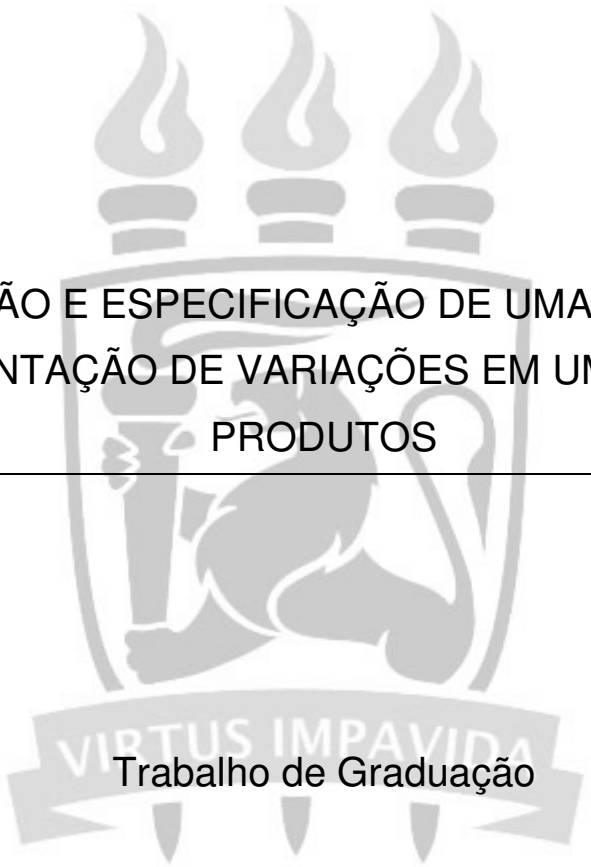
UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA



DEFINIÇÃO E ESPECIFICAÇÃO DE UMA DSL PARA IMPLEMENTAÇÃO DE VARIAÇÕES EM UMA LINHA DE PRODUTOS

Trabalho de Graduação

MARCOS AUGUSTO CAMELO FARIAS XAVIER

Orientador: Paulo Henrique Monteiro Borba
Co-orientador: Leopoldo Motta Teixeira

Recife, 2010

UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA

DEFINIÇÃO E ESPECIFICAÇÃO DE UMA DSL PARA IMPLEMENTAÇÃO DE VARIAÇÕES EM UMA LINHA DE PRODUTOS

Trabalho de Graduação

MARCOS AUGUSTO CAMELO FARIAS XAVIER

Projeto de Graduação apresentado no Centro de Informática da Universidade Federal de Pernambuco por Marcos Augusto Camelo Farias Xavier, como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Paulo Henrique Monteiro Borba
Co-orientador: Leopoldo Motta Teixeira

Recife, 2010

"A persistência é o caminho do êxito."
Charles Chaplin

AGRADECIMENTOS

Agradeço primeiramente a Deus que me deu forças ao longo de todos esses anos. Agradeço a minha família que esteve sempre presente acompanhando de perto meus estudos, me dando apoio sempre que tive necessidade.

Aos meus amigos Felype, Léo e Pedro com quem fiz a maior parte de meus trabalhos da graduação e pela amizade construída ao longo desses anos. A Yane e Everson que também tiveram uma participação importante na minha vida acadêmica e como amigos.

Agradeço também aos meus amigos Edson e Luciano que durante os últimos meses suportaram meu stress e mau humor devido aos trabalhos da faculdade. Às minhas amigas Alê, Marta e Mari que estiveram sempre presentes durante esses anos.

Ao meu co-orientador Leopoldo que me ajudou me dando dicas, revisando minha monografia e auxiliando com detalhes de implementação. Ao meu orientador Paulo que me deu suporte a realizar este trabalho e meu deu idéias para criar soluções mais eficientes. A Carlos Eduardo que me auxiliou com a configuração e uso do Stratego/XT. Novamente a Felype que me ajudou na implementação de funcionalidades do Hephaestus e que em deu dicas úteis para esse trabalho.

E agradeço ao pessoal da Fast que me concedeu férias nesse último mês para que eu conseguisse terminar de implementar a solução e escrever minha monografia.

RESUMO

O uso de linhas de produto de software (LPS) aumenta consideravelmente a produtividade no desenvolvimento de software. Elas consistem em conjuntos de aplicações que possuem funcionalidades em comum e são desenvolvidas a partir de uma mesma base. Seu emprego reduz o esforço e custo para desenvolver, implantar e manter produtos similares. Este trabalho define e especifica uma linguagem de domínio específico que permite a extração e criação de transformações para o software Hephaestus, desenvolvido em Haskell, de forma a criar uma linha de produtos. Tal DSL é definida utilizando a linguagem e ferramenta de programação de transformações Stratego/XT. Com ela extraímos oito transformações gerando produtos distintos para atender a necessidades específicas.

Palavras-chave: Linha de produtos, DSL, Stratego/XT, Haskell.

ABSTRACT

The usage of software product lines (SPL) considerably increases the software development productivity. It consists in a set of applications having the same common functionalities, these being developed from the same core. Its implantation reduces the effort and the cost of development, deployment and maintenance of similar products. This work defines and specifies a domain specific language, that allows extracting and creating transformations for the software Hephaestus, developed with Haskell, generating a product line. Such DSL is defined by the language and toolset for program transformation Stratego/XT. Therefore, we extracted eight transformations which generated distinct programs to support specific needs.

Key-words: Software Product Line, DSL, Stratego/XT, Haskell.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE LISTAGENS

1. INTRODUÇÃO

Linhas de produto de software (LPS) são conjuntos de aplicações que possuem funcionalidades em comum e são desenvolvidas a partir de uma mesma base de artefatos [15]. Sua utilização visa o aumento na produtividade por meio do reuso sistemático de componentes.

Além disso, o desenvolvimento de produtos baseado em LPS diminui os custos para empresa que adota tal técnica. Isso ocorre, uma vez que, cada produto é desenvolvido utilizando o mesmo processo de produção. Dessa forma é mais econômico para a empresa investir em ferramentas para automatizar todo esse procedimento [16]. Entretanto o custo inicial empregado é maior, que o custo para o desenvolvimento de um único produto. Dessa forma, deve-se avaliar a quantidade de variações que serão desenvolvidas a fim de garantir que LPS é a solução mais viável.

Na Figura 1.1 observamos um gráfico que nos mostra a relação do custo de desenvolver, implantar e manter softwares de forma convencional, com respeito a quantidade de produtos desenvolvidos. Nele temos o comparativo de custo entre softwares desenvolvidos de forma convencional e aqueles utilizando uma linha de produtos [31].

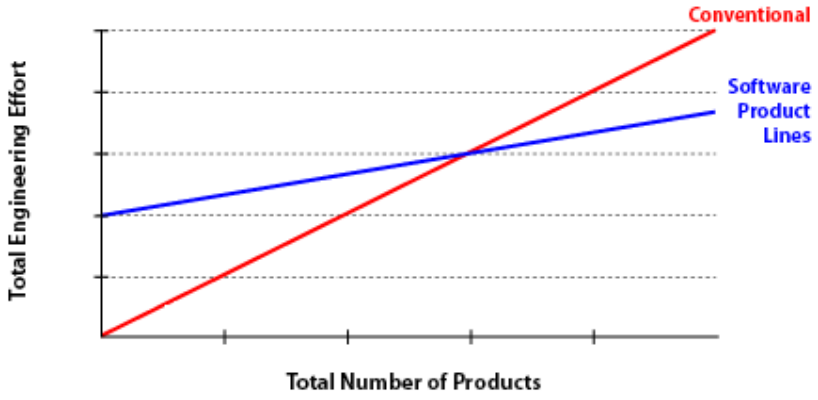


Figura 1.1 Esforço para desenvolvimento de produtos[31]

Existem várias técnicas que dão suporte ao desenvolvimento de LPS utilizando o paradigma de orientação a objetos. Entre elas podemos destacar orientação a aspectos, arquivos de propriedades, compilação condicional e herança. No entanto, quando analisamos linguagens funcionais, percebemos uma deficiência de técnicas consolidadas que nos permitam criar variações em software.

Ao tomarmos Haskell como a linguagem funcional base para nosso estudo percebemos a existência de algumas técnicas, tais como Strafunski, Template Haskell, gramática de atributos, orientação a aspectos e Harpy. Algumas delas não possuem uma versão estável. Outras possuem certas limitações que nos impedem de modularizar completamente um produto específico, para que possamos inserir variações no mesmo. Assim, a criação de uma linguagem de domínio específica é uma alternativa plausível para o desenvolvimento e manutenção de uma LPS.

1.1 Objetivos

Este trabalho tem como propósito a criação de uma linguagem de domínio específico (DSL) a partir da qual será possível extrair variações de um produto, criando uma linha de produtos de software (LPS). O Hephaestus [32], software desenvolvido em Haskell que gera artefatos específicos para um produto a partir de modelos de uma LPS, é utilizado como estudo de caso.

A sintaxe e semântica da linguagem é implementada com o Stratego/XT [19], ferramenta e linguagem de transformação. Com base nesta linguagem, são extraídas as variações existentes no produto de forma a tornar possível a criação de novos produtos da LPS.

1.2 Estrutura do trabalho

Este trabalho é composto por sete capítulos. No Capítulo 2, são analisadas técnicas existentes em Haskell que permitem criar variações em software. No capítulo 3 propomos uma nova linguagem como solução mais apropriada para criação de uma linha de produtos em Haskell. Além disso, especificamos toda sua sintaxe e sua implementação com Stratego/XT.

No Capítulo 4 abordamos os códigos gerados a partir da DSL utilizando as estratégias e transformações de Stratego/Xt. No capítulo 5 extraímos todas as *features* do Hephaestus utilizando a linguagem proposta, de forma que podemos gerar diferentes produtos.

No Capítulo 6 observamos os resultados obtidos após a extração das *features* e as vantagens da utilização da DSL. Por fim, no último capítulo apresentamos as considerações finais acerca do trabalho.

2. ESTUDO DE TÉCNICAS PARA VARIAÇÕES EM HASKELL

Existem diversas técnicas já consolidadas para implementação de variações de linhas de produtos de software desenvolvidas em linguagens orientadas a objeto. Podemos destacar algumas: programação orientada a aspectos, compilação condicional, herança, polimorfismo, arquivos de propriedades e bibliotecas estáticas [1]. No entanto, ao usar uma linguagem funcional, o desenvolvimento de código visando LPS exige um esforço maior, e as técnicas para tal não se encontram bem amadurecidas.

Especificamente em Haskell, existem algumas técnicas que nos permitem adicionar as facilidades da metaprogramação, ou seja, o programa possibilita a geração de código em tempo de compilação [26]. No entanto, tais técnicas possuem certas limitações. Nos próximos tópicos iremos analisar as características dessas técnicas estudadas.

2.1 Strafunski

Strafunski é uma ferramenta de programação funcional baseada em Haskell que permite programação genérica e processamento da linguagem [3]. Teve início como uma biblioteca com ferramentas de suporte a travessias genéricas em árvores gramaticais de forma escalar [2]. Ela sofreu uma evolução e passou a abranger formas de integração com componentes externos tais como *parsers*, *pretty printers* e ferramentas de visualização gráficas.

Strafunski é baseada na noção de uma estratégia funcional [3]. Trata-se de funções que podem fazer análise sobre termos de qualquer tipo. Porém, também permitem combinar tipos específicos a um comportamento genérico, produzindo uma função genérica para um tipo específico de entrada.

Esta técnica é bastante apropriada para o desenvolvimento de processadores de linguagens. Pro exemplo pode-se se utilizá-la para engenharia reversa de Cobol, calcular métricas de um código em Java ou para própria re-engenharia de Haskell [2].

No entanto, as capacidades genéricas de Strafunski são menos ambiciosas que outras abordagens de programação funcional. Ela está limitada a permitir uma estratégia de

programação útil para construção de transformações e consultas sobre grandes conjuntos não parametrizados de tipos algébricos, tais como representações abstratas de sintaxe [32].

2.2 Harpy

Harpy consiste em uma biblioteca para geração de código Haskell em tempo de execução [4]. Ela permite escrever *assembly inline*. Ou seja, é possível que código de baixo nível seja incorporado a linguagem de mais alto nível.

No entanto, o uso de tal técnica é inviável, pois seria necessário escrever todo o código referente às variações do produto utilizando instruções do *Assembler x86*. Tal código seria transformado em Haskell em tempo de execução. Porém, a codificação é bastante árdua e de baixa legibilidade, o que implica em problemas de manutenção do software. Por exemplo, a Figura 2.1 [4] mostra uma função que calcula o fatorial de um número utilizando *assembly inline*.

```
4  fac :: CodeGen e s ()
5  fac = do loopTest  ← newLabel
6           loopStart ← newLabel
7           ensureBufferSize 160
8           push ecx
9           mov  ecx (Disp 8, esp)
10          mov  eax (1 :: Word32)
11          jmp  loopTest
12          loopStart @@ mul ecx
13          sub  ecx (1 :: Word32)
14          loopTest @@ cmp ecx (0 :: Word32)
15          jne  loopStart
16          pop  ecx
17          ret
```

Figura 2.1 Exemplo de função fatorial em assembly inline[6]

2.3 Gramáticas de atributos

Gramáticas livres de contexto vêm sendo utilizadas para permitir a análise sintática de linguagens de programação. Contudo, nem sempre é possível representar, neste tipo de gramática, restrições necessárias a algumas linguagens. Por exemplo, exigir que todas as variáveis estejam declaradas antes de seu uso ou verificar se os tipos envolvidos em uma expressão são compatíveis [5]. Entretanto, há mecanismos alternativos que podem ser incorporados às ações durante a análise. Por exemplo, interações com tabelas de símbolos, que permitem complementar a funcionalidade da análise sintática.

As gramáticas de atributos têm um objetivo similar ao exposto, adicionando semântica a uma gramática livre de contexto. Apesar de ser relativamente simples descrever a sintaxe de uma linguagem usando gramáticas livres de contexto, descrever sua semântica é uma atividade consideravelmente complexa [6]. Gramáticas de atributos especificam a semântica da linguagem decorando tais gramáticas com alguns atributos.

Além disso, elas descrevem encaminhamentos em árvores com o propósito de computar alguns valores [6]. Basicamente, uma gramática de atributos declara que valores precisam ser computados. Assim, quando a linguagem é processada por algum avaliador sintático, os nós da árvore sintática abstrata são avaliados [7].

As gramáticas de atributos provêem um *framework* para programação orientada a aspectos em linguagens funcionais, técnica que será discutida no próximo tópico [6]. Elas permitem quebrar o código em aspectos, que consistem em partes do código dotadas de um comportamento semelhante, e espalhar definições de atributos em diferentes arquivos agrupando-os de acordo com o aspecto.

2.4 Programação orientada a aspectos

POA é um paradigma emergente que ajuda o usuário a modularizar *crosscutting* concerns. Ou seja, permite separar e organizar o código de acordo com sua importância para a aplicação (separação de *concerns*). Tal abordagem é comumente utilizada em linguagens orientadas a objeto para facilitar variabilidade em código e permitir a implementação de linhas de produtos de software.

