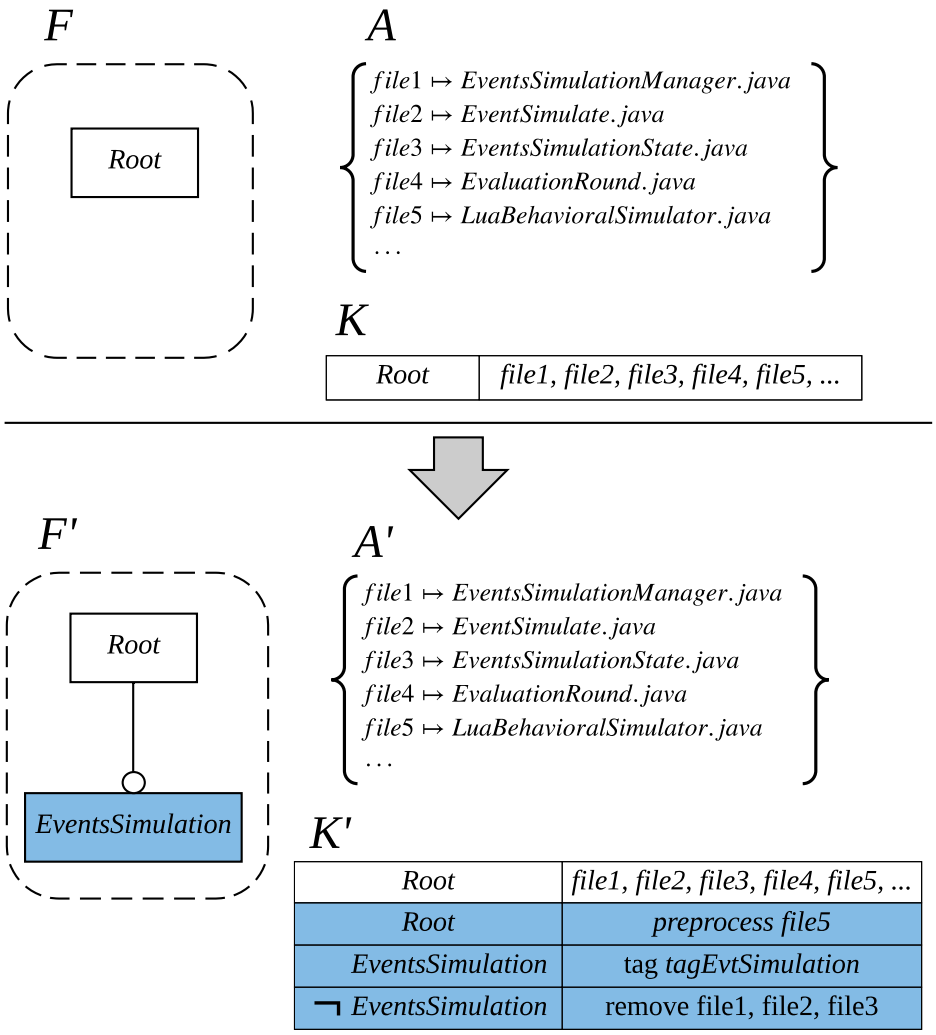

Fig. 7 – Exemplo: Extrair *feature* opcional a partir de código existente

- A segunda linha (em azul) com a nova *feature EventsSimulation* do lado esquerdo e do direito a *tag* de pré-processamento *tagEvtSimulation*, indicando que ao selecionarmos a *feature*, o código envolvido por esta *tag* deve ser incluído no projeto;
- Na última linha, adicionamos uma expressão lógica denotando a ausência da nova *feature* do lado esquerdo e, do lado direito, associamos a ação de transformação *remove* aos artefatos completos associados à nova *feature*. Assim, o produto que não inclui a *feature tagEvtSimulation* não tem os arquivos *file1*, *file2* e *file3*, que passam a estar associados à presença da *feature*.

Desta forma, o produto gerado com a nova *feature EventsSimulation* terá o 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. A configuração {*Root*} na LPS original é equivalente a configuração {*Root*, *EventsSimulation*} na nova LPS. Já a configuração {*Root*} na nova LPS será um novo produto, sem a *feature*

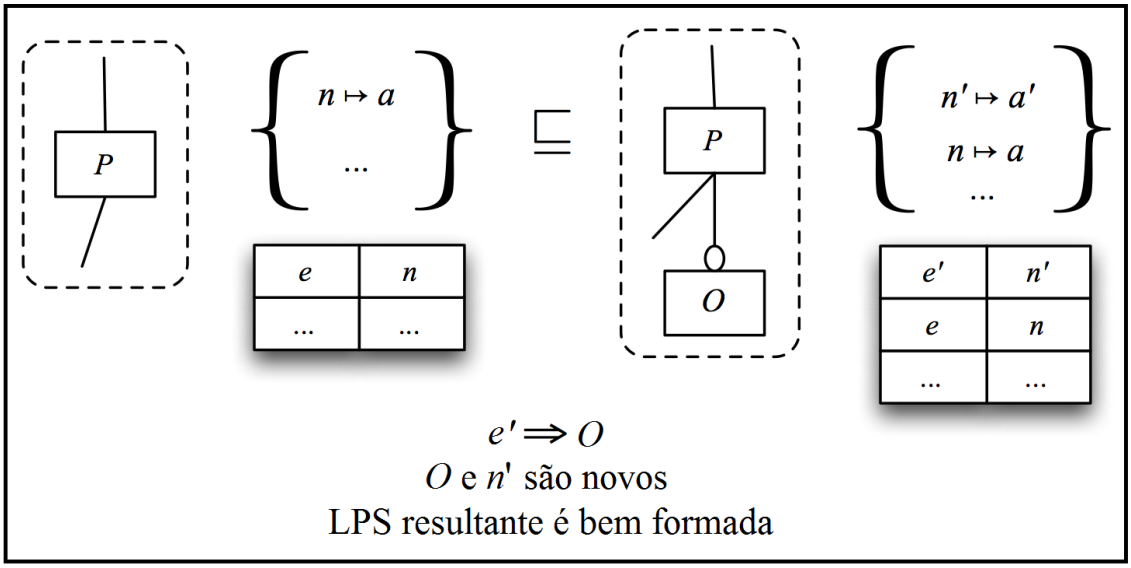

Fig. 8 – Adicionar nova feature opcional [3]

EventsSimulation e não compatível com nenhum produto existente antes das alterações na LPS. Nesse produto, sem a nova *feature EventsSimulation*, os arquivos *file1*, *file2* e *file3* não estão presentes. Note que embora estejamos adicionando uma nova feature, nenhum novo artefato foi adicionado no AM, visto que estamos extraindo essa *feature* a partir do código existente. Além disso, para facilitar o entendimento, estamos mostrando apenas uma pequena parte dos artefatos relacionados à *feature EventsSimulation*.

Evoluir manualmente uma LPS e sem orientação dada pelos *templates* é propenso a erros, como associar incorretamente expressões de *features* a nomes de artefatos no CK [3]. Deste modo, no exemplo do capítulo anterior (ver Figura 5), ao adicionar a nova *feature* opcional **Copy**, o artefato *CopyPhoto.aj* poderia ter sido incluído incorretamente na linha do CK que contém apenas a expressão de *feature* **Photo** (linha 2), sem a *feature* **Copy** associada. Consequentemente, produtos que possuem apenas a *feature* **Photo** presente (sem a *feature* **Copy**), seriam gerados inválidos ou com problema de boa formação. Além disso, os erros introduzidos durante a evolução manual da LPS podem ser difíceis de controlar porque eles estão presentes apenas em certas configurações de produtos. Como nesse caso, apenas os produtos gerados com *feature* **Photo** e sem a *feature* **Copy** poderiam apresentar problemas ou ter seu comportamento alterado.

Para reduzir os problemas durante a evolução de uma LPS, desenvolvedores podem usar *templates* de transformação de LPS [5, 3], que descrevem mudanças nos artefatos da LPS que são consistentes com o conceito de evolução segura. A Figura 8 apresenta um dos *templates* já existentes, usado para adicionar uma nova *feature* opcional sem alterar os artefatos existentes e proposto por Neves et al.[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 [5] (ver Seção 2.2), que é a formalização de evolução segura. Cada lado do *template* possui um FM, representado pelo retângulo com as linhas tracejadas, um AM, pelo conteúdo entre as chaves, e um CK, pela tabela com duas colunas. O *template* indica que podemos introduzir uma nova *feature* opcional O junto com um novo artefato a', desde que este artefato só esteja incluído em produtos com O. O FM mostra as meta-variáveis que explicam a adição da nova *feature* opcional O. Representamos o fato de que uma *feature* pode ter outras *features* relacionados a ela usando uma linha acima ou abaixo dela. Além disso, detalhamos a primeira linha do CK, que associa e com n, para ilustrar que os itens existentes no CK permanecem inalterados após a transformação. Na parte de baixo da figura estão as pré-condições para aplicar o *template*, estabelecendo que não podemos ter outra *feature* chamada O no FM, nem outro nome de artefato n' no AM, pois isso levaria a artefatos inválidos. O *template* também exige que a LPS resultante seja bem formada, para garantir que independentemente de linguagem envolvida, o novo artefato a' deve compor corretamente com outros artefatos nos novos produtos. Isso evita artefatos com nomes conflitantes em um produto, já que na maioria das linguagens isso causa erros de compilação. Podemos verificar a restrição de boa formação usando a abordagem de composição segura [11, 12]. Em alguns *templates* essa restrição não é exigida explicitamente, isso porque podemos deduzi-la das restrições do próprio *template* e outras pré-condições. Nesse *template*, a LPS resultante tem os mesmos produtos que tinha antes, além de produtos que contêm a nova *feature* O, logo, o conjunto de produtos possíveis é aumentado.

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. Por exemplo, se a nova *feature* a ser adicionada fosse uma *feature* obrigatória, o *template* apresentado não poderia ser usado para evoluir de forma segura a LPS, pois o tipo da nova *feature* não estaria de acordo com o *template*. Entenda que, se adicionarmos uma nova *feature* obrigatória, seguindo as orientações do *template* e adicionando também novos artefatos à LPS, o comportamento dos produtos existentes não será preservado na nova LPS, pois todos os novos produtos vão ter a nova *feature*.

Além desse *template* apresentado, existe na literatura um conjunto de *templates* de evolução segura disponível que podem ser usado por desenvolvedores responsáveis pela manutenção de LPS [3].[2] Tais *templates* foram propostos a partir da análise de parte da história de evolução de cinco LPS, onde observou-se que os *templates* poderiam abordar as modificações seguras que os desenvolvedores realizaram nos cenários analisados. Também foi observado que, se os *templates* tivessem sido usados como guia ao evoluir as LPS,

[2] Catálogo de *templates* para evolução segura de LPS:
<https://github.com/spgroup/pl-refinement-templates-catalog/blob/master/templatescatalog.pdf>

poderiam ter ajudado a evitar os erros de evolução que foram identificados durante o estudo. Esses erros são resultantes de modificações que deveriam ser seguras, mas que de fato alteraram o comportamento de produtos existentes. Os resultados também mostram evidências de que a evolução manual da LPS é propensa a erros, pois geralmente envolve a análise e modificação de muitos artefatos de código-fonte, além do FM e do CK.

No entanto, esses *templates* só foram aplicados em LPS de pequeno e médio porte, desenvolvidas para fins de pesquisa e educação, de modo que os cenários identificados em sua evolução poderiam ser simples e exigir o uso de poucos *templates* [3]. 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, com objetivo de transformá-lo em uma LPS. Para isso, iniciamos analisando o cenário apresentado Figura 7, no qual gostaríamos de associar alguns artefatos de código-fonte do sistema a uma nova *feature* opcional. Tentamos utilizar o *template* da Figura 8 para adicionar a *feature* opcional *EventsSimulation* no sistema. Ao tentar aplicar o *template*, verificamos que algumas partes dele estão de acordo com as mudanças necessárias para extrair a *feature* e outras não. Como a *feature* que estamos adicionando é uma *feature* opcional, nesse caso, o que diz respeito ao FM no *template* está de acordo, pois do lado direito da imagem temos a representação da nova *feature* opcional O, que seria substituída pela *feature* opcional *EventsSimulation*. Por outro lado, o AM e o CK do *template* não estão apropriados, pois ambos adicionam n' que está associado a um novo artefato a' no AM. Como todo o código e artefatos de software relacionados à *feature EventsSimulation* já existem no sistema (*file1*, *file2*, *file3* e *file5*), nenhum artefato novo deverá ser adicionado ao AM. Portanto, como algumas alterações na LPS não coincidem com as orientações apresentadas pelo *template* Adicionar nova feature opcional [3], não podemos usá-lo para evoluir de forma segura a LPS.

Deste modo, após analisar a lista de *templates* disponíveis na literatura [3], verificamos que nenhum deles atendia o nosso cenário de evolução. Para extrair a *feature* é necessário adicioná-la ao FM, porém constatamos que alguns *templates* não alteram o FM, como os *templates Add Harmless Preprocessing Directive*, *Preprocess Asset Without Preprocessor Directive*, *Add Dead Preprocessed Code*, *Replace Feature Expression* e *Add Unused Assets*, então esses não seriam aplicáveis. Por outro lado, o *template Add any feature without changing the AM and CK*, modifica o FM, não altera o AM, mas também não altera o CK necessário para as mudanças requeridas para extrair a *feature EventsSimulation*.

A maior parte dos *templates* existentes [3] foram propostos a partir da análise histórica da evolução de duas LPS e as operações observadas geralmente envolvem um grande número de classes modificadas e adição de novas funcionalidades. Como os cenários de evolução não são os mesmos, não conseguimos usar os *templates* existentes para 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, no próximo capítulo apresentamos em detalhe os *templates* propostos.

4 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 extrair *features* a partir de artefatos e partes de artefatos existentes. 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 [13]. 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. Injeção de dependência (*Dependency Injection*, em inglês) e *plugins* do *eclipse*[1] são exemplos de cenários composicionais, onde o nível de acoplamento entre módulos do sistema é baixo.

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, e envolvidos com anotações *#ifdef*, para incluir ou não o trecho de código no arquivo resultante. Nesse caso, é necessário o uso de algum mecanismo de pré-processamento, como compilação condicional para gerar os arquivos a partir das anotações dentro de cada artefato [14]. Também podemos ter os dois contextos, composicional e anotativo, utilizados em combinação para implementar uma mesma *feature*. Nesse caso, a *feature* é composta por artefatos completos e outra parte espalhada em trechos de artefatos que podem pertencer a outras *features*, conforme exemplo (ver Figura 5) apresentado no Capítulo 3.

Para descobrir mais *templates* que se adequam à situações não previstas em estudos anteriores de evolução segura de LPS, utilizamos um sistema industrial familiar, com todo o código obrigatório já implementado, e com demanda para extração de *features* e transformação em uma LPS. 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*). Em seguida, percebemos que a *feature* escolhida envolvia classes completas e um pequeno trecho de código, responsável por iniciar a simulação de eventos, entrelaçado em um artefato não pertencente à *feature* a ser extraída, portanto uma combinação dos dois contextos, composicional e anotativo.

Antes de propor novos *templates*, tentamos usar sequências de *templates* existentes,

[1] <https://eclipse.org/>

aplicando pequenas mudanças e evoluindo a LPS por etapas até conseguir adicionar a *feature* desejada. Tentamos extrair a *feature EventsSimulation* usando a seguinte sequência de *templates* existentes:

1. *Template: Add any feature without changing the AM and CK;*
2. *Template: Merge items with propositionally equivalent feature expressions;*
3. *Template: Preprocess Asset Without Preprocessor Directive;*
4. *Template: Add Harmless Preprocessing Directive;*
5. *Template: Merge items with equivalent feature expressions by the FM;*
6. *Template: Convert Mandatory to Optional;*

Poderíamos fazer isso devido à transitividade da noção de refinamento de LPS [14]. Como cada *template* é provado como um refinamento de LPS, isso nos permite aplicá-los sequencialmente conforme apresentado na Figura 9. Só temos de garantir que a LPS resultante do primeiro *template* coincide com a LPS original do segundo *template*. Contudo, a sequência de *templates* apresentada não serviu para ser usada para a extração da *feature EventsSimulation*, pois as regras e pré-condições existentes nos *templates* não estavam de acordo com o cenário de extração, bem como os *templates* existentes não suportavam a notação de CK adotada pela ferramenta *Hephaestus*, usada para gerar os produtos da LPS. Ao usarmos a transformação do *Hephaestus* para selecionar todos artefatos da LPS, como consequência, precisamos usar a transformação que remove os artefatos quando a *feature* não estiver presente no produto. Como os *templates* usados na sequência apresentada não suportam essas transformações, não pudemos usá-los para extrair 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 8. Partimos do princípio que a estrutura da mudança no FM deve ser a mesma, isto é, a adição de uma nova *feature O*. 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*, uma vez que toda a funcionalidade da *feature* já foi implementada em código. O CK dos *templates* existentes permite apenas associar expressões de *feature* à seleção de artefatos, ou, a transformações simples de pré-processamento dos artefatos e ativação de *tags* de pré-processamento. A Figura 10 mostra no lado esquerdo da imagem o *template* ADICIONAR NOVA FEATURE OPCIONAL [3] sendo aplicado ao exemplo do CK *Mobile Media* usando a notação antiga. Nele, listamos todos os artefatos associados à expressão de *feature Mobile Media* na primeira linha. No lado direito da imagem temos o CK com a notação utilizada pelo *Hephaestus*, onde usamos apenas a transformação *selectAllComponents*, que considera transformações

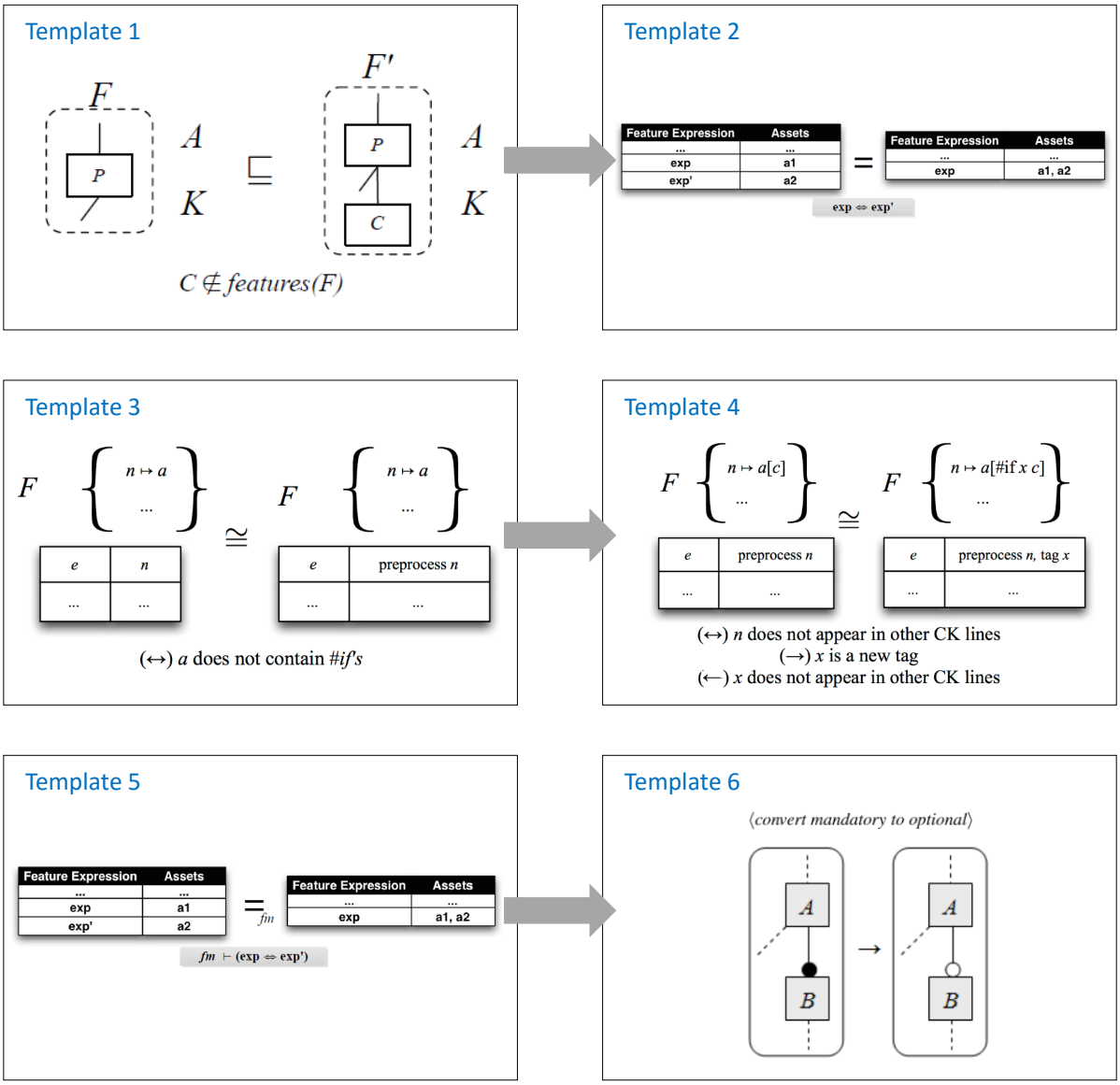

Fig. 9 – Sequência de *templates* usada para tentar extrair a *feature* *EventsSimulation*

adicionais, que permitem selecionar todos os artefatos existentes na LPS. 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*.

A partir desse momento, iniciamos a evolução segura da LPS seguindo o passo a passo do processo de extração de *features* detalhado na Figura 11. 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

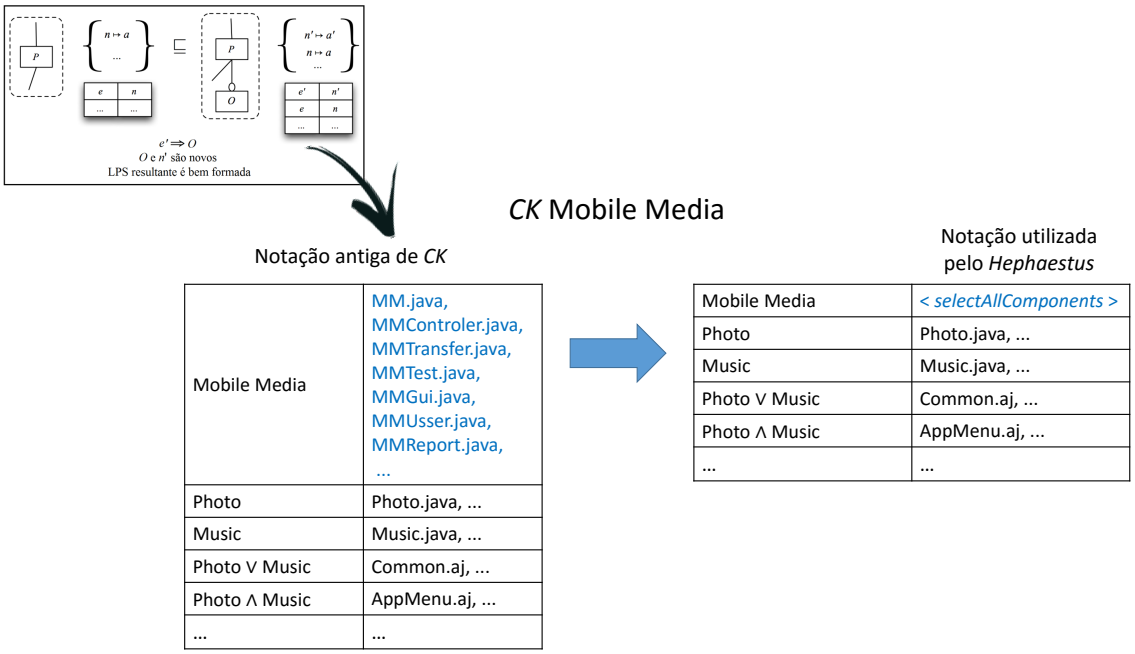

Fig. 10 – Comparação da notação antiga de CK com a notação utilizada pelo *Hephaestus* [1]

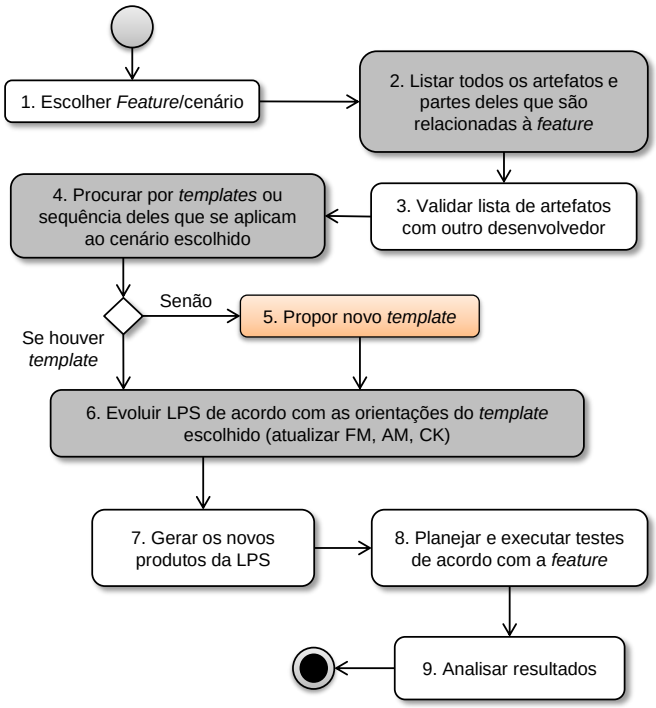

Fig. 11 – Passos do processo de extração de novas *features*

à *feature* escolhida. A seleção incorreta de qualquer arquivo relacionado à *feature* pode impactar no comportamento do produto final e na corretude da LPS como um todo. 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. Nesse passo é preciso ter atenção para verificar se todas as pré-condições e regras existentes no *template* são aplicáveis ao 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. Esse 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 esse passo para propor os *templates* discutidos a seguir. Na medida em que mais *templates* forem propostos, esse 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* [1]. 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. Para cada *feature* executamos um subconjunto diferente de testes já existentes. Para evidenciar que os produtos gerados que contém a *feature* apresentam o mesmo comportamento do produto original, executamos os testes 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 equivalentes. Somando todos os testes executados na versão inicial do sistema e em todas as etapas de extração da LPS, ao total foram executados 8000 testes unitários (automáticos) e 144 testes funcionais (manuais). Isso nos dá certa evidência 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.

Não analisamos o processo de extração de *features* em si (Figura 11), para avaliar se os passos definidos são os melhores a serem seguidos, pois não era o foco do nosso estudo. Por exemplo, poderíamos adotar uma outra estratégia de extração de *features* que o desenvolvedor primeiro realiza as mudanças na LPS e em seguida uma ferramenta tentaria encaixar os *templates* de acordo com as mudanças realizadas. Acreditamos que o resultado obtido durante o processo de desenvolvimento não teve impacto negativo devido ao processo de extração de *features* adotado, que serviu como um guia prático de acompanhamento e planejamento das várias etapas do desenvolvimento. Esse processo de desenvolvimento foi executado para cada nova extração de *feature* durante a evolução da LPS. A seguir apresentamos detalhadamente os novos *templates* propostos.