No entanto, essa técnica ainda não se encontra bem consolidada para linguagens funcionais. Em Haskell especificamente, existe uma abordagem que utiliza *Type classes* suportadas pelo *Glasgow Haskell Compiler* (GHC), comumente utilizadas para permitir polimorfismo *ad-hoc*, mais conhecido por *overloading* [5]. A ideia é transformar os *advices*, comportamentos que são adicionados ao código base, em instâncias de Haskell [9]. A Figura 2-2 apresenta um exemplo de transformação de *advices* [9].

```
-- sortedness aspect
N1@advice #insert# ::  Ord a => a -> [a] -> [a] =
  \x -> \ys ->
     let zs = proceed x ys
     in if (isSorted ys) && (isSorted zs)
        then zs else error "Bug"
      where
       isSorted xs = (sort xs) == xs
is turned into
instance Ord a => Advice N1 (a -> [a] -> [a]) where
  joinpoint _ insert =  \x -> \ys ->
   let zs = insert x ys
   in if (isSorted ys) && (isSorted zs)
      then zs else error "Bug"
    where
         isSorted xs = (sort xs) == xs
```

Figura 2.2 Exemplo de transformação de Advice em Instance[9]

Uma outra abordagem, intitulada AspectH, consiste em uma extensão da linguagem Haskell com conceitos de orientação a aspectos, implementando *Aspect Oriented Programming* (AOP) através de *pointcuts* e *advices* [10]. Trata-se de uma implementação semelhante a AspectJ [27] e foi projetada para atuar em programas Haskell que utilizam mônadas.

2.5 Template Haskell

Consiste em uma extensão da linguagem funcional Haskell que permite adicionar as facilidades da metaprogramação, ou seja, o programa possibilita a geração de código em tempo de compilação [11]. Template Haskell provê novos recursos à linguagem que nos permitem converter tanto sintaxe concreta, isto é, o que o programador usa quando escreve

código Haskell, quanto árvores de sintaxe abstrata. Estas consistem em estruturas de dados em árvores que representam estruturas sintáticas de cadeias, de acordo com alguma gramática formal [28].

Estas árvores de sintaxe abstrata são representadas usando *datatypes* Haskell e, em tempo de compilação, elas podem ser manipuladas pelo código Haskell. Isso permite que o programador converta código da sintaxe concreta para uma árvore de sintaxe abstrata [13]. Além de possibilitar transformá-lo e juntá-lo de volta (converter novamente), ou mesmo produzir um código totalmente novo enquanto o compilador está compilando seu módulo.

Template Haskell pode ser visto como um pré-processador de Macros para Haskell [12]. Para cada macro criado no código deve existir uma função em um módulo separado que definirá as expressões a serem avaliadas e transformadas em código Haskell sem os macros.

As funções definidas devem ter o tipo `Q Expr`, onde o tipo Q representa um *Quotation Monad* e o tipo `Exp` é a representação de uma expressão em Haskell. Tal expressão não é uma *string*, e sim uma estrutura recursiva representando uma árvore sintática abstrata de expressões. O valor da `Exp` pode ser convertido para string que conterá o código Haskell de fato, para tal, pode-se fazer o uso da função `pprint` . Já esses símbolos `[| ... |]` fazem o papel contrário , ou seja, ele recebem código original de Haskell e retornam um estrutura `Exp` que o representa [14]. Podemos observar na listagem 2.1 exemplos de tais recursos oferecidos.

```
ghci> runQ [| 1 |] >>= print
LitE (IntegerL 1)

ghci> runQ [| \x _ -> x |] >>= print
LamE [VarP x_0,WildP] (VarE x_0)

ghci> runQ [| \x _ -> x |] >>=
putStrLn.pprint
\x_0 _ -> x_0
```

Listagem 2.1 Exemplos de conversões de expressões em template Haskell.

Assim, para permitir a concepção de variações em código, seria necessário criar macros com suas devidas funções de acordo com as diferentes *features* do produto. No entanto, tal abordagem apresenta alguns problemas. O primeiro deles é que existe uma

dependência entre o módulo principal e o módulo onde estão definidas as funções, uma vez que é necessário importar para cada módulo, onde esteja definido um macro, aquele responsável por sua implementação. Além disso, Template Haskell é utilizado para geração de código de funções completas e seria útil caso o código do produto, para o qual se pretende produzir uma LPS, estivesse bem modularizado. No entanto, como no código são encontrados muitos concerns que se cruzam em diversos módulos, a extração das *features* que agrupam cada *concern* torna-se inviável utilizando-se tal técnica.

2.6 Conclusão

As técnicas apresentadas nos tópicos anteriores forem estudadas com o objetivo de extrair as variações no Hephaestus. Algumas delas, como o Template Haskell, chegaram a ser parcialmente implementadas. Contudo, não obtivemos o resultado esperado. Dessa forma foi criada uma linguagem de domínio específico para extração das transformações do produto. Nos próximos capítulos apresentaremos a semântica e sintaxe da linguagem, bem como sua evolução.

3. DEFINIÇÃO DA DSL TRANSKELL

A aplicação Hephaestus possui diversas transformações que permitem ao usuário gerar artefatos específicos para um produto a partir de modelos de uma linha de produto de software [17]. Por exemplo, podemos gerar uma especificação de casos de uso para um produto particular. Contudo, nem sempre é necessário que todas estas transformações estejam presentes. Algumas podem não fazer sentido para um cliente em específico. Portanto, o objetivo deste trabalho é transformar o Hephaestus em uma LPS. Para isso, tratamos cada transformação como uma *feature* do produto. Assim, extraímos todas as *features* existentes para permitir que sejam gerados softwares com diferentes configurações. Além disso, facilitamos ao desenvolvedor a criação de novas transformações sem a necessidade de alterações em diversos módulos do Hephaestus como ocorre no cenário atual.

A Figura 3.1 apresenta o modelo de *features* do Hephaestus, nele observamos oitos diferentes transformações:

- *SelectUseCases*: Permite a criação de um documento de casos de uso específico de uma instância de produto. Para isso, o usuário seleciona os casos de uso específicos para cada *feature* e fornece uma seleção de *features* que compõem o produto.
- *SelectScenarios*: Cada caso de uso, por sua vez, é formado por diversos cenários. Assim, esta transformação dá liberdade ao usuário de customizar casos de uso e gerar documentos para uma instância da aplicação.
- *EvaluateAspects*: Os cenários também podem sofrer variações, para isso são criados *advices.* Estes consistem em partes reusáveis de cenários que podem ser inseridas em pontos específicos dos mesmos a partir da interceptação de *pointcuts*.
- *BindParameter:* Dentro de um cenário, ainda, podemos definir parâmetros que variam de acordo com a *feature* escolhida. Por exemplo: a resolução de um jogo para *mobile*. Esta varia de acordo com o modelo do celular para o qual o software é gerado. Assim, a transformação é responsável por atribuir valores aos parâmetros definidos de acordo com a *feature* selecionada.

- *SelectComponents:* Seleciona componentes a partir de identificadores e os move para pasta de *output* do projeto.
- *SelectAndMoveComponent:* É responsável por mover um componente - sendo este uma classe, aspecto, arquivo de configuração entre outros - de um local para outro para ser utilizado no produto final.
- *CreateBuildEntries:* Utilizado para adicionar entradas e gerar o arquivo de *build* da instância do produto.
- *PreprocessFiles:* Permite que uma lista de arquivos sofram um pré-processamento para aplicar compilação condicional. A transformação *CreateBuildEntries* é utilizada como auxiliar para gerar um arquivo com as *tags* de pré-processamento.

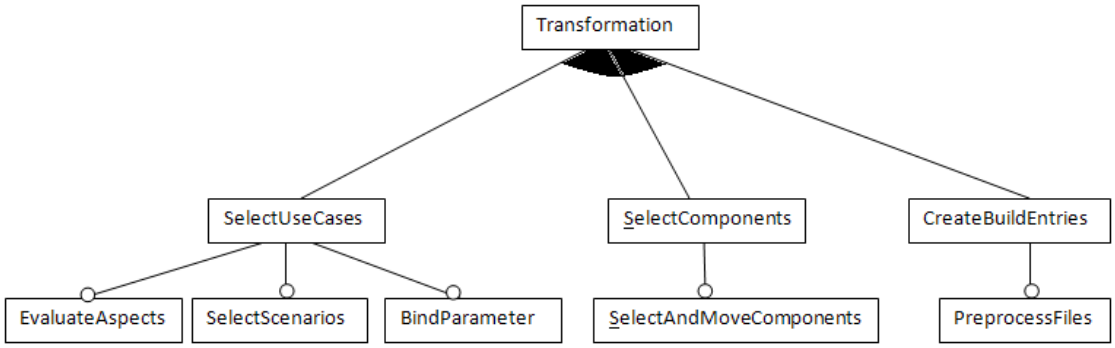


Figura 3.1Modelo de Feature do Hephaestus

Com base em uma análise no código do Hephaestus, observamos que estas transformações não estão modularizadas. Existem diversos *concerns* que se encontram entrelaçados no código. As técnicas apresentadas no capítulo anterior, por sua vez, não nos permitem fazer tal modularização. Sendo assim, a solução escolhida para resolver tal problema foi a criação de uma linguagem de domínio específico (DSL).

A idéia básica de uma DSL consiste em ser uma linguagem computacional voltada para um tipo específico de problema [18], no nosso caso a criação de transformações para a aplicação Hephaestus. Existem duas abordagens diferentes de DSL. Na DSL interna se faz um uso particular de uma linguagem geral dando a mesma uma abordagem de linguagem específica. Por sua vez, na DSL externa temos uma linguagem própria customizada com um *parser* completo para processá-la [18]. Utilizaremos uma DSL externa para resolver o nosso

problema, a mesma será implementada utilizando Stratego/XT, como veremos no próximo tópico.

3.1 Implementação de uma DSL com Stratego/XT

Stratego/XT é uma linguagem e uma ferramenta para transformações de programas. Sua linguagem permite escrever regras de reescrita que possibilitam expressar transformações básicas. Existe a possibilidade também de utilizarmos sintaxe concreta para expressar os padrões das regras de transformação [19].

Um sistema de software de transformação comumente é organizado como um simples *pipeline* de passos de transformação, como podemos observar na Figura 3.1 [20]. Inicialmente um *parser* faz a leitura do texto que é dado como entrada para o programa e o transforma em uma árvore de *parse*, conhecida como árvore de sintaxe abstrata. Posteriormente, essa árvore é percorrida e uma ou várias transformações a modificam. Ao final do *pipeline*, uma *pretty-printer*, aplicação para mostrar os nós de uma árvore de forma legível, transforma a saída da árvore em programa em forma de texto.

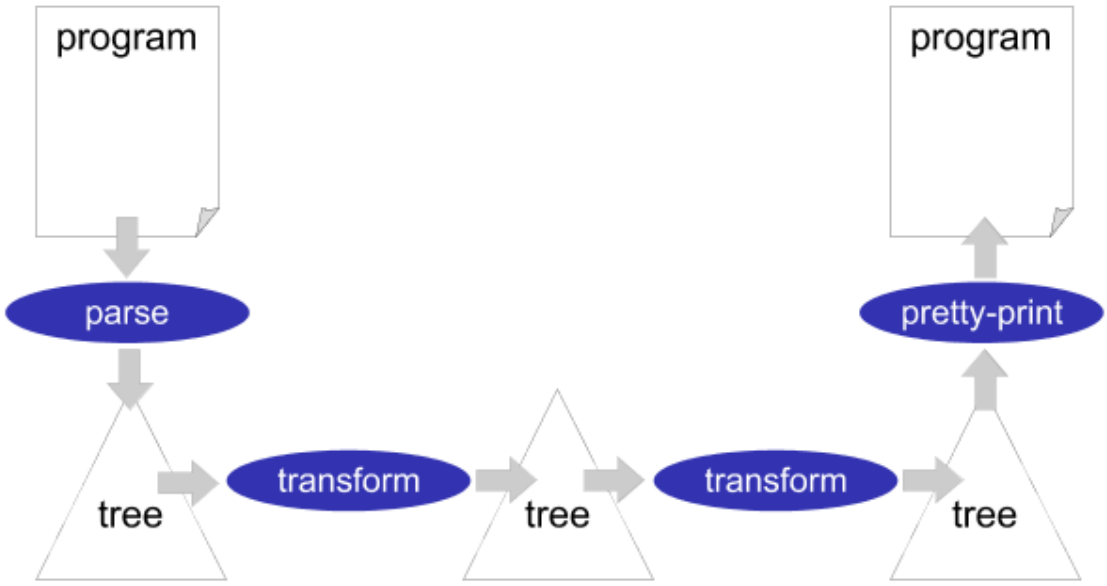


Figura 3.2 Pipeline de um sistema de software de transformação [20]

O sistema de transformação do Stratego/XT não trabalha diretamente com texto, ele utiliza uma representação estruturada, chamada de *Annotated Term Format* (ATerm). A conversão de programas textuais para termos requer um *parser*, este será responsável por

reconhecer se um programa está sintaticamente correto e gerar *ATerms*. O compilador de Stratego emprega um formalismo de definição de sintaxe (SDF) para realizar o *parser* dos programas Stratego. No próximo tópico analisamos a implementação da sintaxe da linguagem através de uma SDF.

3.2 Formalismo para definição da linguagem

A SDF é utilizada para definição da sintaxe de uma DSL. Uma grande vantagem da utilização de SDF consiste nela ser suportada por *Scannerless Generalized LR Parsing.* Por *Scanner parsing* entendemos o fato de não haver a necessidade de uma ferramenta à parte para quebrar, em *tokens*, a entrada antes de realizar o parser. *Generalized LR*, por sua vez, implica que ambigüidade são permitidas, assim ele remover restrições para sub-classes não ambíguas de gramáticas livres de contexto, tal como a classe LR(k) [21]. Assim, o *parsing* de linguagens definidas utilizando SDF opera com caracteres individuais ao invés de *tokens*[22].

3.2.1 Ciclo de vida de uma SDF

As gramáticas SDF, assim como qualquer outro software, têm um ciclo de vida. A Figura 3.2 nos mostra uma visão geral desse ciclo que possui os seguintes passos [23]:

1. O desenvolvedor escreve os módulos, que serão explicados posteriormente, de uma gramática SDF. Neles estão contidos regras de produção e construtores de desambiguações.
2. Os módulos são concatenados em uma definição única.
3. A definição da sintaxe é então dada para uma ferramenta que transforma de SDF para *table* (*sdf2table*):
    a. A definição da sintaxe é checada para detecção de erros básicos através da ferramenta *sdf-checker*.
    b. A sintaxe bem como os construtores de desambiguações são “normalizados” para “*kernelSDF*”. Este trata-se de uma representação intermediária que envolve a remoção de uma estrutura modular e simplificação de construções complexas da linguagem [29].

c. A sintaxe é usada para gerar uma tabela de analisador sintático LR(1). Este se trata de um algoritmo de análise sintática para gramáticas livres de contexto, que lê a entrada da esquerda para direita e produz derivações mais a direita [24]. Por sua vez, as regras de desambiguações são utilizadas para filtrar algumas reduções a partir da tabela de *parse*.