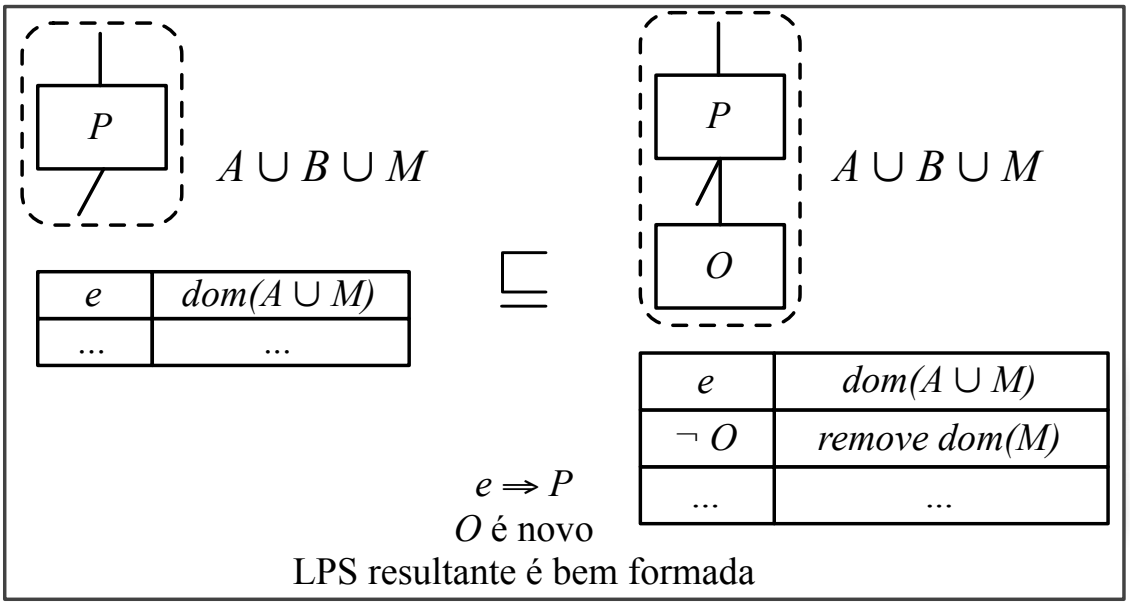

Fig. 12 – Template transformar artefato existente em nova feature

4.1 TEMPLATE TRANSFORMAR ARTEFATO EXISTENTE EM NOVA FEATURE

Esse *template* é útil quando um desenvolvedor precisa associar artefatos existentes a uma nova *feature*, sem alteração ou adição de artefatos, como ilustra a Figura 12, onde $A \cup B \cup M$ no AM representa os artefatos da LPS, que são formados pela união de três conjuntos. O conjunto M representa os artefatos de nosso interesse, isto é, aqueles que serão associados com a nova *feature*. Os conjuntos A e B representam os artefatos associados com outras *features* ou obrigatoriamente incluídos nos produtos da LPS, podendo inclusive ser vazio se não existir outra *feature*, por exemplo. A primeira linha do CK original associa e com $dom(A \cup M)$, onde e é uma expressão de *feature* qualquer que implica na seleção da *feature* P, e a notação $dom(\ldots)$ representa a seleção de um conjunto de nomes de artefatos. No exemplo da Figura 13, a expressão de *feature* e foi substituída por "*Root*" e do lado direito a notação $dom(\ldots)$ por "*selectAllComponents*" para representar a seleção de todos os artefatos da LPS. No caso desse *template* em específico, utilizamos $A \cup M$, indicando que devem ser incluídos todos os artefatos nos conjuntos A e M. As reticências indicam que esse CK pode inclusive ter outros mapeamentos associando artefatos do conjunto B ou não, se este for vazio 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.

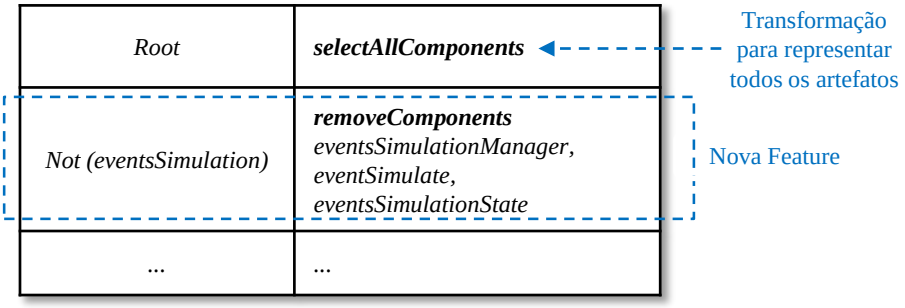

Fig. 13 – Exemplo de *Configuration Knowledge* mais expressivo

A Figura 13 mostra o exemplo real de parte do CK mais expressivo onde uma única transformação (*selectAllComponents*) pode representar todos os artefatos da LPS. A diferença entre a transformação (*selectAllComponents*) usada pelo *Hephaestus* e a abstração dela que aparece no *template*, é que a notação *dom(A ∪ M)* usada pelo *template* é mais geral, pois seleciona os nomes de artefatos do domínio A e M, que podem ser todos os artefatos da LPS ou não. Essa notação usada pelo *template* é independente de ferramenta, e caso a ferramenta não possua uma transformação semelhante a *selectAllComponents* usada pelo *Hephaestus*, a mesma poderá ser implementada usando apenas *select* e a lista dos artefatos relacionados. Nesse 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*.

O FM da linha resultante não especifica o tipo da nova *feature*, podendo ela ser obrigatória, opcional, 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. Representamos o fato de que uma *feature* pode ter outras *features* relacionados a ela usando uma linha acima ou abaixo dela.

A pré-condição do *template* afirma que a expressão e deve implicar a *feature* P, pois caso essa condição não seja satisfeita pode acontecer mudança de comportamento em alguns produtos da LPS resultante. Devido ao fato da nova *feature* O ser filha da *feature* P, é necessário restringir o conjunto de artefatos existentes *dom(A ∪ M)* para a *feature* P que também já existe. Poderíamos usar a expressão de *feature* do lado esquerdo do *template* apenas com P, mas usamos e para dar mais generalidade ao *template*. A segunda pré-condição 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, pois independente da linguagem envolvida na extração da *feature*, os produtos gerados sem os artefatos associados com a nova *feature* devem válidos. A LPS

resultante não seria bem formada se ao gerar um produto sem a nova *feature* ocorresse um erro de compilação relacionado à falta de um artefato, por exemplo. Podemos verificar a restrição de boa formação usando a abordagem de composição segura [11, 12].

Para nos assegurarmos que este *template* está consistente com a noção de evolução segura, precisamos demonstrar que o mesmo adere à noção de refinamento de LPS, ou seja, que todo produto da LPS inicial tem um correspondente na LPS resultante. Para este *template*, isto corresponde a demonstrar que para todo produto da LPS inicial, há ao menos um produto que consiste exatamente dos mesmos artefatos que o produto original, ou seja, é sintaticamente equivalente. Sendo assim, o refinamento é garantido por reflexividade [5]. Isto é demonstrado pelo fato de que todo produto original onde P é selecionada inclui pelo menos os artefatos referenciados por A e M. Na LPS resultante, produtos onde as *features* P e O forem selecionadas terão os mesmos artefatos, uma vez que a única mudança no CK é a introdução da segunda linha, que está associada com a ausência da *feature* O na configuração do produto. Já os produtos onde P é selecionada, mas sem a seleção da *feature* O não tem artefatos referenciados por M.

Embora não tenhamos encontrado nenhum cenário no sistema, puramente composicional, que pudéssemos aplicar esse *template*, acreditamos que o mesmo pode ser útil. Por exemplo, se o sistema escolhido tivesse algum tipo de funcionalidade que implementasse um *plugin* do *eclipse*[2] para análise sintática de aplicações para TV Digital. Dependendo de como tenha sido sua implementação, é provável que esse *template* pudesse ser usado para extrair essa *feature*. Implementações de *plugins* para o *eclipse* são exemplos de cenários composicionais, pois podem ser removidos ou adicionados sem impactar o sistema existente. Tanto esse *template* como os demais, foram formalizados e provados de acordo com a teoria de evolução segura de LPS [5]. A seguir, iremos detalhar os outros dois *templates*, onde iremos apresentar casos reais da utilização dos mesmos.

4.2 TEMPLATE 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. Os artefatos que contêm diretrizes de pré-processamento devem ser listados no CK com a transformação *preprocess*, caso contrário, poderemos ter erros de compilação se o tipo do artefato não aceitar as diretrizes. Por exemplo, para suportar um novo sistema de gerenciamento de banco de dados, um desenvolvedor poderia adicionar diretivas de compilação condicional nas classes que implementam as configurações do banco de dados, e dependendo da escolha do usuário ou

[2] <https://eclipse.org/>

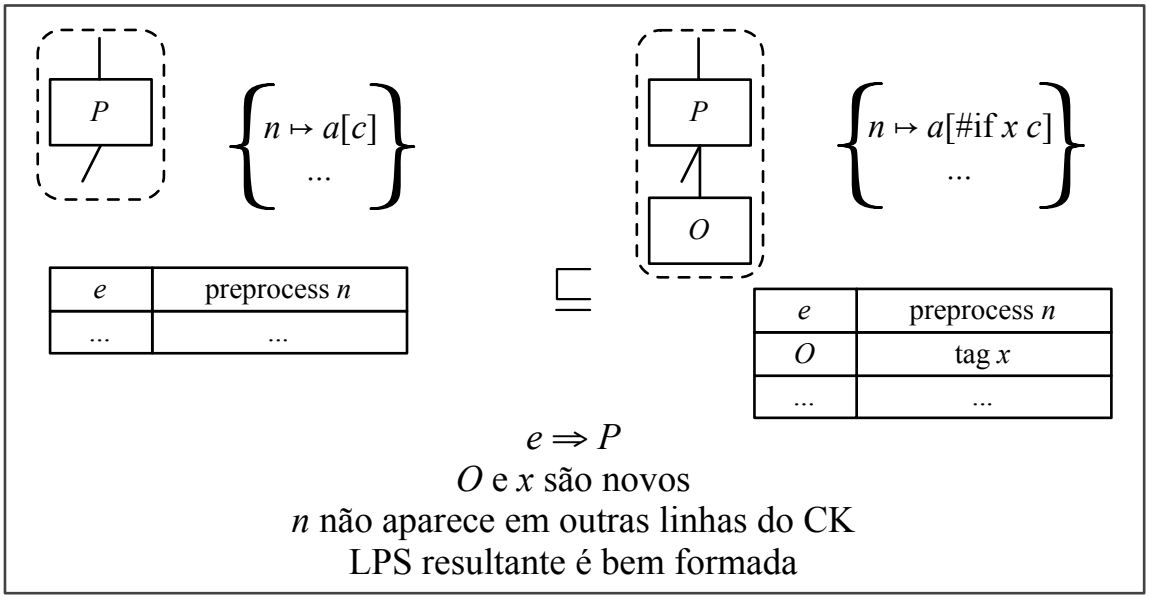

Fig. 14 – Template transformar artefato pré-processado em nova feature

do sistema operacional, teremos duas versões do sistema com diferentes gerenciadores de banco de dados, sem precisar remover ou ter apenas uma opção de sistema de gerenciamento de banco de dados. Desta forma, utilizamos uma notação ligeiramente diferente do template anterior. A Figura 14 mostra o *template* para extrair *feature* a partir do contexto anotativo.

Com este *template* básico é possível anotar um segmento de código c, presente em um artefato a, com uma diretiva de pré-processamento associada a uma *tag* x e preservar o comportamento da *feature*. Para garantir que o segmento de código c existente estará sempre presente no artefato a, precisamos associar tanto a transformação de pré-processamento como a declaração de *tag* x no CK à mesma expressão de *feature* e artefato a associado anteriormente.

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 *preprocess*, responsável por pré-processar artefatos da LPS, de acordo com as diretivas de compilação condicional. Quando e for verdadeiro o artefato associado a n é pré-processado e o resultado adicionado ao produto gerado. A meta-variável n representa o nome do artefato relacionado à nova *feature*, que do lado esquerdo do *template* no AM está associada a notação $a[c]$. Essa notação representa um trecho de código qualquer c presente em um artefato a. Após as mudanças na LPS, parte do seu código c foi 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, e se a *tag* de preprocessamento x for habilitada.

Uma nova linha no CK é adicionada associando a nova *feature* O com a transfor-

mação *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 nesse caso é que além de não poder ter outra *feature* chamada *O* no FM, também não podemos ter outra *tag* chamada *x* no CK. Essas restrições são necessárias porque sem elas poderíamos ter a *tag x* associada a outras expressões de *features*, o que modificaria o comportamento do artefato *a*. Caso isso acontecesse, não seria considerada uma evolução segura, pois o produto original não seria compatível com nenhum produto na LPS resultante, visto que o comportamento do artefato *a* foi alterado. Além disso, o nome do artefato *n* relacionado à nova *feature* não pode aparecer em outras linhas do CK, para assegurar a evolução segura da LPS. Essa condição é necessária, pois se o nome do artefato *n* aparecer em outras linhas do CK pode gerar uma inconsistência, duas expressões de *features* verdadeiras uma para remover e outra para adicionar o artefato no produto final, nesse caso a LPS resultante é considerada bem formada se todos os produtos gerados são bem formados. No *template* em questão, representamos a inclusão de uma diretiva em um arquivo apenas, mas variações desse *template* pode ser aplicado a diversos arquivos ao mesmo tempo.

Os artefatos que não possuem a anotação *#ifdef* normalmente também não estão associados à transformação de pré-processamento no CK necessária para aplicação do *template* proposto. Nesses casos, podemos usar um *template* existente para introduzir a transformação de pré-processamento aos itens do CK relacionados à nova *feature*. Podemos utilizar o *template* já existente *Preprocess Asset Without Preprocessor Directive* [3] apresentado na Figura 15. Esse *template* estabelece que podemos exigir o pré-processamento de um artefato, desde que esse artefato não contenha diretrizes de pré-processamento. De fato, o pré-processamento não tem efeito para esse tipo de artefato, pois o mesmo não contém diretivas *#if's* introduzidas no arquivo, conforme pré-condição do *template*. Assim sendo, a evolução é segura porque preservamos o comportamento de todos os produtos que incluem *a*. Pois *a* não contém diretivas *#if's*. Note que o *template* apresentado usa o símbolo $\cong$, em vez de $\sqsubseteq$, para indicar que o refinamento da LPS é válido em ambas as direções.

O *template Preprocess Asset Without Preprocessor Directive* [3] foi usado na prática como passo intermediário para extrair a *feature EvaluateUnusedNCLFiles*. A Figura 16 mostra a aplicação real do *template* com a adição da transformação *preprocess* para os artefatos pertencentes à nova *feature* a ser extraída. Nela, o FM *F* do *template* foi substituído pelo FM com a *feature Root*, não alterado durante a evolução. O AM do *template* foi substituído pelo conteúdo entre as chaves no exemplo, que também coincide com o *template* e não foi alterado durante a evolução. Como informamos, no CK adicionamos a transformação *preprocess* para os artefatos relacionados à nova *feature*, porém usamos uma variação do *template* que contempla vários artefatos representados por *n*. Após

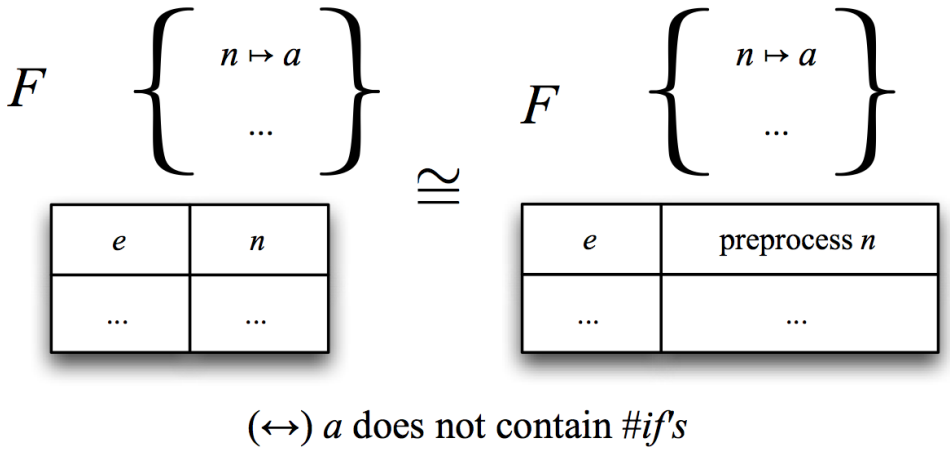

Fig. 15 – *Template Preprocess Asset Without Preprocessor Directive* [3]

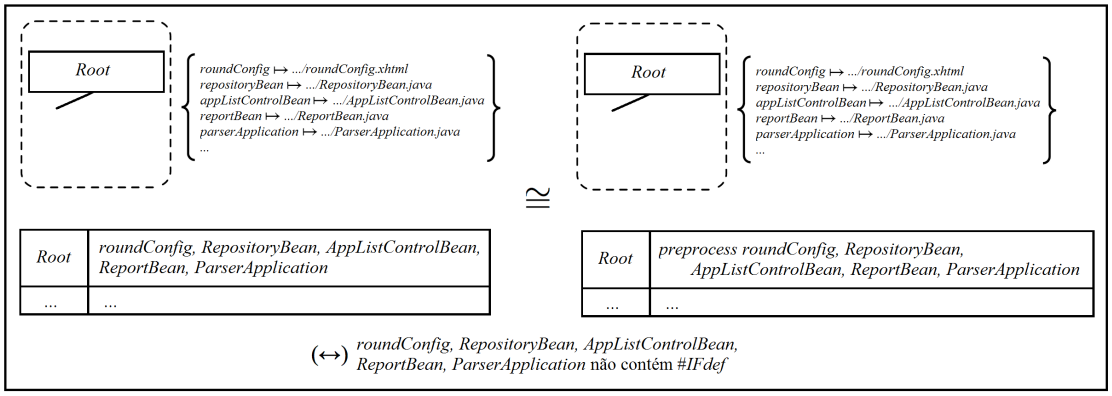

Fig. 16 – Passo intermediário antes de aplicar o *template* 2: uso do *template* *Preprocess Asset Without Preprocessor Directive* [3]

aplicar esse *template*, seguimos as orientações contidas no novo *template* (TRANSFORMAR ARTEFATO PRÉ-PROCESSADO EM NOVA FEATURE) para assim, adicionar a nova *feature* *EvaluateUnusedNCLFiles* conforme apresentado na Figura 17. Nela, verificamos que o FM foi alterado para adicionar a *feature* *EvaluateUnusedNCLFiles* substituindo a *feature* *O* usada pelo *template*. Também adicionamos as diretivas de compilação condicional em todos os artefatos relacionados à nova *feature* conforme ilustrado no AM em azul e por $[\#if\ x\ c]$ no *template* proposto. No CK, adicionamos uma nova referência com a associação entre a nova *feature* *EvaluateUnusedNCLFiles* e a *tag* *tagEvalUnusedNCLFiles*, representadas no *template* por *O* e *tag* *x* respectivamente.

Continuando o detalhamento do exemplo para adicionar a *feature* *EvaluateUnusedNCLFiles*, apresentamos na Figura 18 um trecho de código com a compilação condicional usada em um método de classe. Nesse exemplo, o *a* existente no *template* TRANSFORMAR ARTEFATO PRÉ-PROCESSADO EM NOVA FEATURE seria a classe como um todo, e o *c* seria a declaração do método *addUnusedApplicationNCLData(ApplicationNCLData applicationNCLData, File folder)*. Esse método pertence a uma das classes que controla o fluxo da avaliação sintática de aplicações e o método tem a responsabilidade de adicionar os arquivos

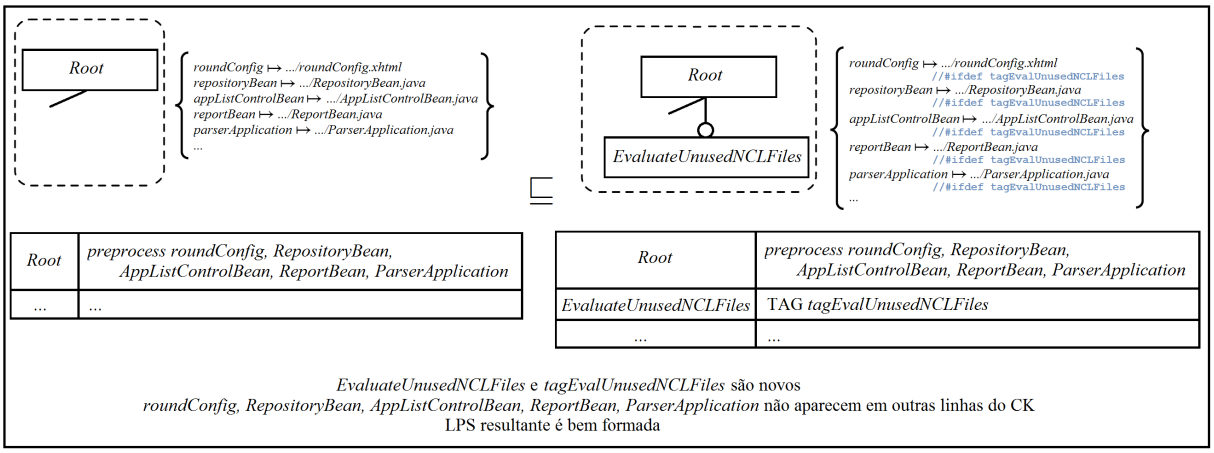

Fig. 17 – Exemplo de uso do *template* 2

NCL[3] não usados pela aplicação para serem avaliados sintaticamente. Introduzimos a anotação *#ifdef* seguida da *tag* referente à nova *feature EvaluateUnusedNCLFiles*, 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 "tagEvalUnusedNCLFiles"*, forem bem-formados. Ela não seria bem formada, se os trechos de código envolvidos com a *tag "tagEvalUnusedNCLFiles"* forem usados por outras *features*, nesse caso ocorreria erro de compilação nos produtos gerados sem a *feature EvaluateUnusedNCLFiles*.

Assumimos que os produtos que incluem a nova *feature* já eram bem formados anteriormente, por já ser uma LPS isso já é garantido por definição, dessa maneira 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, todos os produtos da LPS inicial tem algum produto correspondente sintaticamente equivalente na LPS resultante. De maneira similar ao raciocínio para o *template* anterior, na LPS resultante, a modificação da LPS original resulta na possibilidade de gerar novos produtos que não incluem o comportamento associado com a *feature O*. Desta forma, os produtos que incluem *O* serão exatamente iguais a produtos da LPS inicial, uma vez que o trecho de código *c*, agora associado com a *tag x*, será incluído no produto gerado.

[3] NCL (*Nested Context Language*) é uma linguagem declarativa para autoria de documentos hipermídia baseados no modelo conceitual NCM (*Nested Context Model*). <http://www.ncl.org.br/>

```
//#ifdef tagEvalUnusedNCLFiles
private void addUnusedApplicationNCLData(ApplicationNCLData applicationNCLData, File folder) throws IOException {
	for (ElemNCL unusedNCL : unusedElemNCLs.keySet()) {

		File unusedfile = unusedElemNCLs.get(unusedNCL);
		String fullName = unusedfile.getAbsolutePath();

		if(!nclUnusedFiles.containsKey(fullName)){

			ApplicationNCLData unusedApplicationNCLData =
			this.createApplicationNCLData(unusedfile, unusedNCL,unusedfile,unusedfile.getParentFile());

			applicationNCLData.getNestedElemNCL().add(unusedApplicationNCLData);
			nclUnusedFiles.put(fullName, unusedApplicationNCLData);

		}else if(!applicationNCLData.getNestedElemNCL().contains(nclUnusedFiles.get(fullName))){
			applicationNCLData.getNestedElemNCL().add(nclUnusedFiles.get(fullName));
		}
	}
}
//#endif
```

Fig. 18 – Trecho de código com anotação *#ifdef*

4.3 TEMPLATE TRANSFORMAR MÚLTIPLOS ARTEFATOS EXISTENTES EM NOVA FEATURE

O último *template* proposto foi criado a partir de uma combinação dos dois já detalhados nas seções anteriores. A Figura 19 apresenta o *template* que pode ser usado quando uma *feature* é implementada de forma composicional e anotativa. Durante o desenvolvimento de sistemas esse cenário acontece bastante, o desenvolvedor implementa classes que contém o detalhamento da funcionalidade e essa por sua vez é usada por diferentes partes do sistema. Ou seja, parte da implementação da *feature* está modularizada em classes, e parte entrelaçada e espalhada por meio dos artefatos, sendo necessário o uso de compilação condicional para demarcar o código.

Assim como nos *templates* anteriores, introduzimos uma nova *feature* O no FM. O AM novamente tem três partes, onde a parte A será associada inteiramente com a nova *feature*, como na Figura 12. Outra parte entre as chaves está associada com os trechos de código que serão envolvidos por diretivas, como na Figura 14. Finalmente, a terceira parte B 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 A 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, uma vez que o produto gerado será exatamente igual. Podemos observar que os artefatos referenciados por A estarão presentes, bem como o trecho de código c, associado à *tag* x, que só é ativada na presença da *feature* O. Portanto, todo produto da LPS original tem um produto sintaticamente igual, o que

garante refinamento por reflexividade [5].

Para usar esse *template*, assim como o *template* apresentado na seção anterior, é necessário introduzir a transformação de pré-processamento para os itens do CK relacionados à nova *feature* usando o *template Preprocess Asset Without Preprocessor Directive* [3].

O *template* TRANSFORMAR MÚLTIPLOS ARTEFATOS EXISTENTES EM NOVA FEATURE foi o mais usado durante o estudo, pois é bastante comum a *feature* ser implementada de forma composicional e anotativa. A Figura 20 mostra todas as alterações realizadas na LPS de acordo com o *template* proposto para adicionar a *feature EvaluateNCLFunctionalAreas* responsável por contabilizar as áreas funcionais NCL da aplicação para TV Digital. Na imagem, verificamos que o FM foi alterado para adicionar a nova *feature* opcional *EvaluateNCLFunctionalAreas* e dois artefatos do AM atualizados com as diretivas da compilação condicional. No CK, *dom(A ∪ M)* foi substituído pela transformação do *Hephaestus selectAllComponents* que contemplam no *template* os artefatos do conjunto *A* relacionados à nova *feature EvaluateNCLFunctionalAreas* e os artefatos do conjunto *B* pertencentes a outros artefatos do sistema. Os artefatos *DataBaseFacade* e *NCLFileData* que substituíram *n* no *template* e precisam de compilação condicional já estão estão relacionados à transformação *preprocess*. Na evolução da LPS foram adicionadas duas novas linhas no CK com a expressão associada à nova *feature* associadas à *tag tagEvalNCLFunctionalAreas* e a segunda linha com a transformação *removeComponents* e os artefatos da *feature* representados por *dom(A)* no *template* proposto. A LPS resultante é bem formada pois não houve erro de compilação nos produtos gerados sem a *feature EvaluateNCLFunctionalAreas*. Vale lembrar que, apenas parte da implementação da *feature EvaluateNCLFunctionalAreas* é exibida na imagem. Por fim, resumimos no Apêndice A todos os *templates* de evolução segura propostos neste trabalho.