4. A tabela resultante pode ser utilizada pela ferramenta *sglr* para realizar o parse de *strings.*
5. A ferramenta *slgr* lê um arquivo que contém o programa escrito na linguagem criada e dá como saída uma floresta (caso haja ambiguidade), ou uma única árvore de parse. Caso a entrada esteja incorreta ele retorna um parse de erro.
6. Por fim outras ferramentas utilizam a saída para fazer transformações da linguagem.

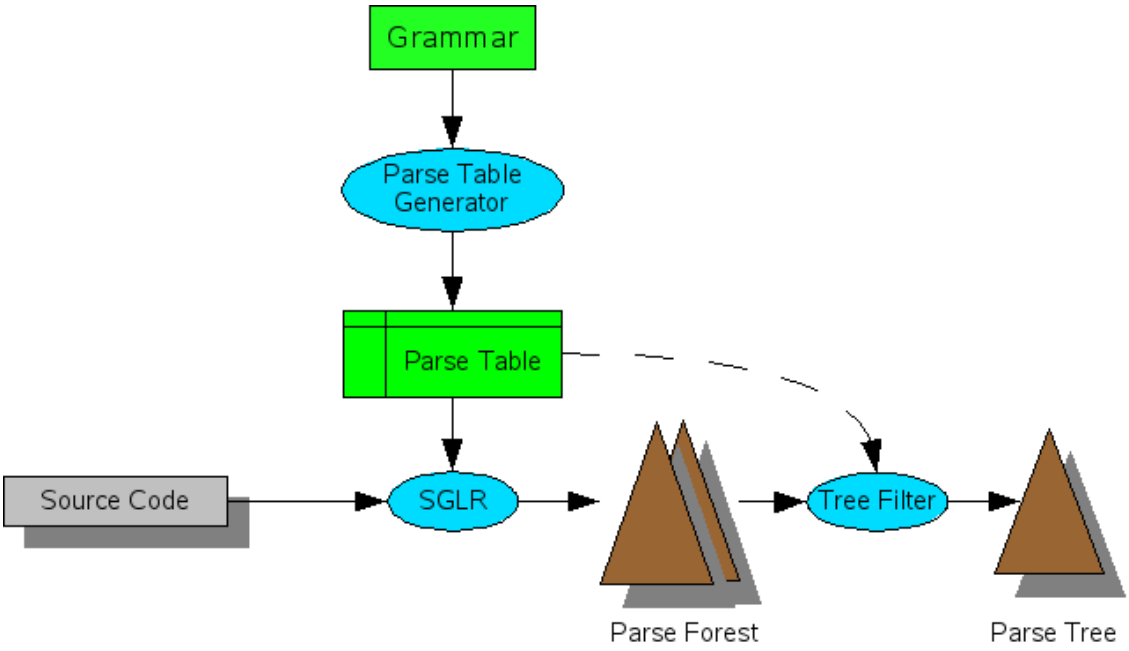


Figura 3.3 Visão geral do fluxo de uma Definição com SDF [23]

3.2.2 Estrutura de uma SDF

Basicamente uma SDF é estruturada em módulos que posteriormente são agrupados em uma única definição. Cada módulo é composto por seções, das quais podemos destacar duas: *lexical syntax* e *context-free syntax*. Na primeira estão descritos os elementos básicos do programa. Nesta seção comumente se define as associações entre alguns símbolos variáveis da gramática, como por exemplo, os identificadores e variáveis da linguagem [23]. Na seção *context-free syntax*, por sua vez, se descreve a estrutura de alto nível de sentenças da linguagem. Basicamente, nela temos um conjunto de regras de produção [23]

No tópico 3.3 apresentamos uma análise do código do Hephaestus. Em seguida observamos os códigos que necessitam ser gerados. Estes são reescritos na nova linguagem de domínio específico, de um modo mais simples, de forma que o programador escreve o mínimo possível. A DSL criada é chamada de *Transkell*, uma menção a transformações do Hephaestus em Haskell.

3.3 Análise do código do Hephaestus

Como foi mencionado anteriormente o propósito de criação da DSL é permitir escrever as transformações do Hephaestus de forma mais modularizada. Uma transformação basicamente consiste em funções que recebem alguns argumentos como entrada e geram algum artefato para uma instância da linha de produtos de software.

Sendo assim, analisamos algumas seções que são importantes no momento de criação de uma nova transformação. Inicialmente precisamos informar a entrada para a transformação. No caso do Hephaestus elas são informadas como extensão do *configuration knowledge*, como *tags* do XML. Na Listagem 3.1 observamos um fragmento do *configuration knowledge* para a transformação *PreprocessFiles.* Esta tem por objetivo pré-processar, com compilação condicional, uma lista de arquivos passados como entrada. A *tag expression* indica o nome da *feature* que uma vez selecionada deve sofrer a transformação. Para tal temos a *tag transformation* que possui um nome, indicado pela *tag name* e parâmetros, passados na *tag arg.*

```
<expression>banking</expression>

<transformation>

        <name>preprocessFiles</name>

        <args>Main.java, Account.java</args>

</transformation>
```

Listagem 3.1 Exemplo de entrada para transformação *PreprocessFiles*

No módulo *Transformations.Parsers.XML.XmlConfigurationKnowledge* existe uma função de parser *xml2Transformation* que fará chamadas às devidas transformações, como observamos na Listagem 3.2.

```
xml2Transformation :: XmlTransformation -> ParserResult GenT
xml2Transformation t =
 let as = splitAndTrim ',' (tArgs t)
 in
 case tName t of
"preprocessFiles" -> Success (GenT (PreProcessor as))
```

Listagem 3.2 *Parser* do xml do *configuration knowlege*

Os argumentos passados no XML, por sua vez, serão repassados para uma função que realizará a transformação. Assim, existem alguns elementos que precisam ser definidos. Tomando como exemplo *PreProcessFiles*, é preciso criar um tipo básico para representar os caminhos do arquivo passado, como observamos na Listagem 3.3.

```
module BasicTypes

where

type Path = String
```

Listagem 3.3 Tipo básico para *PreProcessFiles*

Além disso, na Listagem 3.4 definimos um *Data PreProcessor* que define uma lista de paths e armazena, assim, a entrada da transformação.

```
data PreProcessor = PreProcessor {

     paths :: [Path]

}
```

Listagem 3.4 Exemplo do *Data PreProcessor*

No core do Hephaestus existe uma estrutura que representa instâncias da LPS chamada de InstanceModel. Uma nova transformação criada pode acrescentar modelos a estrutura existente. Na Listagem 3.5 verificamos que ela armazena uma lista de Strings, para os arquivos pré-processados.

```
data InstanceModel = InstanceModel {

      fc  :: FeatureConfiguration,

      req :: RequirementModel,

      preProcessFiles :: [String]

} deriving (Data, Typeable)
```

Listagem 3.5 Modelo de instância do Hephaestus

Uma vez incluída esta nova entrada no *InstanceModel*, precisamos definir como ela será inicializada. A função *build* do módulo *ConfigurationKnowledge.Interpreter* instancia um produto a partir do modelo de entrada. Ela chama cada transformação que deve ser aplicada a uma configuração de feature. Na Listagem 3.6 temos a inicialização para a lista preprocessFiles como uma lista vazia de Haskell: [].

```
build :: FeatureModel  -> FeatureConfiguration  -> ConfigurationKnowledge -> SPLModel ->
InstanceModel
build fm fc ck spl = stepRefinement ts spl emptyInstance
 where
  ts            = tasks ck fc
  ucmodel       = splUCM spl
  emptyUCM      = ucmodel { useCases = [] , aspects = [] }
  emptyReq      = RM { reqs = [] }
  emptyInstance = InstanceModel fc emptyReq emptyUCM [] [] []
```

Listagem 3.6 Inicialização do *InstanceModel*

Na Listagem 3.7 temos a implementação de tal transformação. Para isso foi criado um *Instance* de *Trasformation* em um módulo específico para transformação.

```
module Trasnformation.PreProcessFiles
where
...
instance Transformation PreProcessFiles where
 (<+>) (PreProcessor e) spl product =
        let es = preProcessFiles product
        in product { preProcessFiles = es ++ e }
```

Listagem 3.7 Fragmento da *Instance PreProcessFiles*

Por fim nós precisamos definir a saída da transformação. Neste caso, consiste em uma chamada ao um programa externo em Java para fazer o pré-processamento da lista de arquivos. Na Listagem 3.8 temos um fragmento do código no módulo *ExportProduct* que é responsável pela chamada descrita.

```
exportProduct :: FilePath -> FilePath -> InstanceModel -> IO ()
exportProduct s o p = do
preprocessFiles (o ++ "/build.lst") (preProcessFiles p) o

preprocessFiles :: String -> [String] -> String -> IO()
preprocessFiles _ [] _ = return()
preprocessFiles flags files outputFolder = do
...
```

Listagem 3.8 Saída da transformação *PreProcessFiles*

Outras transformações ainda podem necessitar de alterações na interface gráfica do Hephaestus. A exemplo disso temos SelectUseCases que faz modificações nos artefatos de saída de acordo com os casos de uso que foram selecionados. Esta necessita que o usuário forneça um arquivo que possui estruturadamente os casos de uso para uma linha de produto. Isso é feito através de um componente (*filechooser*) que permite selecionar um arquivo contido no disco do usuário. Na listagem 3.9 observamos alguns fragmentos do módulo *Main* onde está definida a *Gui.*

```
module Main where
...

-- retrieves the file chooser elements.
   rmfc  <- xmlGetWidget f castToFileChooserButton "rmFileChooser"
   ucmfc <- xmlGetWidget f castToFileChooserButton "ucmFileChooser"

...
-- returns the GUI instance
   return $ GUI {
                window       = w,
                ckWindow     = ckw,
                ucmFChooser = ucmfc,

...
parseAllInputModels gui = do
   cDir <- getCurrentDirectory
   let ns = normalizedSchema cDir
   prm <- getInputModel ((rmFChooser  gui), parseRequirementModel (ns rmSchema), "Requirement
model not selected.")
   pum <- getInputModel ((ucmFChooser gui), parseUseCaseFile (ns ucSchema), "Use case model not
selected.")

...
```

Listagem 3.9 Fragmento da Gui de SelectUseCases

Após essa análise podemos perceber que o código relacionado a uma transformação encontra-se espalhado ao longo de vários módulos. Nos próximos tópicos definimos os elementos sintáticos da DSL, agora que temos uma visão geral do código que precisará ser gerado e acoplado ao *core* do Hephaestus.

3.4 Seção Lexical da SDF

A seção responsável pela *lexical syntax* foi definida em um módulo separado da SDF chamado de Common. Nela definimos um símbolo especial chamado de *LAYOUT*, ele é utilizado para formatação da entrada e só pode ser definido dentro da seção lexical. Na listagem 3.10 observamos que a forma como ele foi definido. Ou seja, Layout pode ser entendido como os caracteres espaço vazio, tabulação ou quebra de linha.

```
module Common
  lexical syntax
    [\ \t\n]   -> LAYOUT
```

Listagem 3.10 Definição de LAYOUT

Na definição da sintaxe livre de contexto entre cada elemento descrito no lado esquerdo de uma regra de produção é inserido um símbolo de layout antes da geração da

árvore sintática pelo analisador sintático [23], um exemplo disso pode ser visto na Figura 3.1 [25].

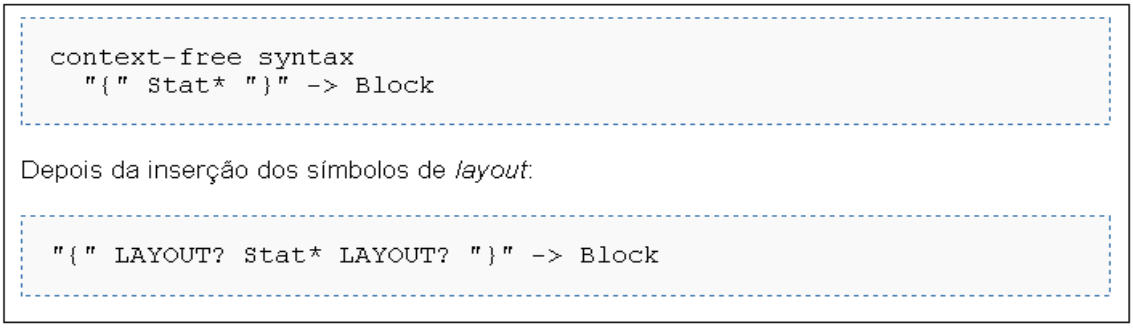


Figura 3.4 Exemplo do emprego do símbolo *LAYOUT* [25]

Para evitar ambigüidades na linguagem definimos algumas restrições nesse módulo. Na Listagem 3.11 o símbolo "-/-" indica que uma determinada produção não pode ser seguida imediatamente pelos caracteres que estiverem do lado direito entre colchetes.

```
context-free restrictions
  LAYOUT? -/- [\ \t\n\r]
  LAYOUT? -/- [\/].[\*]
```

Listagem 3.11 Restrições para definição de LAYOUT

Na Listagem 3.12 temos a definição de identificadores (*Identifier*) que é utilizada em muitos casos, como por exemplo, para indicar o nome da transformação. Ele está definido como uma letra seguida ou não de uma sequência de letras ou números. Temos também a definição de *Extension*, que pode ser "*.hs*" ou "*.lhs*". Trata-se de extensões de arquivos Haskell e são utilizadas para restringir os tipos de arquivos que serão importados na seção livre de contexto como veremos mais adiante.

```
[A-Za-z][A-Za-z0-9]* -> Identifier
".hs"  -> Extension
".lhs" -> Extension
```

Listagem 3.12 Definição de extensão e identificadores

Encontramos outras definições importantes na Listagem 3.13. A definição de *String* é dada como zero ou inúmeras ocorrências - simbolizado na SDF por "*"- da produção *StrChar*. Este, por sua vez, pode ser qualquer caractere exceto quebra de linha, aspas duplas e o caractere especial \r. Temos duas definições de componentes de *Gui,* que pode ser *textbox* ou *fileChooser*. Temos ainda a definição *HaskellWord*, semelhante a de String,

mas nesse caso é possível escrever qualquer sequência de caracteres. Esta definição é utilizada para permitir ao desenvolvedor escrever qualquer código em Haskell no bloco *Code*, que definiremos na *context-free syntax.* E, por fim, algumas derivações que devem ser rejeitadas como, por exemplo, um booleano ("*true*" ou "*false*") não pode ser interpretado como um *Identifier*, bem como os componentes de Gui também não devem ser entendidos como *Strings* ou identificadores.

```
StrChar*   -> String
~[\"\n\r] -> StrChar
HWord* -> HaskellWord
~[\21] -> HWord  %% Just rejects an unknown character
"textBox" -> TextBox
"fileChooser" -> FileChooser

Boolean  ->  Identifier {reject}
FileChooser-> String {reject}
TextBox -> String {reject}
TextBox -> Identifier {reject}
FileChooser-> Identifier {reject}
```

Listagem 3.13 Definições da *Lexical Syntax*

No Apêndice G temos a definição completa do módulo Common, com todas as derivações que serão utilizadas na seção livre de contexto.