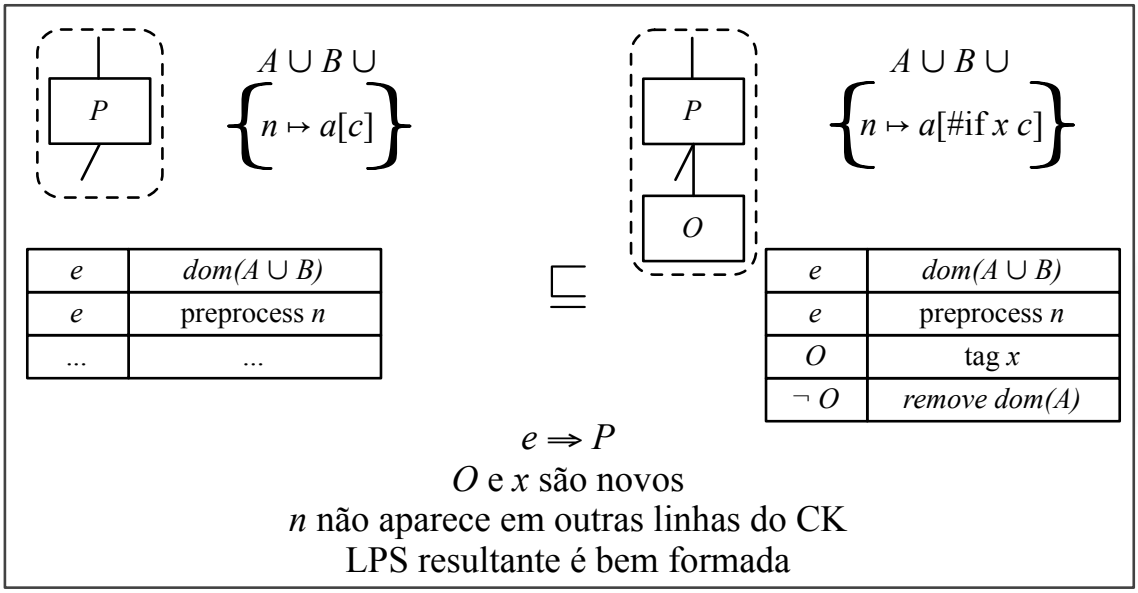

Fig. 19 – Template transformar múltiplos artefatos existentes em nova feature

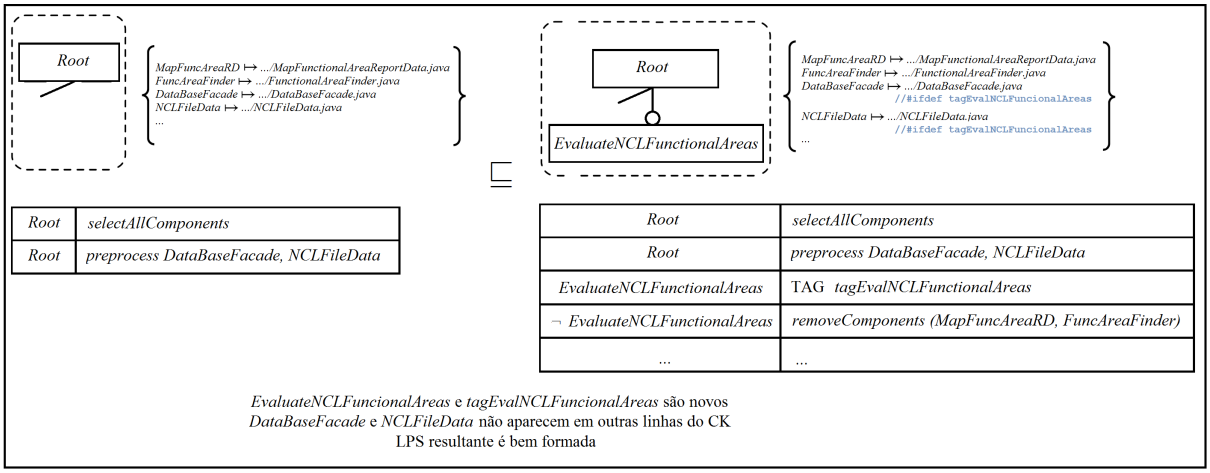

Fig. 20 – Exemplo de uso do *template* 3

5 AVALIAÇÃO

Os *templates* propostos foram avaliados do ponto de vista de corretude e expressividade. No aspecto de corretude, todos os *templates* foram formalizados e provados usando a ferramenta *Prototype Verification System* (PVS) [15],[1] pelo co-orientador desse trabalho, Leopoldo Teixeira, onde foi demonstrado que os mesmos estão de acordo com a teoria de evolução segura de LPS [5]. Além da verificação formal para verificar que o *template* está de acordo com a teoria, também avaliamos a utilidade e expressividade deles na prática. A fim de evidenciar que ao realizar uma evolução na LPS o desenvolvedor fez corretamente a aplicação dos *templates*, executamos testes que verificaram que o comportamento dos produtos foi preservado durante a evolução. Nesta seção, detalhamos o estudo de caso realizado, os resultados obtidos e, finalmente, suas ameaças à validade. O objetivo desse estudo consiste em responder a seguinte questão:

Os novos *templates* para extração de *features* podem ser usados para evolução segura de LPS?

A fim de responder essa pergunta, realizamos um estudo utilizando um sistema industrial desenvolvido em Java, o mesmo usado como exemplo em capítulos anteriores, com a finalidade de usar os *templates* propostos durante sua transformação em uma LPS. Com o propósito de evidenciar a evolução segura da LPS, executamos o mesmo subconjunto de testes no produto original (sem as mudanças da LPS) e no produto completo, gerado após todas as mudanças nos artefatos e na LPS. Após verificar os resultados dos testes em ambos os produtos (original e com todas as 15 *features* presentes), constatamos que os resultados foram os mesmos, portanto o comportamento do novo produto, com todas as 15 *features*, foi preservado. Além disso, ao extrair cada *feature* executamos um subconjunto diferente de testes relacionados à *feature* extraída, que nesse caso, serviu para evidenciar que os produtos gerados que contém a *feature* apresentam o mesmo comportamento do produto original, pois os resultados dos testes executados foram equivalentes. Para os novos produtos da LPS, gerados sem a nova *feature*, é esperado que o teste relacionado falhe e os testes realizados nos produtos com a nova *feature* passem, no sentido de que o resultado encontrado seja o mesmo encontrado no produto original. Adicionalmente, com relação a preservar comportamento, reforçamos que os *templates* foram formalizados e provados de acordo com a teoria de evolução segura de LPS [5]. Deste modo, colhemos evidência 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.

[1] <https://github.com/spgroup/theory-pl-refinement/>

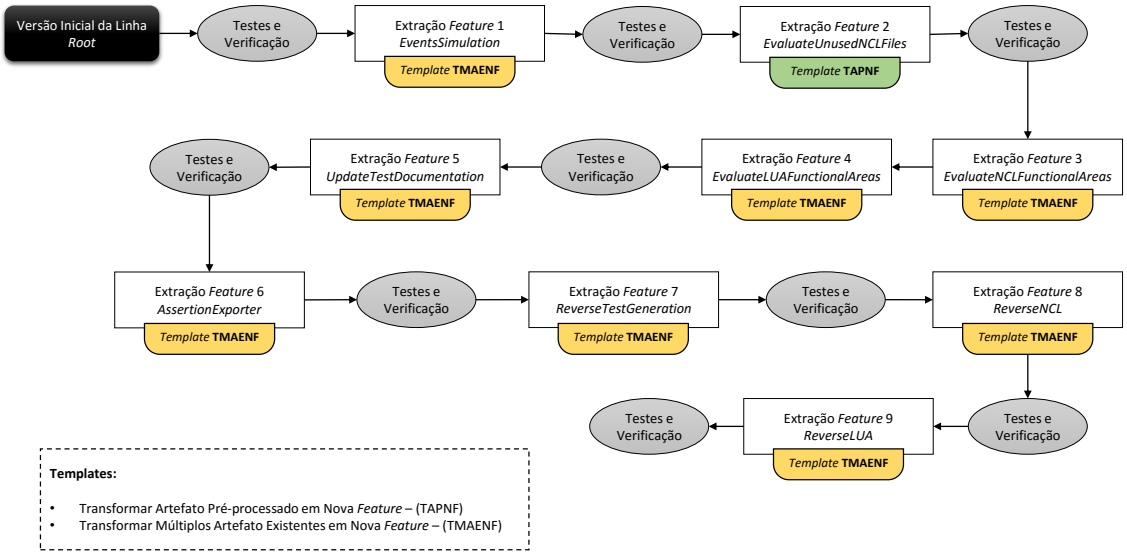

Fig. 21 – Evolução da nova LPS

5.1 ESTUDO DE CASO

Utilizamos um sistema familiar que possui duas funcionalidades principais com foco para o sistema de TV Digital brasileira, a geração de aplicações a partir de regras do modelo e a análise sintática e semântica de aplicações. Esse sistema pertence a 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 presentes e aproximadamente 400 KLOC. O propósito inicial era ter pelo menos dois produtos, um produto com o sistema completo e outro apenas sem a funcionalidade de análise sintática e semântica de aplicações. Como consequência negativa de ter apenas um produto, o sistema executa processos desnecessários para análise de aplicações que poderiam ser mais eficientes. 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. O autor desse trabalho esteve presente 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. Nesse caso, a LPS consistia de apenas um produto com todas as funcionalidades do sistema, conforme ilustrado na Figura 7 do Capítulo 3. Com a versão inicial da LPS iniciamos sua evolução conforme ilustrado na Figura 21. Nela mostramos 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 (elipse cinza) após as extrações de *features*. Apenas as *features* 3 e 4 que tiveram os testes e verificação realizados após a extração de ambas as *features*. As mesmas tinham vários artefatos em comum, portanto acreditamos que seria mais produtivo proceder com a extração de ambas, para em seguida

Tab. 1 – Descrição das *Features* extraídas

#	***Features***	**Descrição**
1	*EventsSimulation*	Simula eventos LUA[1] para avaliação sintática de código fonte de aplicações para TV Digital.
2	*EvaluateUnusedNCLFiles*	Avalia arquivos NCL[2] não usados pela aplicação para TV Digital.
3	*EvaluateNCLFunctionalAreas*	Mapeia e contabiliza as áreas funcionais NCL da aplicação para TV Digital.
4	*EvaluateLUAFunctionalAreas*	Mapeia e contabiliza as áreas funcionais LUA da aplicação para TV Digital.
5	*UpdateTestDocumentation*	Atualizar a documentação das aplicações geradas para TV Digital a partir das regras do modelo.
6	*AssertionExporter*	Exportar as assertivas (regras do modelo) para o formato XLSX.
7	*ReverseTestGeneration*	Ler e atualizar aplicações para TV Digital de acordo com as novas regras do modelo.
8	*ReverseNCL*	Ler e atualizar aplicações NCL para TV Digital de acordo com as novas regras do modelo.
9	*ReverseLUA*	Ler e atualizar aplicações LUA para TV Digital de acordo com as novas regras do modelo.

realizar os testes e verificações apenas uma vez, considerando os produtos da LPS com e sem as *features* extraídas. Todos os testes executados em cada produto, bem como a configuração de cada um deles, como estava a LPS no momento em que cada produto foi gerado, quais *features* estavam presentes e ausentes, serão detalhados na Seção 5.3.

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 experiência 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 descoberta e definição dos novos *templates* foi realizada durante a extração da primeira *feature*. Após constatar que nenhum *template* disponível poderia ser aplicado para evoluir com segurança a LPS, iniciamos a definição de um novo *template* para suportar o cenário encontrado. No estudo, apenas o *template* Transformar artefato existente em nova feature não foi usado, pois não encontramos nenhum cenário no sistema, puramente composicional, em que pudéssemos aplicar esse *template*. Apesar disso, acreditamos que esse *template* possa ser útil para ser aplicado em outros sistemas, com outros ambientes de desenvolvimento. Por outro lado, o mais usado foi o *template* Transformar múltiplos artefatos existentes em nova feature, que adiciona *features* implementadas com contexto composicional e anotativo.

A Figura 22 mostra o FM final da nova LPS, com todas as 9 *features* extraídas, onde apenas uma das *features* é obrigatória e todas as outras são opcionais. Também não foi extraída nenhuma *feature* alternativa e *Or*, pois o sistema não possui nenhum cenário com esses tipos de *features*. A implementação dessas 9 *features* envolve aproximadamente 6 KLOC e a intuição de cada uma delas é descrita na Tabela 1.

A Figura 23 mostra a configuração de cada produto gerado durante a evolução da LPS. O verde representa que a *feature* está presente no produto, enquanto que o vermelho

[1] LUA é uma linguagem de programação ideal para configuração, automação (*scripting*) e prototipagem rápida. <https://www.lua.org/>

[2] NCL (*Nested Context Language*) é uma linguagem declarativa para autoria de documentos hipermídia baseados no modelo conceitual NCM (*Nested Context Model*). <http://www.ncl.org.br/>

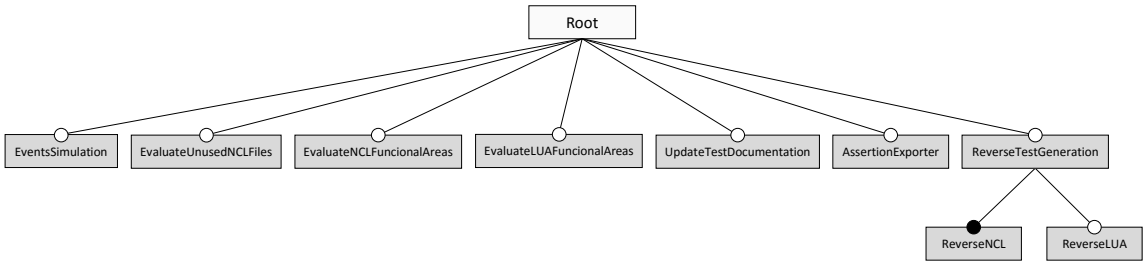

Fig. 22 – *Feature Model* da nova LPS

significa a ausência. Nessa figura, também é possível identificar o momento em que cada produto foi gerado e quais *features* já haviam sido extraídas. Por exemplo, no momento em que o produto #1 foi gerado, apenas a *feature EventsSimulation* havia sido extraída. No instante em que os produtos #7, #8, #9, #10 foram gerados a LPS possuía apenas 6 *features*. Quando a LPS estava com todas as 9 *features* extraídas geramos os produtos #13, #14 e #15 conforme ilustrado na figura. A fim de evidenciar a evolução segura, executamos o mesmo subconjunto de testes (automático e manual) no produto original (sem as mudanças) e nos produtos #9, #12 e #14, que independente da quantidade de *features* extraídas no momento, deveriam ter o mesmo comportamento da LPS original. Após verificar os resultados dos testes em ambos os produtos (original e com as mudanças), constatamos que os resultados foram os mesmos, portanto o comportamento dos produtos #9, #12 e #14 foram preservados durante a evolução.

Durante a evolução da LPS foram gerados 15 novos produtos, porém o FM permite gerar mais que 15 produtos. Decidimos gerar apenas 1 produto sem a *feature* extraída no momento e algumas outras configurações de produtos que escolhemos de forma aleatória conforme detalhamos a seguir. 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, e as *features* 1, 3 e 5 estiveram ausentes nesse produto. O produto #9 inclui as *features* 1, 2, 3, 4, 5 e 6 e o produto #10 tem todas as 6 respectivas *features* ausentes. Os produtos #11 e #13 foram gerados sem as *features* 7 (*ReverseTestGeneration*) e 9 (*ReverseLUA*), respectivamente e o produto #12 com a *feature* obrigatória 8 (*ReverseNCL*).

Os produtos #14 e #15 só foram gerados após realizar todas as alterações na LPS para extrair as 9 *features*. O produto #14 inclui todas as 9 *features* e o produto #15 tem todas as 9 *features* ausentes. Todas essas configurações de produtos estão representadas

Produto #1	Produto #2	Produto #3	Produto #4	Produto #5
Root	Root	Root	Root	Root
f1 - EventsSimulation	*f1 - EventsSimulation*	*f1 - EventsSimulation*	*f1 - EventsSimulation*	*f1 - EventsSimulation*
	f2 - EvaluateUnusedNCLFiles	*f2 - EvaluateUnusedNCLFiles*	*f2 - EvaluateUnusedNCLFiles*	*f2 - EvaluateUnusedNCLFiles*
		f3 - EvaluateNCLFunctionalAreas	*f3 - EvaluateNCLFunctionalAreas*	*f3 - EvaluateNCLFunctionalAreas*
		f4 - EvaluateLUAFunctionalAreas	*f4 - EvaluateLUAFunctionalAreas*	*f4 - EvaluateLUAFunctionalAreas*

Produto #6	Produto #7	Produto #8	Produto #9	Produto #10
Root	Root	Root	Root	Root
f1 - EventsSimulation	*f1 - EventsSimulation*	*f1 - EventsSimulation*	*f1 - EventsSimulation*	*f1 - EventsSimulation*
f2 - EvaluateUnusedNCLFiles	*f2 - EvaluateUnusedNCLFiles*	*f2 - EvaluateUnusedNCLFiles*	*f2 - EvaluateUnusedNCLFiles*	*f2 - EvaluateUnusedNCLFiles*
f3 - EvaluateNCLFunctionalAreas	*f3 - EvaluateNCLFunctionalAreas*	*f3 - EvaluateNCLFunctionalAreas*	*f3 - EvaluateNCLFunctionalAreas*	*f3 - EvaluateNCLFunctionalAreas*
f4 - EvaluateLUAFunctionalAreas	*f4 - EvaluateLUAFunctionalAreas*	*f4 - EvaluateLUAFunctionalAreas*	*f4 - EvaluateLUAFunctionalAreas*	*f4 - EvaluateLUAFunctionalAreas*
f5 - UpdateTestDocumentation	*f5 - UpdateTestDocumentation*	*f5 - UpdateTestDocumentation*	*f5 - UpdateTestDocumentation*	*f5 - UpdateTestDocumentation*
	f6 - AssertionExporter	*f6 - AssertionExporter*	*f6 - AssertionExporter*	*f6 - AssertionExporter*

Produto #11	Produto #12	Produto #13	Produto #14	Produto #15
Root	*Root*	*Root*	*Root*	*Root*
f1 - EventsSimulation	*f1 - EventsSimulation*	*f1 - EventsSimulation*	*f1 - EventsSimulation*	*f1 - EventsSimulation*
f2 - EvaluateUnusedNCLFiles	*f2 - EvaluateUnusedNCLFiles*	*f2 - EvaluateUnusedNCLFiles*	*f2 - EvaluateUnusedNCLFiles*	*f2 - EvaluateUnusedNCLFiles*
f3 - EvaluateNCLFunctionalAreas	*f3 - EvaluateNCLFunctionalAreas*	*f3 - EvaluateNCLFunctionalAreas*	*f3 - EvaluateNCLFunctionalAreas*	*f3 - EvaluateNCLFunctionalAreas*
f4 - EvaluateLUAFunctionalAreas	*f4 - EvaluateLUAFunctionalAreas*	*f4 - EvaluateLUAFunctionalAreas*	*f4 - EvaluateLUAFunctionalAreas*	*f4 - EvaluateLUAFunctionalAreas*
f5 - UpdateTestDocumentation	*f5 - UpdateTestDocumentation*	*f5 - UpdateTestDocumentation*	*f5 - UpdateTestDocumentation*	*f5 - UpdateTestDocumentation*
f6 - AssertionExporter	*f6 - AssertionExporter*	*f6 - AssertionExporter*	*f6 - AssertionExporter*	*f6 - AssertionExporter*
f7 - ReverseTestGeneration	*f7 - ReverseTestGeneration*	*f7 - ReverseTestGeneration*	*f7 - ReverseTestGeneration*	*f7 - ReverseTestGeneration*
	f8 - ReverseNCL	*f8 - ReverseNCL*	*f8 - ReverseNCL*	*f8 - ReverseNCL*
		f9 - ReverseLUA	*f9 - ReverseLUA*	*f9 - ReverseLUA*

Fig. 23 – Configuração de cada produto gerado durante a evolução da LPS

na Tabela 2, coluna *Configuração dos Produtos*. Na próxima seção, iremos detalhar como foi a criação da LPS até a geração dos produtos usando a ferramenta *Hephaestus* [1].

5.2 CRIANDO LINHAS DE PRODUTOS COM HEPHAESTUS

Inicialmente, para gerar o primeiro produto do sistema escolhido, usando o *Hephaestus*, tivemos que criar, a partir do zero, os três elementos da LPS, conforme descrito a seguir:

- *Feature Model* (FM): Como todo o sistema já estava implementado, consideramos apenas a *feature* inicial *Root*, com todas as funcionalidades principais já implementadas inicialmente. Nesse caso, como temos apenas uma *feature* no FM, só é possível gerar apenas um produto, sem variação e com todas as funcionalidades presentes. Depois disso, fomos adicionando as novas *features* ao FM, de modo que a versão final é apresentada no Arquivo 5.1.

Arquivo 5.1 – FM da nova LPS usado pelo *Hephaestus*

```
<featureModel chosenLayoutAlgorithm="1">
  <struct>
        <and mandatory="true" name="root">
           <feature name="eventsSimulation"/>
           <feature name="evaluateUnusedNCLFiles"/>
           <feature name="evaluateNCLFunctionalAreas"/>
           <feature name="evaluateLUAFunctionalAreas"/>
           <feature name="updateTestDocumentation"/>
           <feature name="assertionExporter"/>
           <and name="reverseTestGeneration">
              <feature mandatory="true" name="reverseNCL"/>
              <feature name="reverseLUA"/>
           </and>
        </and>
  </struct>
  <calculations Auto="true" Constraints="true" Features="true" Redundant="true"/>
  <comments/>
  <featureOrder userDefined="false"/>
</featureModel>
```

- *Configuration Knowledge* (CK): Para relacionar a *feature* inicial *Root* com todos os artefatos existentes do sistema adicionamos no CK a expressão de *feature* associando a *feature Root* à transformação *selectAllComponents*, que seleciona todos os artefatos do projeto para a geração do produto final. O Arquivo 5.2 mostra a transformação citada além de parte do CK usado para gerar os produtos da LPS.

- *Asset Mapping* (AM): Como usamos a transformação *selectAllComponents* no CK, não foi preciso relacionar todos os artefatos do sistema no AM, portanto esse elemento da LPS ficou vazio inicialmente. No entanto, ao extrair as novas *features*, foi necessário listar no AM os nomes de artefatos e os caminhos completos para os artefatos usados por cada *feature*, conforme apresentado no Arquivo 5.3.

Todos os arquivos completos utilizados pelo *Hephaestus* para geração dos novos produtos estão disponíveis em outro lugar.[2] Após criar cada arquivo da LPS e gerar o primeiro produto com todas as funcionalidades, iniciamos a extração de *features* conforme apresentado na seção anterior. Para extrair as *features* partimos do princípio que todos os artefatos do sistema estavam presentes no produto, logo precisaríamos remover os artefatos relacionados à *feature* quando a mesma não estiver presente na configuração do produto a ser gerado. Para selecionar todos os arquivos do sistema usamos a transformação *selectAllComponents* no CK e para remover os artefatos relacionados à nova *feature* quando ela não for selecionada, usamos a expressão de *feature* com operador de negação (¬) e a transformação *remove*. Quando a mesma estiver presente, a expressão de *feature* será falsa e os artefatos relacionados à mesma permanecerão presentes no produto.

[2] <https://sites.google.com/cin.ufpe.br/fcb-msc/>

Arquivo 5.2 – Parte do CK da nova LPS usado pelo *Hephaestus*

```
<configurationModel>
  <configuration>
        <expression>root</expression>
        <transformation>
               <name>selectAllComponents</name>
           <args>-</args>
        </transformation>
  </configuration>

  <configuration>
        <expression>Not(eventsSimulation)</expression>
        <transformation>
               <name>removeComponents</name>
           <args>eventsSimulationManager, eventSimulate, eventsSimulationState,
        simulationDataFolder</args>
        </transformation>
  </configuration>

  <configuration>
        <expression>eventsSimulation</expression>
        <transformation>
               <name>createBuildEntries</name>
               <args>tagEvtSimulation</args>
        </transformation>
  </configuration>
```

Arquivo 5.3 – Parte do AM da nova LPS usado pelo *Hephaestus*

```
eventsSimulationManager => luabehavioral/simulation/EventsSimulationManager.java;
eventSimulate => luabehavioral/simulation/event/EventSimulate.java;
eventsSimulationState => luabehavioral/simulation/data/EventsSimulationState.java;
simulationDataFolder => luabehavioral/simulation/data;
luaBehavioralSimulatorBean => luabehavioral/simulation/LuaBehavioralSimulatorBean.java;
luaBehavioralSimulator => luabehavioral/simulation/LuaBehavioralSimulator.java;
register => luabehavioral/simulation/event/register/Register.java;
globals => luabehavioral/simulation/environment/Globals.java;
luaBehavioralDocumentation => luabehavioral/reporter/LuaBehavioralDocumentation.java;
luaMedia => luabehavioral/nclextractor/data/LuaMedia.java;
evaluationRound => structural/data/model/EvaluationRound.java;
evaluationProjectLoader => structural/data/EvaluationProjectLoader.java;
executionMonitoringListLoader => structural/data/ExecutionMonitoringListLoader.java;
```

Para gerar os novos produtos a partir de contexto anotativo usamos a anotação *#ifdef* e uma versão atualizada do *Hephaestus*.[3] Desta forma, todos os produtos foram gerados com segurança e todas as alterações realizadas nos elementos da LPS seguiram as orientações apresentadas nos novos *templates*. Todos os produtos gerados, bem como a quantidade de testes realizados em cada um deles e os resultados obtidos nesse estudo são apresentados na próxima seção.

5.3 RESULTADOS

Para validar o comportamento dos produtos gerados, executamos testes automáticos e manuais. Os testes automáticos já existiam e foram usados durante a implementação das funcionalidades. Selecionamos um conjunto de 500 testes automáticos a partir de um total de aproximadamente 9,5K testes existentes. Esses testes automáticos são relacionados apenas a regras sintática e semântica do modelo de aplicações para TV Digital e como nenhuma *feature* extraída é relacionada diretamente à essas regras, optamos por não executar todo o conjunto de 9,5k testes existentes. A escolha dos testes foi feita buscando diversificar ao máximo as regras do modelo que o teste estava relacionado. Desse modo, acreditamos que o conjunto de testes automáticos selecionado é aceitável e confiável para verificar a corretude das funcionalidades básicas do sistema. Para todos os 15 produtos da LPS resultante executamos o mesmo conjunto com 500 testes automáticos. Os testes automáticos são considerados testes unitários ou pequenas aplicações de TV Digital que são executados e verificados pelo próprio sistema, sem a necessidade de uma ferramenta externa para execução dos testes.

A Figura 24 mostra um exemplo de aplicação NCL[4] para TV Digital. Essa aplicação NCL é parte de um teste unitário que usamos no trabalho. Além da aplicação NCL, o teste unitário é formado também pelo resultado esperado da análise sintática para essa aplicação, conforme mostrado na Figura 25. Nesse caso, o teste unitário verifica a corretude da regra do modelo para o atributo mandatório *id* do elemento *<region>*. Após cada aplicação NCL ser analisada pelo sistema, o resultado encontrado é comparado automaticamente com o esperado, e no final da execução é gerado um relatório com todos os testes executados e os resultados individuais conforme apresentado na Figura 26. A figura mostra uma parte do relatório com os testes unitários (automáticos) executados durante a validação dos produtos. Cada linha na tabela representa um teste unitário. O teste destacado em azul, é o mesmo teste apresentado nas Figuras 24 e 25. Pintados em verde são os testes que passaram durante a execução e em vermelho os testes que falharam. Como podemos observar alguns testes falharam, porém quando comparamos os resultados em ambos os

[3] <https://github.com/hephaestus-pl/hephaestus-base>

[4] NCL (*Nested Context Language*) é uma linguagem declarativa para autoria de documentos hipermídia baseados no modelo conceitual NCM (*Nested Context Model*). <http://www.ncl.org.br/>

```
<!--*****************************************
# TESTUNIT:BEGIN
 *****************************************-->

 <ncl id="KKdfGcLJv6OA3" >
       <head>
             <regionBase>
                   <region id="Iaf2avasg" >
                   </region>
             </regionBase>
       </head>
 </ncl>

<!--*****************************************
# TESTUNIT:END
*****************************************-->
```

Fig. 24 – Exemplo de teste unitário

```
CONFIG_APPLICATION_NAME=TestUnit_GingaAppEvaluator_region_id_mandatory_00_00001
ACTORS=Element x Attribute
STRUCTURAL_ANALYSIS_TYPE=SYNTACTIC
RELATIONSHIP_TYPE=Mandatory
RELATIONSHIP_TYPE_DESCRIPTION=Element with mandatory Attribute (child).
RELATIONSHIP_TYPE_POSSIBLE_SOLUTION=The element must have the attribute.
ELEMENT=Region
FIELD_NAME=id
RULE_CLASSIFICATION=ERROR
ASSERTIONS=layout_TA-005
ANNOTATION=@Mandatory
INDEX_ANNOTATION=N/A
WRAPPERS=N/A
TEST_UNIT_RESULT=true
```

Fig. 25 – Resultado esperado do teste unitário para ser verificado automaticamente

Test Case Name	Elemen	Field Nar	Relationship Typ	jUnit	Execution	Exception Comments
UT_GE_area_id_0DEADCODESET_00001	Area	id	Dead Code	FALSE	PASSED	
UT_GE_transition_id_0DEADCODESET_00007	Transition	id	Dead Code	TRUE	PASSED	
TestUnit_GingaAppEvaluator_area_beginposition_dependentrelationship_0	Area	beginPosition	Dependent Attribute	TRUE	PASSED	
TestUnit_GingaAppEvaluator_area_begintext_dependentrelationship_00_0	Area	beginText	Dependent Attribute	FALSE	PASSED	
TestUnit_GingaAppEvaluator_media_area_dependentrelationshipset-deper	Media	type	Dependent Attribute	FALSE	PASSED	
TestUnit_GingaAppEvaluator_media_src_dependentrelationshipset-depenc	Media	src	Dependent Attribute	TRUE	PASSED	
TestUnit_GingaAppEvaluator_media_src_dependentrelationshipset-depenc	Media	src	Dependent Attribute	TRUE	FAILED	Test Unit Result is TRUE but exist Exceptions: The resource defined by the attribute 'documentURI' of <importNCL> element does not exist.
TestUnit_GingaAppEvaluator_media_src_dependentrelationshipset-depenc	Media	src	Dependent Attribute	TRUE	PASSED	
TestUnit_GingaAppEvaluator_media_src_dependentrelationshipset-depenc	Media	src	Dependent Attribute	TRUE	PASSED	
TestUnit_GingaAppEvaluator_media_src_dependentrelationshipset-depenc	Media	src	Dependent Attribute	TRUE	PASSED	
TestUnit_GingaAppEvaluator_media_src_dependentrelationshipset-depenc	Media	src	Dependent Attribute	TRUE	FAILED	Test Unit Result is TRUE but exist Exceptions: The resource defined by the attribute 'documentURI' of <importNCL> element does not exist.
TestUnit_GingaAppEvaluator_property_name_mandatory_00_00002	Property	name	Mandatory	FALSE	PASSED	
TestUnit_GingaAppEvaluator_region_id_mandatory_00_00001	Region	id	Mandatory	TRUE	PASSED	
TestUnit_GingaAppEvaluator_region_id_mandatory_00_00002	Region	id	Mandatory	FALSE	PASSED	
TestUnit_GingaAppEvaluator_rule_comparator_mandatory_00_00001	Rule	comparator	Mandatory	TRUE	PASSED	

Fig. 26 – Relatório de execução de teste unitário (automático)

produtos (original e com todas as 15 *features* presentes), constatamos que os resultados foram os mesmos, portanto o comportamento do novo produto, com todas as 15 *features*, foi preservado. Como os testes unitários não verificam todas as partes do sistema, foi preciso executar outros tipos de testes (funcionais) a fim de coletar maior evidência de que o comportamento dos produtos existentes é preservado após as alterações na LPS.