3.5 Seção Context-free Syntax

A seção *context-free syntax* é definida em outro modulo que traz o nome da linguagem: module Transkell. Este importa as definições do módulo *Commom*, apresentado anteriormente. Na seção *context-free start-symbols* está definido o símbolo inicial, a partir do qual se iniciam as derivações, conforme podemos verificar na Listagem 3.14.

```
module Transkell
imports Common
exports
  sorts BasicType Input Output SPLInstance Gui Imports PrimitiveType Types
  context-free start-symbols
    Transf
```

Listagem 3.14 Fragmento inicial do modulo Transkell

Como observamos no tópico 3.3, de análise do código do Hephaestus, existem vários blocos de código que precisam ser gerados cada vez que uma nova transformação é

inserida. Esses blocos são definidos como o corpo de uma transformação, que por sua vez deve possuir um nome. Na Listagem 3.15 temos cada um dos elementos que compõe o corpo da transformação. Aqueles seguidos do símbolo de interrogação indicam blocos opcionais. Por sua vez, a expressão *cons,* entre chaves, é utilizada para indicar o nome dos nós da árvore que serão gerados pela ferramenta de *parser*.

```
context-free syntax

     "Transformation" TransfName "{" Body "}" -> Transf {cons("Transformation"),prefer}
     Identifier -> TransfName {cons("TransformationName")}
     BasicType? Output? Input SPLInstance? Gui? Imports HaskellCode? ->  Body
{cons("Body"),prefer}
```

Listagem 3.15 Elementos que compõe uma transformação

Como podemos perceber, para se definir uma transformação utiliza-se a palavra reservada "*Transformation*" seguida de um identificador que indica o nome da mesma. Logo após entre chaves defini-se o corpo da transformação, como temos no exemplo da Listagem 3.16. Nos próximos sub-tópicos definiremos cada um dos blocos que compõe o corpo de uma transformação.

```
Transformation NomeDaTransformação {

...

}
```

Listagem 3.16 Exemplo de codificação de uma transformação

3.5.1 Seção BasicType

Esta seção é opcional e responsável por definir os tipos básicos que serão criados no módulo basicTypes. Esses tipos serão traduzidos como *Type* de Haskell e auxiliam a legibilidade do código. Para isso temos a palavra reservada *BasicType* seguida de chaves que limitam o corpo da seção. Neste, nós definimos uma ou mais declarações, cada uma delas é composta por um identificador seguido de um símbolo de igualdade juntamente com o tipo da declaração. Na Listagem 3.17 temos as derivações necessárias para definir esta seção.

```
     "BasicType" "{" Types "}"-> BasicType {cons("BasicType"),prefer}
     Declaration+ -> Types {cons("Declarations")}
     IdDeclaration "=" Type -> Declaration {cons("Declaration")}
     Identifier -> IdDeclaration {cons("TypeName")}
```

Listagem 3.17 Derivações da seção BasicType

Na Listagem 3.18 temos um exemplo de codificação do BasicType. Nele definimos Id e Nome como identificadores do tipo String. O tipo além de String pode ser inteiro, booleano, ou mesmo tuplas e listas.

```
BasicType {
      Id = String
      Name = String

}
```

Listagem 3.18 Exemplo de codificação de BasicType

3.5.2 Seção Output

A seção opcional de *Output* tem por objetivo permitir ao usuário escrever a saída da transformação. Ela gera código que será incorporado ao módulo *ExportProduct.* Assim, temos a palavra reservada *Output,* seguida do corpo do bloco delimitado por chaves. Neste basicamente temos:

1. Seção *Imports* que permite ao usuário escrever zero ou mais importações de código. A sintaxe de um *Import* é bastante simples, como observamos abaixo:

   ```
   Import(”caminhoAtehAlgumArquivo.hs”)
   ```

   O arquivo passado é limitado a arquivos Haskell (.hs ou .lhs). E neste caso tem como objetivo importar um arquivo com as definições de *import* para módulos que serão utilizados em *ExportProduct.* Por exemplo, se o desenvolvedor preferir definir a função que será chamada para gerar a saída da transformação em um módulo separado, ele escreve a importação de tal módulo no arquivo Haskell.

2. Por outro lado, o desenvolvedor pode preferir definir suas funções dentro do próprio módulo *ExportProduct.* Para isso ele pode utilizar a seção opcional *HaskellCode* que permite escrever qualquer código Haskell dentro dos delimitadores de *Code*. Isso, no entanto, não o impede de importar outras definições, que serão utilizadas em sua função, através do bloco *Imports*.

3. Seção obrigatória *ExportCode,* nela o desenvedor escreve o código que será inserido na função *exportProduct.* Geralmente consiste em chamadas a funções previamente definidas ou importadas para o módulo.

Na Listagem 3.19 temos as derivações na SDF de cada seção que foi explicada anteriormente.

```
"Output" "{" OutputBody "}"-> Output {cons("Output")}

Imports HaskellCode? ExportCode -> OutputBody {cons("OutputBody")}
Import* -> Imports {cons("ExtraImports")}

"Code" "{" Code "}" -> HaskellCode {cons("HaskellCode")}

"ExportCode" "{" Code "}" -> ExportCode {cons("ExportCode")}
```

Listagem 3.19 Seção Output

Uma vez definido o comportamento de cada componente na Listagem 3.20 temos um exemplo com todas as seções. Trata-se de um fragmento da extração de feature *PreprocessFiles*. Na seção *ExportCode* temos a chamada à função *preprocessFiles* definida no bloco *Code*.

```
Output{

        Import ("ImportModule.hs")

        Code{
                        preprocessFiles :: String -> [String] -> String -> IO()
                        preprocessFiles _ [] _ = return()
                        preprocessFiles flags files outputFolder = do
                           pid <- runCommand ("java -jar preprocessor.jar \"" ++ flags
++ "\"" ++ (concatPaths files outputFolder))
                           waitForProcess pid >>= exitWith

                        …
        }

        ExportCode{
                 preprocessFiles (outputPath ++ "/build.lst") (preProcessFiles
splInstance) outputPath
         }

    }
```

Listagem 3.20 Fragmento da seção Output de PreprocessFiles

3.5.3 Seção Input

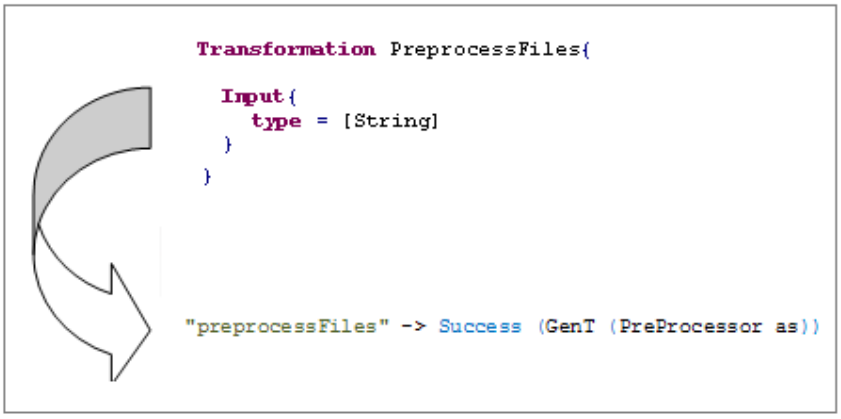
A seção obrigatória Input tem por objetivo informar o tipo de entrada para a transformação. Como observamos no tópico 3.2.2, os argumentos para a função de transformação são passados a partir do XML do ConfigurationKnowledge.

Na Listagem 3.21 temos a especificação desse bloco na SDF. O corpo do *Input* compreende a seção de *Imports*, já explicada anteriormente, e o *TypeDeclaration.* Neste o desenvolvedor define o tipo de entrada, que na maioria das transformações trata-se de uma lista de Strings.

```
"Input" "{" InputBody "}" -> Input {cons("Input"),prefer}
Imports TypeDeclaration -> InputBody {cons("InputBody")}
"Type" "=" TypeInput -> TypeDeclaration {cons("TypeDeclaration")}
```

Listagem 3.21 Seção Input da SDF

Na Listagem 3.22 observamos um fragmento de codificação em *Transkell* da transformação *PreProcessFiles* e o código Haskell gerado. Observamos que é feita uma chamada à função de transformação com o mesmo nome fornecido no nó *TransformationName*.



Listagem 3.22 Exemplo de transformação de código de Input

3.5.4 Seção SPLInstance

A seção SPL é responsável por criar uma nova entrada para o modelo de instância como vimos na Listagem 3.5 e inicializá-la como o fizemos na Listagem 3.6. Sua definição consiste na palavra reservada *SPL* seguida de alguns blocos delimitados por chaves. O *SPLBody* possui *Imports* e *HaskellCode* já mencionados em tópicos anteriores. Além disso, existe o bloco *SPLType* no qual identificamos o nome e tipo de entrada a ser adicionada no *SPLInstance*. Por fim temos o *SPLInitialize*, no qual podemos escrever o nome de uma variável, cuja inicialização foi feita previamente no bloco *HaskellCode*, ou um tipo básico.

Na Listagem 3.23 temos um fragmento da SDF que mostra a definição de *SPL,* bem como os componentes de *SPLBody*.

```
"SPL" "{" SPLBody "}" -> SPLInstance {cons("SPLInstance")}

Imports HaskellCode? SPLType SPLInitialize -> SPLBody {cons("SPLBody")}

"Type" "{" SPLIdentifier "=" IdentifierType "}" -> SPLType {cons("SPLType")}

Identifier -> SPLIdentifier {cons("SPLIdentifier")}

String -> IdentifierType {cons("IdentifierType")}

"Initialize" "{" InitializationBody "}" -> SPLInitialize {cons("SPLInitialize")}

Variable -> InitializationBody {cons("InitializationBody"),prefer}
```

Listagem 3.23 Fragmento da SDF de SPLInstance

Na Listagem 3.24 temos um exemplo da transcrição da seção SP para a transformação *PreProcessFiles.* Nela criamos uma entrada do tipo lista de *String*, que foi inicializada como uma lista vazia.

```
SPL{
  Type {
    preprocessFiles = [String]
  }

  Initialize {
        []
  }
}
```

Listagem 3.24 Fragmento SPL de PreprocessFiles

3.5.5 Seção Gui

Na seção *Gui* nós definimos os componentes necessário para compor a interface gráfica relativa à transformação criada. Na Listagem 3.25 temos os seguintes componentes no escopo da *Gui*:

1. *Imports* – Utilizado para indicarmos o arquivo que contém os *imports* de módulos que contém definições úteis a serem utilizadas na *Gui*.
2. *ComponentType* – Nele definimos o tipo de componente gráfico que será inserido. Nesta versão da SDF temos apenas os tipos que nos permitem inserir uma caixa de texto para o usuário entrar com algum valor e um selecionador de arquivos - *textbox* e *filechooser*, respectivamente.
3. *InputName* – Indica o texto que aparecerá ao lado do componente na interface gráfica.

4. *ValidationFunction* – Utilizado para informar o nome da função de validação para a entrada fornecida à transformação
5. *HaskellCode* – Seção utilizada para escrevermos qualquer código Haskell, pode ser útil para escrever funções de *parser* de entradas, caso o desenvolvedor não deseje importá-las de um módulo separado.

```
     "Gui" "{" Imports ComponentType InputName ValidateFunction  HaskellCode? "}"-> Gui
{cons("Gui")}

     "Type" "=" Component -> ComponentType  {cons("ComponentType")}

     "InputName" "=" "\"" String "\""-> InputName {cons("InputName")}

     "ParserFunction" "=" "\"" Identifier "\"" -> ValidateFunction {cons("ValidateFunction")}
```

Listagem 3.25 Seção Gui da SDF

3.5.6 Seção Imports

A seção de imports tem por objetivo permitir ao desenvolvedor importar arquivos Haskell, com extensão *.hs* ou *.lhs*. Tais arquivos devem conter definições completas de módulos que serão compilados juntamente com os módulos do *core* do Hephaestus. A Listagem 3.26 traz a definição dessa seção.

```
Import* -> Imports {cons("ExtraImports")}
 "Import" "(" FileExp ")"  ->  Import {cons("Import"),prefer}
 "\"" File "\"" -> FileExp  {cons("FileExp")}
 StringPath ExtensionCons -> File {cons("FilePath")}
 Extension -> ExtensionCons {cons("Extension")}
```

Listagem 3.26 Seção Import da SDF

3.5.7 Seção HaskellCode

Por fim temos a seção HaskellCode. Trata-se de uma seção opcional, útil quando o desenvolvedor deseja escreve a definição completa de um módulo no mesmo arquivo da transformação, ao invés de usar a sessão Imports para definir em arquivos separados. Todo código Haskell escrito nesse bloco é transformado em um módulo em um arquivo separado após a transformação e compilado juntamente com os outros módulos do Hephaestus. Na Listagem 3.27 temos sua definição na SDF.

```
"Code" "{" Code "}" -> HaskellCode {cons("HaskellCode")}
 HaskellWord  -> Code
```

Listagem 3.27 Seção HaskellCode da SDF

4. IMPLEMENTAÇÃO DAS TRANSFORMAÇOES DA LINGUAGEM

Neste capítulo apresentamos a solução implementada para geração de código em Haskell a partir da codificação em *Transkell*. Utilizamos as transformações do Stratego para interceptar cada nó da árvore gerada e transformar seu conteúdo. Existem duas formas de fazer as transformações de código. Podemos utilizar a sintaxe concreta ou sintaxe abstrata.

Na sintaxe abstrata nós interceptamos cada nó da árvore de acordo com o nome que foi declarado na com a expressão cons na SDF. Como exemplo, tomamos a seguinte derivação da definição sintática:

```
"Transformation" TransfName "{" Body "}" -> Transf {cons("Transf")}
```

Na Listagem 4.1 observamos um exemplo de regra de transformação do Stratego, para a definição acima.

```
to-Haskell :
    Transf(tName, tBody) ->  $[
                               [transformBody]
                             ]

  with transformBody := <transform-Body(|tName)>tBody
```

Listagem 4.1 Exemplo de transformação utilizando sintaxe abstrata

Assim, temos `to-Haskell` como o nome da regra que será aplicada e o construtor `Transf` que recebe dois parâmetros `tName` e `tBody`. Estes correspondem respectivamente ao `TransfName` e `Body` definidos na SDF.

O que aparece do lado direito da transformação, após o símbolo `->`, indica a reescrita do código. Neste caso, tudo que aparecer entre os delimitadores `$[` e `]` será interpretado como string. No entanto, se escrevermos um termo entre colchetes, este é interpretado como uma variável, que precisa ser definida posteriormente. No exemplo acima temos a variável `transformBody` cuja expressão `with :=` a define logo em seguida. Tal definição consiste em aplicar uma nova regra ao `tBody` passando o nome da transformação (`tname`) como parâmetro.