A fim de verificar outras partes do sistema e focar mais a validação na *feature* extraída, executamos manualmente testes funcionais para cada produto. Esses testes já existiam e verificam as funcionalidades básicas do sistema, se estão todas funcionando corretamente e sem nenhum travamento. Além desses, planejamos e executamos alguns testes específicos para as *features* extraídas. Nesse caso, foram criados novos testes com objetivo de verificar o comportamento da *feature* extraída. Como os testes dependem de qual *feature* está sendo verificada, os testes executados e as quantidades de testes para cada produto são diferentes. Pelo menos um dos testes funcionais verifica especificamente a existência ou não da *feature* extraída de acordo com o produto. Portanto, pode ser que o comportamento esperado do teste seja verificar manualmente que uma determinada *feature* não esteja disponível no produto.

A Tabela 2 ilustra a quantidade de testes executados, bem como a configuração de cada produto (coluna 3, mesmas configurações apresentadas na Figura 23) gerado durante a evolução segura da LPS. Os testes (automáticos e manuais) executados nos produtos #0, #9, #12 e #14 são os mesmos. Com exceção desses citados agora, alguns produtos possuem a mesma quantidade de testes manuais. Embora isto possa sugerir que são os mesmos testes, isso é apenas uma coincidência, pois os testes e os resultados esperados para cada um deles são diferentes. Os testes foram executados em diferentes momentos da evolução da LPS (ver coluna 3), com exceção dos testes realizados nos produtos #13, #14 e #15, que foram realizados quando todas as 9 *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.

Para o produto #1, gerado no inicio da evolução da LPS e sem a *feature* opcional *EventsSimulation*, executamos o subconjunto de 500 testes automáticos mais 6 testes manuais. Dentre os testes manuais, um deles verifica que ao executar o processo para avaliação sintática de código fonte de aplicações para TV Digital a parte que simula eventos não foi realizada. Na Figura 27 apresentamos dois relatórios de avaliação sintática, o primeiro executado com o produto na versão inicial (produto #0) mostra 9 *warnings* encontrados na aplicação e o segundo sem *warnings*, pois a *feature EventsSimulation* não está presente. Desse modo, com a comparação de resultados verificamos que os novos produtos da LPS resultante são válidos e possuem comportamento diferente do produto original da LPS, conforme esperado para o produto sem a *feature EventsSimulation*.

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. Para verificar se a LPS resultante é bem formada, compilamos todos os novos produtos após cada geração e 3 deles (produtos #2, #3 e #5 na Tabela 2) apresentaram falhas de compilação. Mesmo com a validação e revisão feita por outro desenvolvedor, 2 artefatos foram associados incorretamente, um

Tab. 2 – Quantidade de testes executados por produto

Produto	*Descrição*	Configuração dos Produtos	*Testes Unitários (Automáticos)*	*Testes Funcionais (Manuais)*
#0	Versão inicial da linha	Sistema original, sem alterações	500	14
#1	Sem a *feature* 1 (*EventsSimulation*)	f1( )	500	6
#2	Sem a *feature* 2 (*EvaluateUnusedNCLFiles*)	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 (*EvaluateNCLFunctionalAreas*)	f1(**x**) f2(**x**) f3( ) f4(**x**)	500	6
#5	Sem a *feature* 4 (*EvaluateLUAFunctionalAreas*)	f1(**x**) f2(**x**) f3(**x**) f4( )	500	6
#6	Sem a *feature* 5 (*UpdateTestDocumentation*)	f1(**x**) f2(**x**) f3(**x**) f4(**x**) f5( )	500	6
#7	Sem a *feature* 6 (*AssertionExporter*)	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 as *features* 1,2,3,4,5 e 6	f1(**x**) f2(**x**) f3(**x**) f4(**x**) f5(**x**) f6(**x**)	500	14
#10	Sem as *features* 1,2,3,4,5 e 6	f1( ) f2( ) f3( ) f4( ) f5( ) f6( )	500	12
#11	Sem a *feature* 7 (*ReverseTestGeneration*)	f1(**x**) f2(**x**) f3(**x**) f4(**x**) f5(**x**) f6(**x**) f7( )	500	7
#12	Com a *feature* 8 (*ReverseNCL*)	f1(**x**) f2(**x**) f3(**x**) f4(**x**) f5(**x**) f6(**x**) f7(**x**) f8(**x**)	500	14
#13	Sem a *feature* 9 (*ReverseLUA*)	f1(**x**) f2(**x**) f3(**x**) f4(**x**) f5(**x**) f6(**x**) f7(**x**) f8(**x**) f9( )	500	7
#14	Com todas as 9 *features*	f1(**x**) f2(**x**) f3(**x**) f4(**x**) f5(**x**) f6(**x**) f7(**x**) f8(**x**) f9(**x**)	500	14
#15	Sem todas as 9 *features*	f1( ) f2( ) f3( ) f4( ) f5( ) f6( ) f7( ) f8( ) f9( )	500	14
			8000	144

Relatório 1: Versão Inicial da LPS ***(Produto #0)***

```
Lua File: /medias/SPECIALIST_TEST_UNNECESSARY_FLUSHS.lua
    Execution Flow Warnings: 9
        Summary:
            UNNECESSARY_FLUSHS: 9
        1.
        Rule description: The function: canvas:flush () must be called after a draw and composition operations sequence.
        Violation description: The function: canvas:flush () was called unnecessarily.
        Possible Solution: Remove unnecessary function: canvas:flush ()
        Line 14: flush
        Structure violated: FUNCTION
        Evaluator type: UNNECESSARY_FLUSHS
        Stack trace:
            Line 14: flush
```

Relatório 2: Versão sem a *feature EventsSimulation* ***(Produto #1)***

```
Lua File: /medias/SPECIALIST_TEST_UNNECESSARY_FLUSHS.lua
    TOTAL WARNINGS: 0
```

Fig. 27 – Relatórios de avaliação sintática de aplicação para TV Digital

à *feature* 2 *EvaluateUnusedNCLFiles* e outro à *feature* 4 *EvaluateLUAFunctionalAreas*. Os produtos gerados sem essas *features* apresentam problema de compilação, pois outras partes do sistema necessitavam desses artefatos que foram removidos do produto devido a ausência da *feature*.

Após identificadas, as falhas foram corrigidas e os novos produtos compilaram corretamente. Essas falhas foram encontradas antes da etapa de execução de testes automáticos e manuais, ocorreram devido a uma falha humana e não por deficiência do *template* proposto. Essa ocorrência reforça a necessidade de atenção às pré-condições e modificações dos elementos da LPS listadas pelos *templates*, pois produtos mal formados não condizem com a definição de LPS, que afirma que todos os produtos devem ser bem formados.

Com exceção da verificação da condição "LPS resultante é bem formada", nenhum outro defeito durante a evolução da LPS foi encontrado. Os testes executados nos produtos que não possuem a *feature* extraída certamente não tem comportamento compatível com o produto da LPS original. Após executar os testes e verificar seus resultados em ambos os produtos (original e com todas as 15 *features* presentes - produto #14 na Tabela 2), constatamos que os resultados foram os mesmos, portanto o comportamento do novo

produto, com todas as 15 *features*, foi preservado. Logo, podemos concluir que os nossos *templates* podem ser usados para extração de *features* durante a evolução segura de uma LPS.

5.4 AMEAÇAS À VALIDADE

Com relação à validade **interna**, identificamos duas principais: 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, o autor deste trabalho em conjunto com o orientador e co-orientador revisaram a validade da aplicação dos *templates*.

Além dessas, outra ameaça à validade interna é o fato de não ter utilizado os nossos *templates* para extrair *features* dos tipos alternativa e *Or*. Durante a análise de quais *features* utilizar no estudo, não encontramos nenhuma *feature* com essas características. Entretanto, nossos *templates* foram formalizados e provados de acordo com a teoria de evolução segura de LPS [5] sem definir o tipo específico de *feature*, o que nos dá confiança que o mesmo pode ser usado com segurança em outros cenários.

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 autor desse trabalho, que tinha 3 anos de experiência 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 pelo autor do trabalho que possui conhecimento tanto do sistema quanto de 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 manual 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 generalizados com confiança para outros projetos. Temos convicção que os *templates* propostos podem ser úteis para extração de *features* em outros sistemas, mas é possível que eles não sejam suficientes para cobrir todas as situações. Os projetos podem ter práticas de desenvolvimento e *features* com características diferentes, e como consequência, outros *templates* podem ser necessários. 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 dos testes foi feita buscando diversificar ao máximo a área do sistema com que o teste estava relacionado, porém outros testes poderiam revelar mudanças de comportamento não identificadas. Na verdade, os testes aproximam a noção de preservação de comportamento do software, por esse motivo o risco de construto. 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 útil para lidar com cenários de extração de *features* e reduz o risco de erros durante a evolução segura de uma LPS.

6 CONSIDERAÇÕES FINAIS

Neste capítulo serão apresentadas as considerações finais. Entre elas são discutidas as conclusões obtidas com a pesquisa, os trabalhos relacionados ao estudo e finalmente, as recomendações para trabalhos futuros.

6.1 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. Além de estarem de acordo com a noção de refinamento da LPS em que nos baseamos [5], os *templates* descritos abstraem e generalizam os cenários analisados. Dessa forma, eles complementam a lista de *templates* existentes e podem orientar os desenvolvedores durante a evolução segura de uma LPS. Observamos que nossos *templates* podem abordar as modificações seguras que os desenvolvedores realizaram nos cenários de extração de *features* analisados. Não avaliamos a extração de *features* sem o uso dos *templates*, porém estudos anteriores mostram que os *templates* ajudam a evitar erros de evolução que deveriam ser seguras, mas realmente alteraram o comportamento dos produtos existentes [3]. Acreditamos que alguns desses problemas poderiam ser evitados com um processo de teste de regressão mais rigoroso. No entanto, muitas vezes é caro gerar e testar todos os produtos da LPS.

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 LPS. Usando essa transformação para selecionar todos os artefatos, o esforço do desenvolvedor é menor, pois não vai precisar listar todos os artefatos da LPS no AM, como também evita que erros aconteçam durante a atualização manual do AM e do CK.

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 9 *features* e, a partir delas, foram gerados 15 novos produtos. Para evidenciar a evolução segura, executamos os testes antes e após as mudanças na LPS e verificamos que os resultados foram equivalentes. Nos produtos em que a *feature* extraída está presente o resultado do teste é o mesmo do teste executado na LPS original. Com base nesses resultados entendemos 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.

6.2 TRABALHOS RELACIONADOS

Alves et al.[16] propõem uma noção de refatoração informal para LPS, e apresentam um catálogo de refatoração para FM. Gheyi, Massoni e Borba[17] 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*, suporta 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 [14], ilustrando diferentes tipos de transformações que podem ser úteis para derivar e evoluir LPS. Borba, Teixeira e Gheyi[5] 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 et al.[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, usamos apenas uma transformação (*selectAllComponents*) para selecionar todos os artefatos das LPS, sem a necessidade de listá-los no CK.

Passos et al.[18] 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, como a remoção de *features*. Nossa intenção é fornecer uma guia para adicionar *features* e evoluir com segurança as LPS, enquanto eles estão interessados em relatar os cenários de evolução que acontecem em um período de tempo específico. Outra diferença é que a notação do CK dos nossos *templates* não exploram a cardinalidade e informações de atributos como *Kconfig* e *Kbuild*.

Thaker et al.[12] 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 FM e soluções SAT. Teixeira, Borba e Gheyi[11] apresentam uma abordagem para verificar a composição segura de LPS. 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 [19, 20, 21, 22] 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 et al.[20] 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, Apel e Batory[19] focam apenas em transformar artefatos de código, implicitamente utilizando noções de refinamento de programas orientado à aspectos [23]. Como discutido aqui e em outros lugares [14] esses não são suficientes para justificar o refinamento de LPS, pois temos de considerar não apenas código como FM e CK, por exemplo. Trujillo, Batory e Diaz[22] 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.

6.3 TRABALHOS FUTUROS

Embora tenhamos avaliado a utilidade do nosso conjunto de *templates*, sabemos que nossos resultados são limitados pelo contexto de apenas uma LPS que analisamos e que novos *templates* para extração de *features* podem ser necessários para justificar outras transformações que não consideramos neste trabalho. Além disso, é nossa intenção reforçar a utilidade dos *templates* sugeridos e complementar esses resultados, analisando outras LPS de diferentes domínios, com características e ambientes de desenvolvimento distintos. Ainda assim, pretendemos também ampliar a quantidade de *features* e produtos gerados pela LPS resultante desse estudo, pois novos cenários de extração de *features* poderiam surgir. Por outro lado, seria importante também analisar a cobertura dos testes executados.

Outra possibilidade para trabalhos futuros é a construção de ferramentas de apoio à aplicação dos *templates*. Podemos implementar uma ferramenta para definir critérios de recomendação de refatoramento para uso dos *templates* propostos. Acreditamos que a automação de *templates* integrada com uma ferramenta de desenvolvimento poderia ajudar a resolver problemas durante a evolução manual das LPS. Geralmente a evolução de LPS envolve a análise e modificação de um grande número de artefatos, além do FM, AM e CK.

Também planejamos executar experimentos controlados usando diferentes desenvolvedores e diferentes LPS para avaliar o tempo durante a evolução das LPS e, consequentemente, avaliar os ganhos de produtividade ao usar nossos *templates* integrados em um ambiente de desenvolvimento com ferramentas de apoio à aplicação dos *templates*.

REFERÊNCIAS

1 BONIFÁCIO, R.; TEIXEIRA, L.; BORBA, P. Hephaestus: A tool for managing product line variabilities. *III SBCARS*, p. 26–34, 2009.

2 SAMPAIO, A.; BORBA, P. Transformation laws for sequential object-oriented programming. *Refinement Techniques in Software Engineering*, Springer, p. 18–63, 2006.

3 NEVES, L.; BORBA, P.; ALVES, V.; TURNES, L.; TEIXEIRA, L.; SENA, D.; KULESZA, U. Safe evolution templates for software product lines. *Journal of Systems and Software*, p. 42–58, 2015.

4 POHL, K.; BÖCKLE, G.; LINDEN, F. J. van D. *Software product line engineering: foundations, principles and techniques.* [S.l.]: Springer Science & Business Media, 2005.

5 BORBA, P.; TEIXEIRA, L.; GHEYI, R. A theory of software product line refinement. *Theoretical Computer Science*, Elsevier, p. 2–30, 2012.

6 BENBASSAT, F.; BORBA, P.; TEIXEIRA, L. Safe evolution of software product lines: Feature extraction scenarios. In: IEEE. *Software Components, Architectures and Reuse (SBCARS), 2016 X Brazilian Symposium on.* [S.l.], 2016. p. 11–20.

7 LINDEN, F. vd; SCHMID, K.; ROMMES, E. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering. Secaucus.* [S.l.]: NJ, USA: Springer-Verlag New York, Inc, 2007.

8 KANG, K. C.; COHEN, S. G.; HESS, J. A.; NOVAK, W. E.; PETERSON, A. S. *Feature-oriented domain analysis (FODA) feasibility study.* [S.l.], 1990.

9 SCHOBBENS, P.-Y.; HEYMANS, P.; TRIGAUX, J.-C.; BONTEMPS, Y. Generic semantics of feature diagrams. *Computer Networks*, Elsevier, v. 51, n. 2, p. 456–479, 2007.

10 CZARNECKI, K.; PIETROSZEK, K. Verifying feature-based model templates against well-formedness ocl constraints. In: ACM. *Proceedings of the 5th international conference on Generative programming and component engineering.* [S.l.], 2006. p. 211–220.

11 TEIXEIRA, L.; BORBA, P.; GHEYI, R. Safe composition of configuration knowledge-based software product lines. *J. Syst. Softw.*, p. 1038–1053, 2013.

12 THAKER, S.; BATORY, D.; KITCHIN, D.; COOK, W. Safe composition of product lines. In: *Proceedings of the 6th International Conference on Generative Programming and Component Engineering.* [S.l.: s.n.], 2007. (GPCE '07), p. 95–104.

13 APEL, S.; BATORY, D.; KÄSTNER, C.; SAAKE, G. *Feature-Oriented Software Product Lines.* [S.l.]: Springer, 2013.

14 BORBA, P. 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.* [S.l.: s.n.], 2011. (GTTSE'09), p. 1–26.

15 OWRE, S.; SHANKAR, N.; RUSHBY, J. M.; STRINGER-CALVERT, D. W. PVS Language Reference. *SRI International*, 2001. Version 2.4.

16 ALVES, V.; GHEYI, R.; MASSONI, T.; KULESZA, U.; BORBA, P.; LUCENA, C. Refactoring product lines. In: *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*. [S.l.: s.n.], 2006. (GPCE '06), p. 201–210.

17 GHEYI, R.; MASSONI, T.; BORBA, P. Algebraic laws for feature models. *J. UCS*, p. 3573–3591, 2008.

18 PASSOS, L.; GUO, J.; TEIXEIRA, L.; CZARNECKI, K.; WĄSOWSKI, A.; BORBA, P. Coevolution of variability models and related artifacts: a case study from the linux kernel. In: ACM. *Proceedings of the 17th International Software Product Line Conference*. [S.l.], 2013. p. 91–100.

19 KASTNER, C.; APEL, S.; BATORY, D. A case study implementing features using aspectj. In: IEEE. *Software Product Line Conference, 2007. SPLC 2007. 11th International*. [S.l.], 2007. p. 223–232.

20 KOLB, R.; MUTHIG, D.; PATZKE, T.; YAMAUCHI, K. A case study in refactoring a legacy component for reuse in a product line. In: IEEE. *21st IEEE International Conference on Software Maintenance (ICSM'05)*. [S.l.], 2005. p. 369–378.

21 LIU, J.; BATORY, D.; LENGAUER, C. Feature oriented refactoring of legacy applications. In: ACM. *Proceedings of the 28th international conference on Software engineering*. [S.l.], 2006. p. 112–121.

22 TRUJILLO, S.; BATORY, D.; DIAZ, O. Feature refactoring a multi-representation program into a product line. In: ACM. *Proceedings of the 5th international conference on Generative programming and component engineering*. [S.l.], 2006. p. 191–200.

23 COLE, L.; BORBA, P. Deriving refactorings for AspectJ. In: ACM. *Proceedings of the 4th international conference on Aspect-oriented software development*. [S.l.], 2005. p. 123–134.

APÊNDICE A – TEMPLATES PARA EVOLUÇÃO SEGURA DE LINHAS DE PRODUTOS DE SOFTWARE

Neste apêndice, resumimos todos os *templates* de evolução segura propostos neste trabalho.

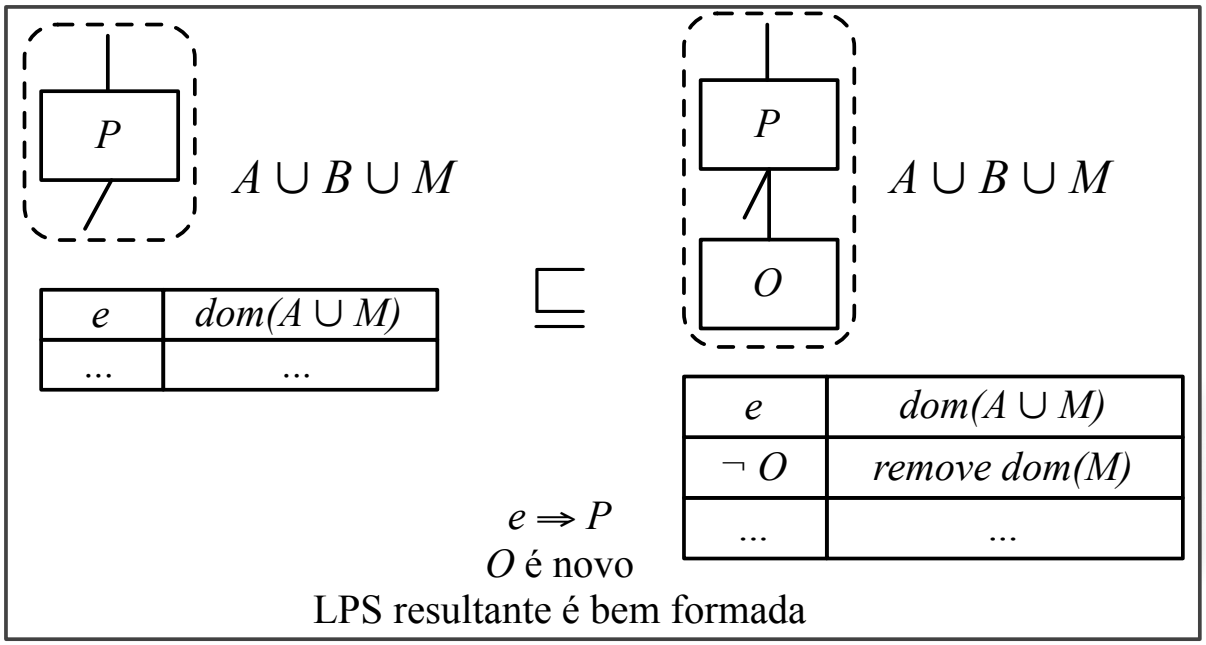

Fig. A.1 – Template transformar artefato existente em nova feature

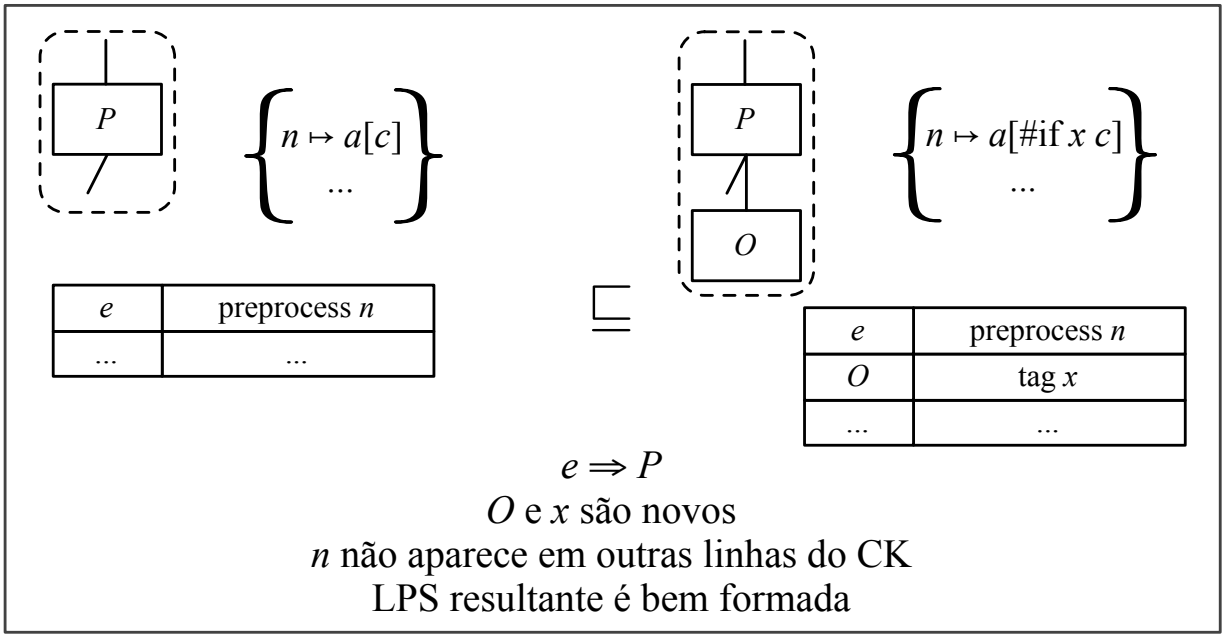

Fig. A.2 – Template transformar artefato pré-processado em nova feature

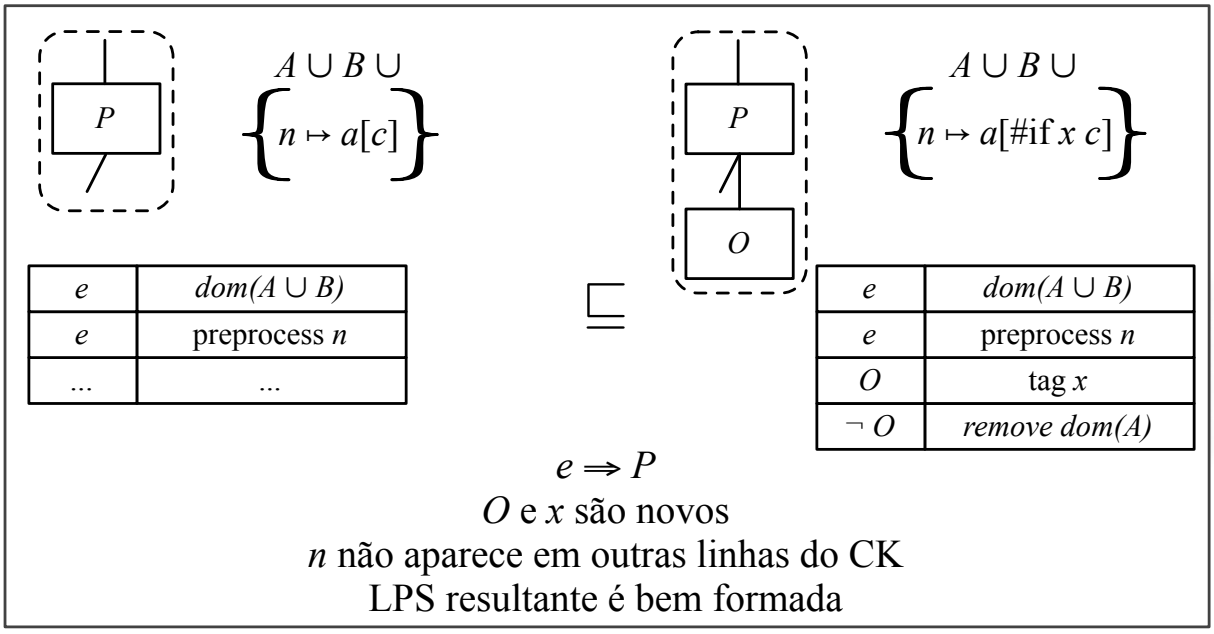

Fig. A.3 – Template transformar múltiplos artefatos existentes em nova feature