Por sua vez, na sintaxe concreta escrevemos o lado esquerdo da transformação da mesma maneira que o usuário escreve o código em Transkell. Na Listagem 4.2 temos a mesma transformação que analisamos, reescrita com sintaxe concreta.

```
to-Haskell:
                |[  Transformation tName {
                      tBody
                     }
                ]| -> $[
                          [transformBody]
                 ]
                with transformBody:= <transform-Body(|tName)>tBody
```

Listagem 4.2 Exemplo de transformação utilixando sintaxe concreta.

Percebemos que a sintaxe concreta é mais simples de entender. Ela é escrita entres os símbolos `|[` e `]|`, chamados de *quotations*. Contudo seu uso requer certo trabalho adicional. É necessário criar um novo módulo onde definimos os termos que aparecerão entre *quotations* e as *meta-variáveis.* Estas são variáveis de Stratego que representam identificadores, listas de argumentos de função e expressões. Na Listagem 4.3 temos os termos e variáveis que foram utilizados para implementação desta transformação. Percebemos que a *string* `tName` seguida ou não de uma sequência numérica, representa o símbolo não terminal `TransfName`. Este está definido no módulo Transkell que traz a implementação da sintaxe livre de contexto. Caso semelhante ocorre com a *string* `tBody` representando a expressão `Body`.

```
module Stratego-Transkell
imports
  StrategoMix[StrategoHost]
  Transkell

exports
  context-free start-symbols Module[[StrategoHost]]

  context-free syntax
    "|[" Transf "]|" -> Term [[StrategoHost]] {cons("ToMetaExpr")}

variables
    "tName"[0-9]*  -> TransfName     {prefer}
    "tBody"[0-9]*     -> Body         {prefer}
```

Listagem 4.3 Declaração de termos e meta-variáveis

Nos próximos tópicos analisamos as estratégias empregadas para gerar código para cada componente do corpo de uma transformação. Nelas utilizamos tanto sintaxe concreta como abstrata.

4.1 Input

Existem dois padrões de códigos que podem ser gerados a partir da seção Input. O código Haskell varia de acordo com o tipo que foi declarado. No caso de definirmos o tipo de lista de string (`[String]`), geramos o código, para o case do parser do *configurationKnowledge*, no seguinte padrão:

```
"transformationName" -> Success (GenT(TransformationName  as))
```

Por sua vez, quando temos o tipo de entrada `[String,String]`, é feita uma validação no formato da entrada antes da chamada da função de transformação.

```
"bindParameter" -> case as of
        [x,y] -> Success (GenT (BindParameter x y))
        otherwise -> Fail "Invalid number of arguments to the
        bind parameter transformation"
```

Utilizamos a sintaxe abstrata para definir esta regra, que recebe o nome da transformação como parâmetro. Na Listagem 4.4 observamos o código dessa transformação. Definimos `tnameModified` como a chamada a uma transformação que torna minúscula a primeira letra de uma *string* passada como parâmetro. Enquanto que `tname'` corresponde a chamada a uma regra que transforma o nó TransformationName em uma *string* que será o nome da função de transformação.

```
transformInputDeclaration(|tname):
        TypeDeclaration(TypeInputSingle(t)) -> $[
                "[tnameModified]" -> Success (GenT ([tname'] as))
        ]
        ...
transformInputDeclaration(|tname):
        TypeDeclaration(TypeInputPair(t)) -> $[
        "[tnameModified]" -> case as of
                [x,y] -> Success (GenT ([tname'] x y))
                otherwise -> Fail "Invalid number of arguments to the bind parameter transformation"
        ]
        ...
```

Listagem 4.4 Trasnformação da seção Input

4.2 Imports

A seção de `Imports`, como já discutimos, tem o objetivo de importar código de um arquivo externo. Como `Imports` é composto por uma lista de `Import`, utilizamos a estratégia `map`, que aplica uma função a cada elemento de uma lista. Na Listagem 4.5

observamos a aplicação da função `map` e a transformação de cada Import. Esta consiste na aplicação da transformação `<read-text-file>`, predefinida na biblioteca Stratego, que recebe o caminho de um arquivo e carrega o texto contido no mesmo.

```
to-haskell:
       ExtraImports(imports) -> $[[imports']]
       with imports':= <map(trasformImports)>imports

trasformImports:
      Import(FileExp(path)) -> $[
                      [read]
                      ]
        with read :=  <read-text-file>(<transformFilePath>path)
```

Listagem 4.5 Transformação da seção Imports

4.3 Gui

Na seção referente à interface gráfica geramos vários fragmentos de código que são incorporados ao módulo *Main* do Hephaestus. Utilizamos a sintaxe concreta para realizar essa transformação. Na Listagem 4.6 temos fragmentos da regra para geração de código. Como podemos observar o nome de componente da *Gui* é obtido a partir de uma transformação no nome da função de *parser*.

```
to-java:
      |[ Gui {
       tImport*
       tCompType
       tInputName
       tValidateFunct
      }]| ->
         $[
                [imports]
                      ...
                [identifier]FChooser  :: FileChooserButton,
                [compName]  <- xmlGetWidget f castToFileChooserButton "[identifier]FChooser"
                [identifier]FChooser = [compName],
                 Core.Success [compName]
                     ...
                [compName] <- getInputModel (([identifier]FChooser  gui), [tValidateFunct'],
"[tInputName'] not selected.")
                    ...
          ]
          with identifier := <transformFuncName>tValidateFunct
          with compName := <lower-case>(<transformFuncName>tValidateFunct)
          with tValidateFunct' := <showParseFunction>tValidateFunct
          with tInputName':= <showInputName>tInputName
         with imports:= <map(trasformImports)>tImport*
```

Listagem 4.6 Trasnformação da seção Gui

4.4 BasicType

A seção *BasicType* é responsável pela geração de códigos que são agregados ao módulo *BasicTypes* do core do Hephaestus. A Figura 4.1 ilustra o código gerado a partir da definição em Transkell.

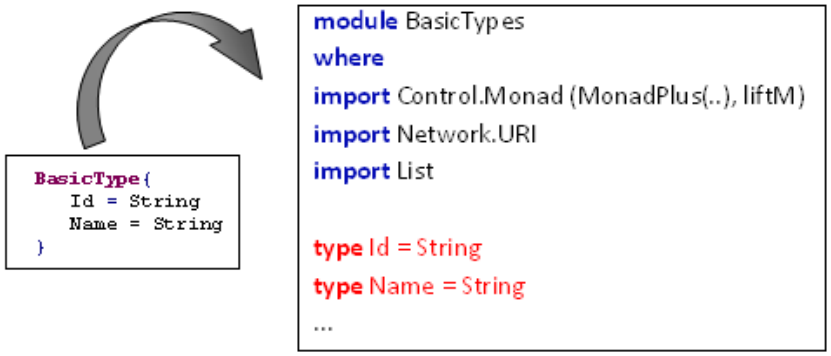


Figura 4.1 Exemplo de código gerado de BasicType

Como o corpo desta seção é composto por uma lista de `Declaration`, nós utilizamos o *map*, que aplica a transformação `transformDeclaration` a cada declaração. Esta é implementada com sintaxe abstrata, onde temos o construtor `Declaration`, que recebe dois parâmetros: `TypeName`, como no identificador do tipo a ser criado o `Type`.

```
to-java:
        BasicType(decl) ->
                $[ <module name="BasicType">
                        [decl']
                   </module>
                 ]
        with decl':= <transformDeclarations>decl

       transformDeclarations:
        Declarations(dec*) -> $[[dec']]
        with dec' := <map(transformDeclaration)> dec*

       transformDeclaration:
        Declaration(TypeName(identifier), Type(type)) -> $[
                                                type [identifier] = [type']
                                               ]
          with type':= <transformType>type


         transformType:
                PrimitiveType(t) -> $[[t]]
```

Listagem 4.7 Transformação de BasicType

4.5 SPL

A seção SPL gera código para dois módulos: *ConfigurationKnowledge.Interpreter e ConfigurationKnowledge.Types* . Para o módulo *ConfigurationKnowledge.Interpreter* ela adiciona a declaração de alguma variável, que pode está descrita no `HaskellCode` e faz a inicialização do *SPLInstance*. Na Listagem 4.8 podemos observar a estratégia de geração de código utilizada.

```
transfomIntiliaze:
               SPLInitialize(body) -> $[[body']]
                      with body' := <transformInitBody> body

               transformInitBody:
                        InitializationBody(Variable(var)) -> $[[var]]
```

Listagem 4.8 Trasnformação da inicialização do SPL

No módulo *ConfigurationKnowledge.Types*, por sua vez, ela adiciona uma entrada ao *SPLInstance*. Utilizamos a sintaxe abstrata para gerar esse trecho de código, que consiste em uma simples declaração em Haskell como podemos observar na Listagem 4.9.

```
transformSPLType:
                SPLType(SPLIdentifier(a), IdentifierType(b)) -> $[ [a] = [b]]
```

Listagem 4.9 Trasnformação da entrada para o SPL

As demais entradas do SPL, como *Imports* e *HaskellCode* sofreram transformações através das regras já apresentadas.

4.6 Output

Nesta seção temos o código Haskell que é gerado para o módulo *ExportProduct*. A transformação é bastante simples, utilizamos a mesma regra que transforma `ExportCode` para gerar o código que é inserido na função `exportProduct` e para escrever a funções auxiliares. A Figura 4.2 nos dá uma visão melhor da geração de código.

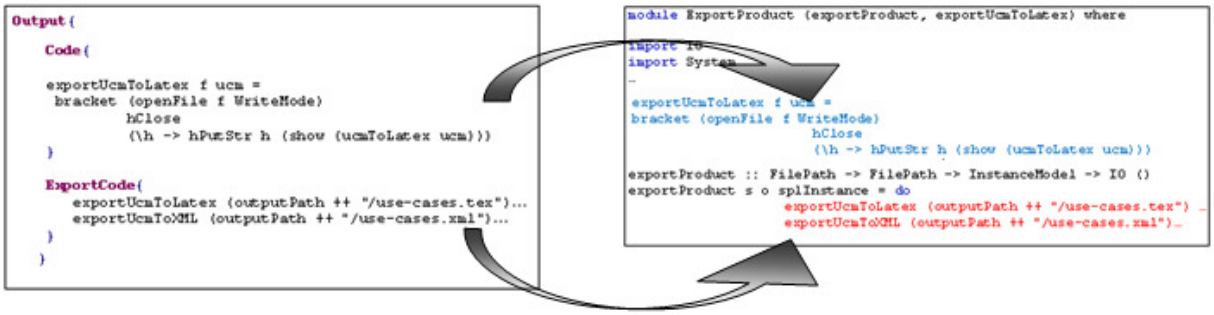


Figura 4.2 Geração de código do Output

5. VALIDAÇÃO E EVOLUÇÃO DA LINGUAGEM

Este capítulo tem por objetivo avaliar o poder de expressão da linguagem definida. Foram codificadas oito transformações existentes no Hephaestus. Durante o processo de codificação das transformações, a sintaxe da linguagem foi ajustada até chegar ao resultado final, apresentado no Apêndice H Uma transformação consiste em um grupo de funções que, uma vez fornecidos uma especificação da linha de produtos de software e um modelo de instância (que representa um produto da linha), aplica alguns tipos de transformações a tal instância. Assim o produto vai sendo refinado até chegar em sua versão final, que é retornada.

Nos próximos tópicos realizamos a extração de cada uma das *features* do Hephaestus, que correspondem às diferentes transformações que podem ser utilizadas. Tomamos por base o modelo de *features* apresentado no terceiro capítulo. Cada transformação está transcrita para linguagem de domínio específico Transkell, definida nos capítulos anteriores. Assim, poderemos observar as evoluções sofridas pela linguagem, a medida que codificamos as transformações, e sentimos a necessidade de expressar elementos até então não previstos.

5.1 Transformação PreProcessFiles

Inicialmente, foi extraída a transformação PreprocessFiles. Seu objetivo é indicar uma lista de arquivos que serão pré-processados, utilizando compilação condicional, e salvos no diretório de saída. O pré-processamento dos arquivos selecionados é realizado por um programa externo em Java (preprocessor.jar), o qual é invocado a partir do programa em Haskell como resultado da transformação. Observamos que não houve nenhuma necessidade de alteração na SDF, uma vez que foi possível expressar todas as informações necessárias para a geração de código.

Na listagem 5.1 podemos observar que para esta transformação foi necessário criar um tipo básico String para representar os paths dos arquivos que serão pré-processados. Além disso, verificamos que a entrada para a transformação é uma lista de Strings, que representa os paths descritos anteriormente.

```
Transformation PreprocessFiles{

 BasicType{
    Path = String
 }

 Input{
   type = [String]
 }
```

Listagem 5.1 Tipos básicos e Input da transformação PreProcessFiles

Na listagem 5.2 temos a seção de output da transformação. Em `Code`, existe a chamada ao executável em Java que fará o pré-processamento dos arquivos. Por sua vez na sessão `ExportCode` nó temos o código que será inserido no módulo de `ExportProduct`.

```
     Output{
           Code{
                          preprocessFiles :: String -> [String] -> String -> IO()
                          preprocessFiles _ [] _ = return()
                          preprocessFiles flags files outputFolder = do
                             pid <- runCommand ("java -jar preprocessor.jar \"" ++ flags ++
"\"" ++ (concatPaths files outputFolder))
                             waitForProcess pid >>= exitWith


concatPaths :: [String] -> String -> String
                          concatPaths [] _ = ""
                             concatPaths (str:strs) outputFolder = " \"" ++ outputFolder </>
str ++ "\"" ++ (concatPaths strs outputFolder)


          }

           ExportCode{
                    preprocessFiles (outputPath ++ "/build.lst") (preProcessFiles splInstance)
outputPath
            }

     }
```

Listagem 5.2 Região output de PreprocessFiles

Por fim, a Listagem 5.3 ilustra o bloco SPL, onde acrescentamos à instância da linha de produtos uma lista de Strings, representando Paths. E, por sua vez, em `Code` temos o código relacionado a instancia da transformação. A extração completa, com todos os trechos reunidos pode ser encontrada no Apêndice A.

```
SPL{
  Type {
   preprocessFiles = [String]
  }

  Initialize {
        []
  }
}

Code{
        data PreProcessor = PreProcessor
        {
            paths :: [Path]
        }

        instance Transformation PreProcessor where
        (<+>) (PreProcessor e) spl product =
                    let es = preProcessFiles product
                    in product { preProcessFiles = es ++ e }
}
```

Listagem 5.3 Regiões SPL e Code de PreprocessFiles

5.2 Transformação SelectUseCases

Esta *feature* visa selecionar casos de uso de uma linha de produtos de software a partir de uma lista de IDs, fornecida como entrada da transformação. Durante o processo de extração foram necessárias algumas modificações na SDF:

- Na seção *Gui* a produção “Import” foi inserida no início para dar uma idéia melhor do código que será gerado. O objetivo do “Import” é permitir ao usuário escrever em um arquivo haskell todas as importações de módulos que ele vai necessitar utilizar no módulo da *Gui*. Assim, temos a seguinte definição:
  "Gui" "{" GuiBody "}"-> Gui {cons("Gui")}

  Import* ComponentType InputName ValidateFunction? HaskellCode? ->
  GuiBody {cons("GuiBody")}

  Dessa forma fica mais claro que será gerado um código, para ser agregado ao módulo *Gui* do Hephaestus, cujo início possui alguns imports especificados pelo desenvolvedor.

- De forma semelhante, foi necessário acrescentar a seção de Imports na parte de Input, pois sentimos a necessidade de importar os módulos onde as funções, chamadas pela transformação, estavam definidas. Então ficamos com a seguinte definição na SDF:

```
"Input" "{" InputBody "}" -> Input {cons("Input"),prefer}
 Import* TypeDeclaration -> InputBody {cons("InputBody")}
“Type” “=” Type  = TypeDeclaration { cons(“TypeDeclaration”)}
```

Na listagem 5.4 podemos observar a seção de input com um import para os módulos que serão utilizados. O tipo de entrada é uma lista de string, representando os identificadores do casos de uso que serão selecionados.

```
Transformation SelectUseCases{

       ...

       Input{

          Import("ImportsToImput.hs")
           Type = [String]
       }

       ...
```

Listagem 5.4 Seção input para transformação SelectUseCases

Esta transformação, diferentemente da anterior, necessita de modificações na interface gráfica para permitir a indicação de um arquivo com os casos de uso para o produto. Na listagem 5.5 temos que o tipo de componente de *GUI* a ser criado será um *fileChooser* que permitirá ao usuário escolher o arquivo de entrada. Além disso, definimos um arquivo que possui os *imports* de módulos (ImportsToGui.hs), entre eles o módulo que define a função de parser do arquivo de entrada (parseUseCaseFile).

```
        ...

    Gui{
            Import ("ImportsToGui.hs")
            Type = fileChooser
            InputName = "Use case model"
            ParserFunction =  "parseUseCaseFile"
    }

     ...

 }
```

Listagem 5.5 Seção Gui para transformação SelectUseCases

No apêndice B encontramos a extração completa da transformação SelectUseCases, ao final dela encontramos uma sequência de *Imports*, onde cada um deles indica um novo módulo que será integrado à base do Hephaestus.

5.3 Transformação SelectScenarios

Essa transformação tem por objetivo selecionar os cenários de casos de uso da linha de produto de software a partir de uma lista de identificadores. Estes são fornecidos como uma sequência de strings separadas por vírgula, como input da transformação.

Ao realizar a extração dessa *feature*, percebemos a necessidade de tornar a sessão que corresponde ao estado da transformação, o SPL, também opcional. Isso ocorre, pois uma transformação pode se utilizar de mesmos elementos de outra pré-definida. Por exemplo, a transformação *SelectScenarios* utiliza a estrutura *UseCasseModel*, já definida na transformação *SelectUseCases*. Tal estrutura é formado por uma lista de componentes chamadas *UseCase*. Estes, por sua vez, concentram uma lista de *Scenarios* utilizados nessa transformação. Vejamos a Listagem 5.6 para entender melhor a organização do código:

```
data UseCaseModel = UCM {
 ucmName :: Name,
 useCases :: [UseCase],
 aspects :: [AspectualUseCase]
} deriving (Show, Typeable, Data)

data UseCase = UseCase {
 ucId :: Id,
 ucName :: Name,
 ucDescription :: Description,
 ucScenarios :: [Scenario]
} deriving (Show, Typeable, Data)

data Scenario = Scenario {
 scId :: Id,
 scDescription :: Description,
 from :: FromStep,
 steps :: Flow,
 to :: ToStep
} deriving (Show, Typeable, Data)
```

Listagem 5.6 Exemplo do código referente ao UseCaseModel

Dessa forma após a alteração na SDF temos a seguinte derivação para o corpo da transformação:

BasicType? Input Output? SPLInstance? Gui? Import* HaskellCode? -> Body

Percebe-se que o único bloco que permaneceu obrigatório foi o *Input*, referente à entrada que deve ser fornecida para transformação. Entretanto a modificação da seção

SPLInstance tornando-a opcional trouxe efeitos colaterais e algumas definições anteriores tornaram-se ambíguas. O problema ocorre devido a derivação *ExportCode*, referente ao código relacionado com a saída da transformação, que compõe a seção Output. Na seção *ExportCode* é permitido ao usuário escrever qualquer código Haskell delimitado por chaves conforme a definição abaixo:

"ExportCode" "{" Code "}" -> ExportCode {cons("ExportCode")}

HaskellWord -> Code

Temos que uma HaskellWord é uma sequência de quaisquer caracteres, inclusive "}", usada para delimitar o código Haskell. A falta de uma definição da linguagem Haskell incorporada à DSL levou a tal definição de código Haskell de forma tão abrangente. Então antes, quando havia algo escrito dentro do ExportCode o *parser* conseguia identificar seu final, pois logo após havia a definição obrigatória do SPLInstance, como podemos observar na listagem exemplo abaixo.

```
...
Output{

        ExportCode{

          print "\n Copying source files to output directory \n"
          print $ map (++ "\n") [fst c | c <- components p]

        }
   }
        SPL{

                ...

        }

...
```

Listagem 5.7 Exemplo de código referente a ExportCode

Entretanto, ao tornar tal seção opcional não há como identificar se o que vem após o *ExportCode* é uma nova seção ou simplesmente faz parte da definição do código, uma vez que se trata de uma sequência de caracteres. Para o problema ser resolvido, precisamos alterar a ordem dos elementos que compõem o corpo da Transformação e mover o *Output* imediatamente para antes do *Input*, que foi a única parte obrigatória restante. Assim, ficamos com a seguinte definição:

BasicType? Output? Input SPLInstance? Gui? Import* HaskellCode? -> Body

Dessa forma resolvemos o problema de ambiguidade. Para as transformações extraídas anteriormente serem reconhecidas pelo *parser* da linguagem, basta que movamos a declaração *SPL* para antes do "Input". Na Listagem 5.4 podemos observar a codificação de *SelectScenarios* após as alterações de sintaxe realizadas.

```
Transformation SelectScenarios{

        Input {
          Type = [String]
        }


        Code{
                module Transforamation.SelectScenarios
                where
                import Transformations.UseCaseModel
                -- | Transformation that selects SPL scenarios
                --   from a list of IDs. This transformation deals
                --   with the kind of transformation named 'variability
                --   in function'.

                data SelectScenarios = SelectScenarios  {
                 scIds :: [Id]
                }

                instance Transformation SelectScenarios where
                 (<+>) (SelectScenarios ids) spl product =
                     addScenariosToInstance (scs, spl, product)
                     where scs = [s | s <- splScenarios spl, scId s `elem` ids]

                instance Show SelectScenarios where
                 show (SelectScenarios ids) = "selectScenarios " ++ (show ids)
        }
}
```

Listagem 5.8 Extração da feature SelectComponents

5.4 Transformação SelectAndMoveComponents

Esta transformação é responsável por mover um componente - sendo este uma classe, aspecto, arquivo de configuração entre outros - de um local para outro para ser utilizado no produto final. Para tal ela se utiliza de uma lista com o mapeamento entre identificadores e os nome dos componentes. Ao realizar sua extração, foram verificadas algumas necessidades de alteração no arquivo de definição da sintaxe, como observaremos a seguir.

Na versão anterior da SDF tínhamos as seguintes derivações para a seção referente ao input da transformação:

```
"Input" "{" InputBody "}" -> Input {cons("Input"),prefer}
Import* TypeDeclaration -> InputBody
```

"Type" "=" Type -> TypeDeclaration

Onde “Type” estava definido em outro módulo, responsável pelos componentes léxicos da linguagem, da seguinte forma:

"String" -> Type
"Bool" -> Type
"Char" -> Type
"Int"-> Type
"[" Type "]" -> Type

Entretanto, houve a necessidade de escrever o seguinte código:

Input{

Type = [String,String]

}

Tal produção, não era possível com a definição existente, portanto a SDF foi modificada para dá suporte a listas, e também foi inserido o suporte a tuplas, como podemos observar na Listagem 5.9.

```
PrimitiveType -> Type {cons("Type")}
TypeList -> Type {cons("Type")}
TypeTuple -> Type {cons("Type")}
"[" TypeStringSequence "]" -> TypeList {cons("TypeListString")}
"[" TypeCharSequence "]" -> TypeList {cons("TypeListChar")}

…

  "(" TypeStringSequence ")" -> TypeTuple {cons("TypeTupleString")}
  "(" TypeCharSequence ")" -> TypeTuple {cons("TypeTupleChar")}



…
```

Listagem 5.9 Definição de tipos na nova versão da DSL

Assim, podemos observar na Listagem 5.10 parte da codificação de SelectAndMoveComponents reconhecida sintaticamente após as alterações na linguagem. Nela temos o tipo de entrada para a transformação que representa uma lista que mapeia Ids em nomes de componentes. E no Apêndice C temos todo o código em Transkell, referente à extração dessa feature.

```
Transformation SelectAndMoveComponent{

      Input{

       Import("TransforSelectAndMoveComponents.hs")
       Type = [String,String]

      }
...
```

Listagem 5.10 Seção Input da transformação SelectAndMoceComponents

5.5 Transformação BindParameter

Essa funcionalidade tem por objetivo fazer a ligação entre um parâmetro de cenário e uma seleção de *features*. Ao realizarmos sua extração verificamos que ela possui um comportamento bastante similar a SelectScenarios, para tal transformação não houve a necessidade de criar tipos básicos ou definir o SPL instance, uma vez que ele utiliza componentes definidos no modelo da casos de uso. Os códigos específicos dessa transformação foram extraídos do modulo UseCaseModel para a seção “Code”, conforme podemos observar na listagem 5.11.

```
Transformation BindParameter {

...
Code{
            module Trasformations.BindParameter
            where
            import Transformations.UseCaseModel


            data BindParameter = BindParameter {
             pmtId :: Id,       -- ^ formal parameter of scenarios
             featureId :: Id    -- ^ feature identifier
            }

            ...
      }

...

}
```

Listagem 5.11 Seção Code da transformação BindParameter

Assim temos um “data” criado que possui um identificador da *feature* e outro identificador para o cenário, ambos do tipo String. A extração completa da transformação encontra-se no Apêndice D.

5.6 Transformação EvaluateAspects

Essa transformação é responsável por avaliar uma lista de cenários de caso de uso aspectuais a partir de seus identificadores. Para extraí-la foi apenas necessário criar um modulo *Transformation.EvaluateAspects* o qual importa definições de tipo e funções de *UseCaseModel*. O código relacionado encontra-se na listagem 5.12. Nela, observamos que o input consite em uma lista de Strings que representa os Ids dos aspectos que deverão ser avaliados.

```
Transformation EvaluateAspects{

        Input{

                Type = [String]
        }

        Code{

                module Trasformations. EvaluateAspects
                where

                data EvaluateAspects = EvaluateAspects {
                 aspectIds :: [Id]  -- ^ Ids of the aspects that will be evaluated
                }

                instance Transformation EvaluateAspects where
                 (<+>) (EvaluateAspects ids) spl product = evaluateListOfAdvice as product
                  where
                   as = concat [advices a | a <- aspects (splUCM spl), (aspectId a) `elem` ids]


                instance Show EvaluateAspects where
                 show (EvaluateAspects ids) = "evaluateAspects " ++ (show ids)
        }
}
```

Listagem 5.12 Extração da trrasnformação EvaluateAspects

5.7 Transformação SelectComponents

SelectComponents tem o propósito de recuperar o nome de um componente a partir de um Id fornecido. Para isso ela utiliza a mesma estrutura do modelo de instância definida pela transformação *SelectAndMoveComponents*, ou seja uma lista de tuplas que servem para fazer tal mapeamento descrito. Sua definição na DSL proposta não levantou a

necessidade de alterações, para tal foi apenas necessário definirmos a entrada do tipo lista de String que trará o nome dos componentes que serão selecionados. Além disso foi utilizada a seção "*Code*" para criação do novo modulo que contém a definição do Data "*SelectComponent*", bem como a implementação da função de transformação associada. Na listagem 5.13 temos um fragmento do código, e a extração completa encontra-se no apêndice E.

```
Transformation SelectComponents{

      Input{
      Type = [String]
      }

      Code{
            module Transformations. SelectComponents
            where

            import BasicTypes
            import ConfigurationKnowledge.Types

       …
}

…
}
```

Listagem 5.13 Fragmento da transformação SelectComponents

5.8 Transformação CreateBuildEntries

Esta última transformação tem por objetivo criar novas entradas para o arquivo de build do novo produto que será gerado. Bem como, auxiliar a transformação *PreprocessFiles* gerando um arquivo com *tags* de pré-processamento. Sua entrada consiste em uma lista de strings que serão escritas no arquivo de saída da transformação.

Na listagem 5.14 temos um fragmento da extração que mostra o estado intermediário da *feature SelectBuildEntries*. Verificamos que foi criado um novo *Type* (*buildEntries*), que consiste em uma lista de strings, as quais guardarão o input da transformação. A codificação completa encontra-se no Apêndice F.

```
SPL{
     Type{
      buildEntries = [String]
     }

        Initialize{
          []
      }
}
```

Listagem 5.14 SPL da transformação SelectBuildEntries

6. RESULTADOS OBTIDOS APÓS EXTRAÇÃO DAS FEATURES

Neste capítulo apresentaremos os resultados obtidos após a extração das transformações do Hephaestus utilizando a DSL especificada.

Nos capítulos 3 nós analisamos o código original do Hephaestus e identificamos alguns problemas:

- Falta de modularização;
- Código de transformações diferente mesclados em um mesmo módulo (*crosscutting concerns*);
- Espalhamento de código de cada transformação ao longo de vários módulos (*scattering concern*s)

Esses problemas levantados dificultam a manutenção de código. Além disso, a falta de modularização diminui o reuso de componentes semelhantes, levando a duplicações de código.

6.1 Espalhamento do código do Hephaestus

Neste tópico mostraremos como o código relativo às transformações encontra-se espalhado no Hephaestus. Para isso selecionamos três transformações que nos servirão como exemplo. Para cada uma delas, temos gráficos de espalhamento que nos dão uma visão mais geral do problema.

6.1.1 PreprocessFiles

O código relativo a essa transformação encontra-se distribuído em seis módulos diferentes. Na Figura 6.1 observamos o código de *PreprocessFiles* destacado nos seguintes módulos:

1. *BasicTypes* – Neste módulo definimos a tipo *path* como uma String, que representa o caminho absoluto dos arquivos pré-processados.
2. *Transformations.ComponentModel* – Esse módulo é responsável por definir o tipo *Data* que encapsula uma lista de *paths.* Além disso, ele define a função de transformação.
3. *ConfigurationKnowledge.Interpreter* – Tem o propósito de inicializar a lista de *paths*.

module ConfigurationKnowledge.Interpreter

```
module ConfigurationKnowledge.Interpreter (build)
where

import UseCaseModel.Types
import RequirementModel.Types
import ConfigurationKnowledge.Types

build :: FeatureModel
      -> FeatureConfiguration
      -> ConfigurationKnowledge
      -> SPLModel
      -> InstanceModel
build fm fc ck spl = stepRefinement ts spl emptyInstance
 where
  ts           = tasks ck fc
  ucmodel      = splUCM spl
  emptyUCM     = ucmodel { useCases = [] , aspects = [] }
  emptyReq     = RM { reqs = [] }
  emptyInstance = InstanceModel fc emptyReq emptyUCM [] [] []
```

module Transformations.ComponentModel

```
module Transformations.ComponentModel
Where
 import BasicTypes
import ConfigurationKnowledge.Types


data PreProcessor = PreProcessor {
    paths :: [Path]
}


instance Transformation PreProcessor where
(<+>) (PreProcessor e) spl product =
      let es = preProcessFiles product
      in product { preProcessFiles = es ++ e }

instance Show PreProcessor where
show (PreProcessor e) = "preProcessor " ++ (show e)

data CreateBuildEntries = CreateBuildEntries {
  entries :: [String]
}
```

module BasicTypes

```
module BasicTypes
where

import qualified System.FilePath.Posix as P
import qualified System.FilePath.Windows as W
import Control.Monad (MonadPlus(..), liftM)

type Id = String
type Name  = String
type Path = String

data ParserResult a = Success a | Fail String
 deriving (Read, Show, Eq, Ord)


instance Monad ParserResult where
 return = Success fail = Fail
Success a >>= f = f a
Fail s    >>= f = Fail s
```

Figura 6.1 Espalhamento de PreprocessFiles

Na Figura 6.2 observamos os outros três módulos onde encontramos código dessa transformação.

1. *Transformations.Parsers.XML.XmlConfigurationKnowledge* – Faz o parser do XML do configurationKnowledge e chama a transformação.
2. *ConfigurationKnowledge.Types* – Neste modulo é criada uma entrada para o *InstanceModel.*
3. *ExportProduct* – Responsável pela chamada e definição da função que pré-processa os arquivos.

module Transformations.Parsers.XML.XmlConfigurationKnowledge

```
module Transformations.Parsers.XML.XmlConfigurationKnowledge
where

import BasicTypes
import ConfigurationKnowledge.Types

xml2Transformation :: XmlTransformation -> ParserResult GenT
xml2Transformation t =
 let as = splitAndTrim ',' (tArgs t)
 in
 case tName t of
"selectScenarios" -> Success (GenT (SelectScenarios as))

"selectUseCases" -> Success (GenT (SelectUseCases as))

"bindParameter" -> case as of
                                [x,y] -> Success (GenT (BindParameter x y))
                                otherwise -> Fail "Invalid number of
arguments to the bind parameter transformation"

"evaluateAspects" -> Success (GenT (EvaluateAspects as))
"selectComponents" -> Success (GenT (SelectComponents as))
"selectAndMoveComponent" -> case as of
                          [x,y] -> Success (GenT (SelectAndMoveComponent
x y))
                          otherwise -> Fail "Invalid number of arguments to the
select and move transformation"

"createBuildEntries" -> Success (GenT (CreateBuildEntries as))

"preprocessFiles" -> Success (GenT (PreProcessor as))
otherwise -> Fail ("Invalid transformation: " ++ tName t)
```

module ConfigurationKnowledge.Types

```
module ConfigurationKnowledge.Types
where

import Data.Generics

data SPLModel =  SPLModel {
      splFM  :: FeatureModel,
      splReq :: RequirementModel,
      splUCM :: UseCaseModel,
      splMappings :: ComponentModel
}


data InstanceModel =  InstanceModel {
      fc  :: FeatureConfiguration,
      req :: RequirementModel,
      ucm :: UseCaseModel,
      components :: [(Id, Id)],
      buildEntries :: [String],
      preProcessFiles :: [String]
} deriving (Data, Typeable)

data Priority = High | Medium | Low

data GenT = forall t . Transformation t => GenT t

type ConfigurationKnowledge = [Configuration]
```

module ExportProduct

```
module ExportProduct (exportProduct, exportUcmToLatex, exportAspectInterfacesToLatex)

Where
import IO
import System
import System.Directory

exportProduct :: FilePath -> FilePath ->  InstanceModel -> IO ()
exportProduct s o p = do  exportUcmToLatex (o ++ "/use-cases.tex") (ucm p)
  exportUcmToXML (o ++ "/use-cases.xml") (ucm p)


  print "\n  Copying source files to output directory \n"
  print $ map (++ "\n") [fst c | c <- components p]

  copySourceFiles s o (components p)
  exportBuildFile  (o ++ "/build.lst") (buildEntries p)
  preprocessFiles (o ++ "/build.lst") (preProcessFiles p) o

preprocessFiles :: String -> [String] -> String -> IO()
preprocessFiles _ [] _ = return()
preprocessFiles flags files outputFolder = do
   pid <- runCommand ("java -jar preprocessor.jar \"" ++ flags ++ "\"" ++ (concatPaths
files outputFolder))
   waitForProcess pid >>= exitWith


-- ------------------------------------------------------------------------
-- Exports a use case model as a latex document.
-- ------------------------------------------------------------------------
exportUcmToLatex f ucm =  bracket (openFile f WriteMode)
        hClose        (\h -> hPutStr h (show (ucmToLatex ucm)))
```

Figura 6.2 Espalhamento do código de PreprocessFiles

6.1.2 SelectUseCases

A transformação *SelectUseCases* possui código espalhado por doze módulos distintos, iremos exemplificar uma amostra deles. Na Figura 6.3 temos os módulos *BasicTypes*, *Transformations.Parsers.XML.XmlConfigurationKnowledge*, *Transformations.UseCaseModel* e *ConfigurationKnowledge.Interpreter.*

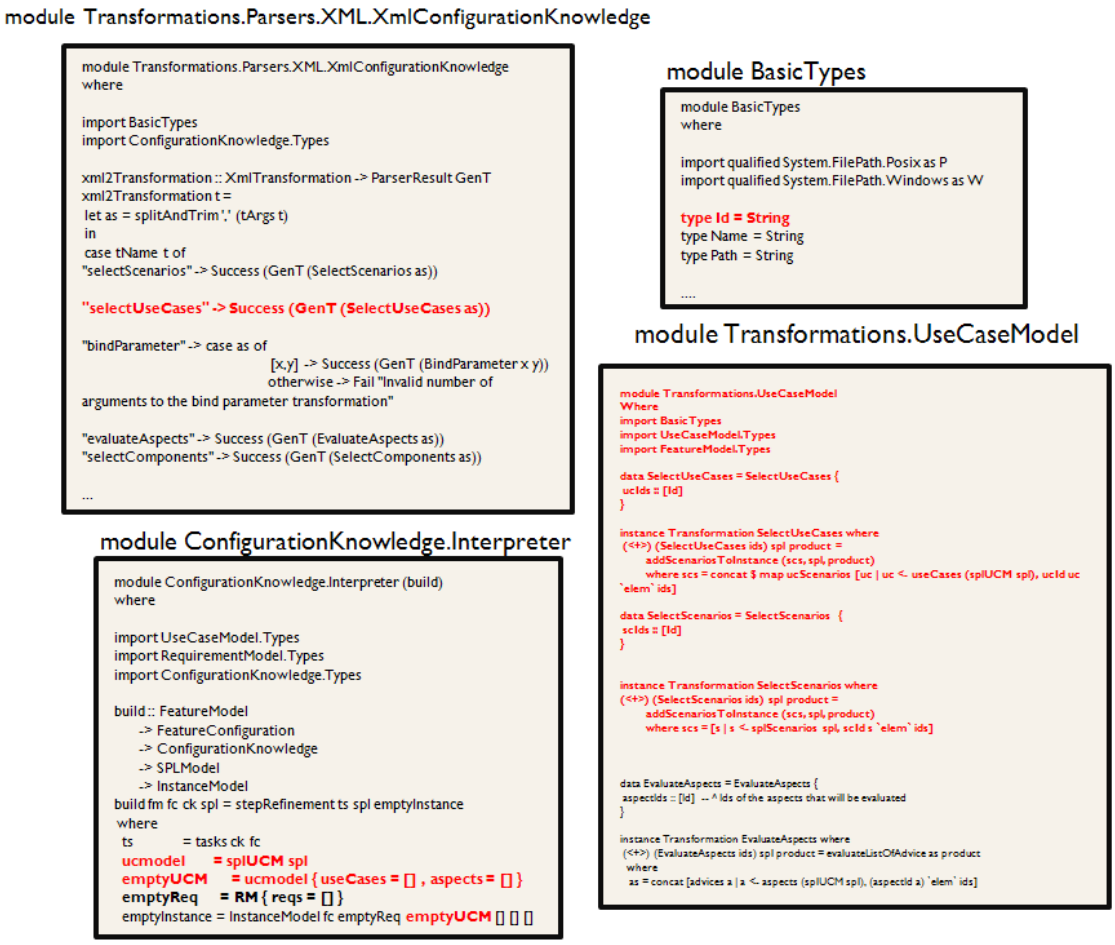


Figura 6.3 Espalhamento de SelectUseCases

Na Figura 6.4 temos código destacado em mais 4 módulos : *ExportProduct*, *ConfigurationKnowledge.Types*, *Main,* que traz a definição dos componentes de interface gráfica necessários nesta transformação e *UseCaseModel.Types*, com a definição de todos os tipos em Haskell exclusivos para *SelectUseCases*.

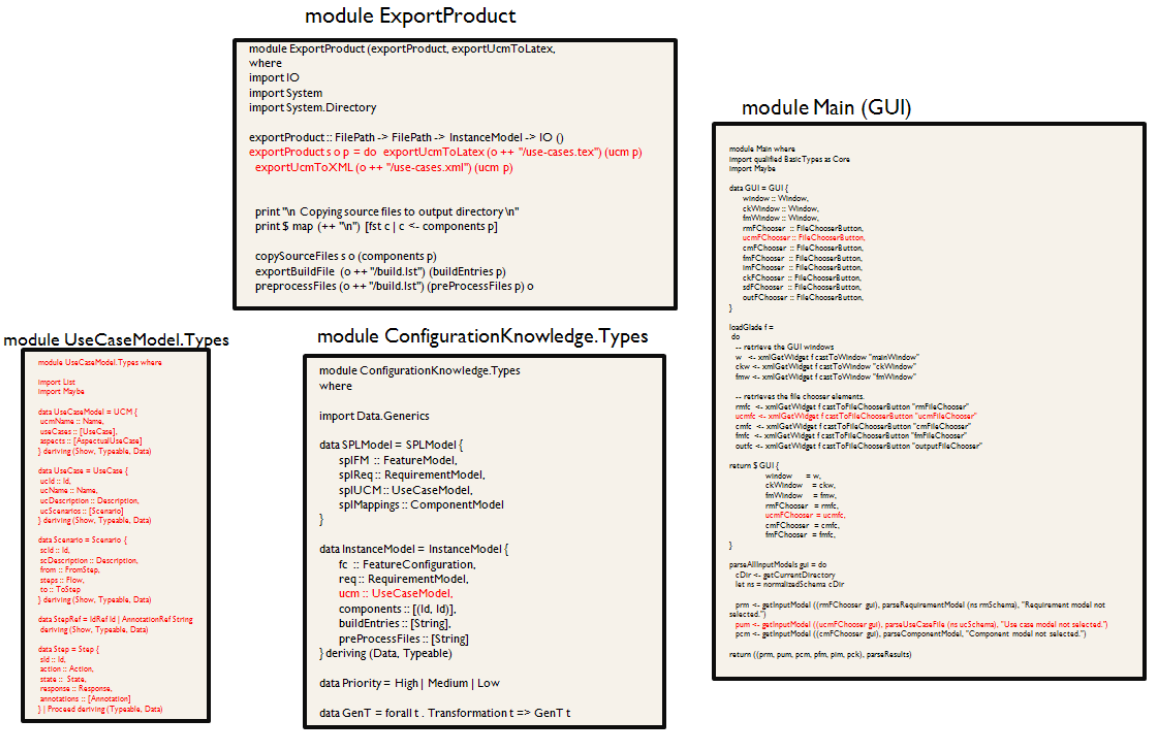


Figura 6.4 Outros módulos com espalhamento de SelectUseCases

6.1.3 SelectAndMoveComponent

Na transformação *SelectAndMoveComponent* encontramos códigos relacionados em treze diferentes módulos. Para efeitos de praticidade iremos apenas exemplificar alguns deles. Na Figura 6.5 observamos o *ConfigurationKnowledge.Types, ExportProduct* e *ConfigurationKnowledge.Interpreter*.

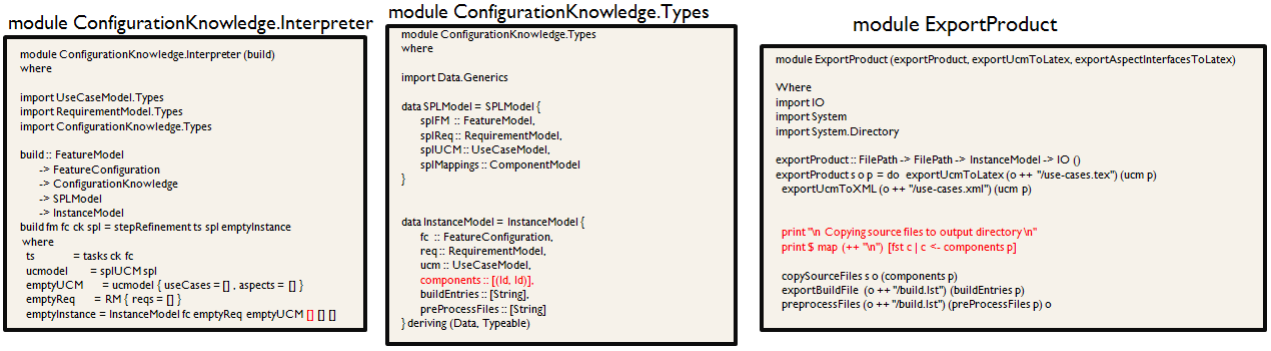


Figura 6.5 Espalhamento de SelectAndMoveComponent

Na Figura 6.6 temos o módulo *Main*, onde podemos perceber vários trechos de código de interface gráfica mesclados com código de outras transformações. Ainda temos o módulo *Transformations.Parsers.XML.XmlConfigurationKnowledge* e o módulo onde especificamos a função de transformação: *Transformations.ComponentModel.*

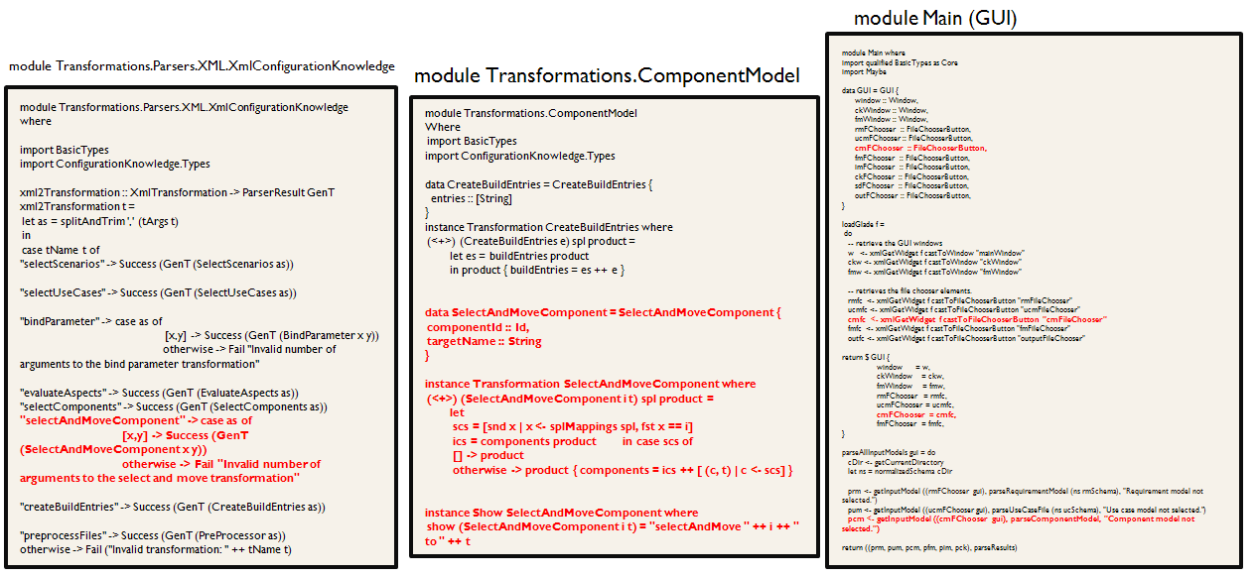


Figura 6.6 Exemplo de espalhamento de SelectAndMoveComponent

6.2 Crosscutting concerns

As transformações exemplificadas anteriormente além de estarem espalhadas também se cruzam em vários pontos. Na Figura 6.7 observamos os concerns de cada transformação, que se encontram entrelaçados basicamente nos seguintes módulos:

- *BasicTypes* – Onde definimos os tipos básicos que representam entradas das transformações.
- *ConfigurationKnowledge.Types* – Onde definimos uma entrada que representa o estado da transformação.
- *ExportProduct* – Neste módulo especificamos chamadas às funções responsáveis pela saída da transformação.
- *Main* – Utilizado para definir todos os componentes de interface gráfica relacionados à transformação.
- *ConfigurationKnowledge.Interpreter* – Módulo onde são inicializados componentes necessários para transformação.
- *Transformations.Parsers.XML.XmlConfigurationKnowledge*

Para facilitar o entendimento destacamos o código relacionado a cada concern com cores diferentes.

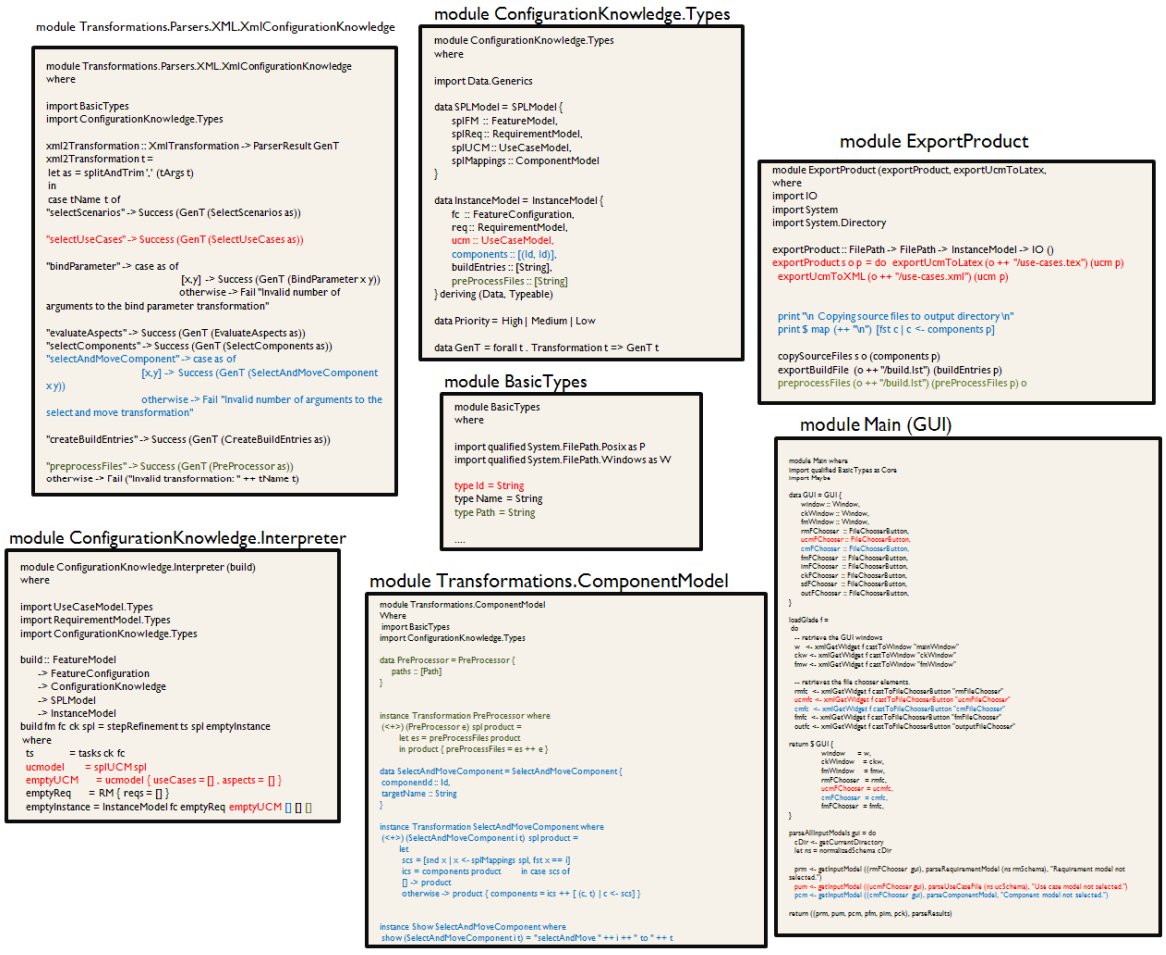


Figura 6.7 Exemplo de crosscutting concerns do Hephaestus.

6.3 Melhorias obtidas com a DSL

Ao extrairmos as transformações do Hephaestus utilizando a DSL Transkell obtivemos um código mais modularizado e de fácil manutenção. Isto ocorreu devido à reunião das definições relativas a cada transformação em arquivos Transkell separados. Dessa forma eliminamos o problema de espalhamento de código por vários módulos, que existia anteriormente. Na Figura 6.8 temos as transformações *PreprocessFiles*, *SelectUseCases* e *SelectAndMoveComponent*,escritas na linguagem Transkell, a fim de exemplificar tal melhoria.

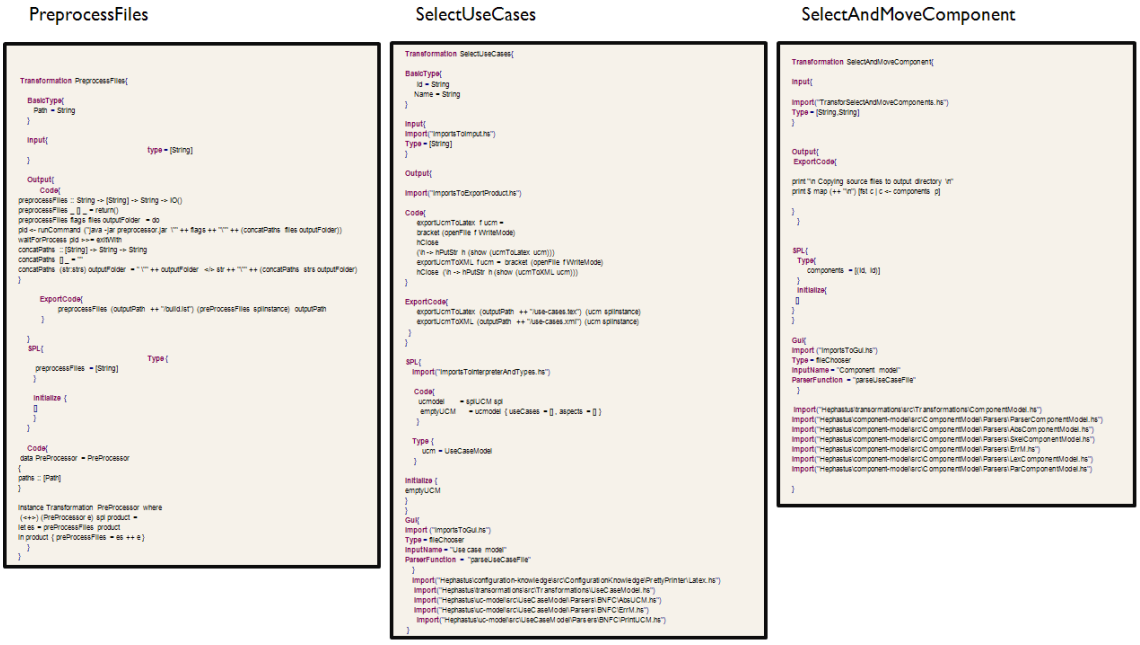


Figura 6.8 Exemplo de modularização do código com Transkell

Percebemos que o problema de entrelaçamento também é resolvido, cada *concern* encontra-se centralizado em seu respectivo módulo. A solução apresentada permite ao desenvolvedor algumas facilidades na escrita do código. Por exemplo, é possível criar componentes de interface gráfica com poucas linhas de códigos como podemos observar na Figura 6.10.

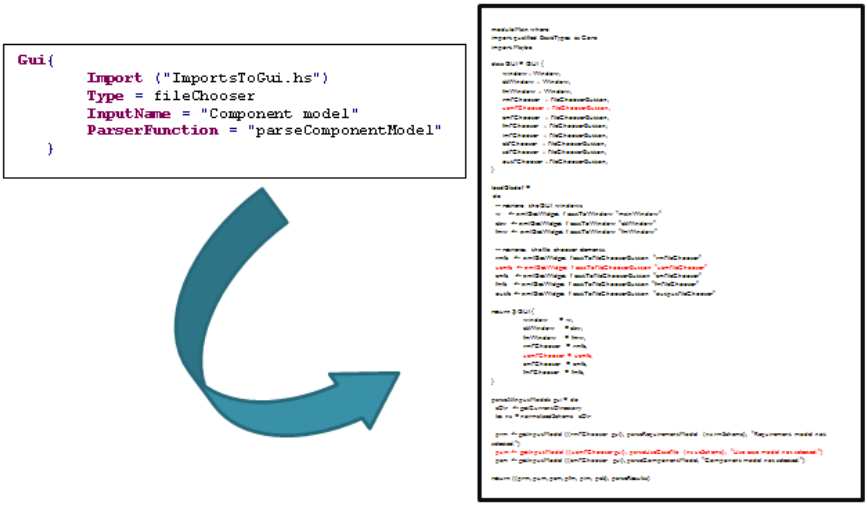


Figura 6.9 Comparativo entre código da GUI em Transkel e em Haskell

Também poupamos ao programador o trabalho de especificar o *parser* do *xml* do *configurationknowledge*. O código responsável por tais verificações bem como a chamada à função de transformação é gerado automaticamente.

Dessa forma, Transkell oferece ao programador facilidades no momento de criar uma nova transformação. A fim de simplificar a escrita na nova linguagem foi criado um *plugin* para o eclipse. Para isso, utilizamos o Spoofax [30], ferramenta do Stratego para desenvolvimento de linguagens de domínio específico que permite criar *plugins* customizados para o eclipse.

Uma vez instalado, o *plugin* oferece *syntax highlighting* e geração do código do novo produto com apenas um clique. Na Figura 6.9 temos uma imagem mostrando o editor de transformações no eclipse.

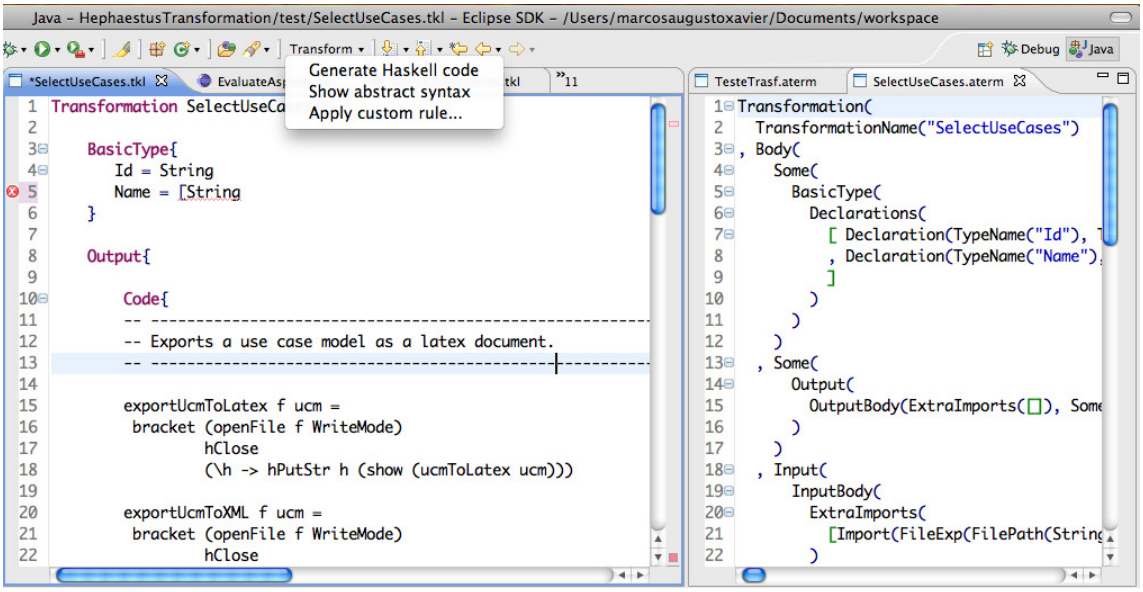


Figura 6.10 Plugin criado para escrever as Transformações no Eclipse.

Após compilarmos o código com as novas transformações temos o executável de um novo produto. Assim, a criação de diferentes produtos se tornou um trabalho relativamente simples que consiste em:

1. Especificar a nova transformação em um arquivo Transkell. Ou reunir várias transformações em um único arquivo com extensão .tkl.
2. Selecionar todo o código do arquivo e clicar no botão *Transform*, escolhendo a opção *Generate Haskell Code.*
3. Compilar o código gerado.

Na Figura 6.11 temos um exemplo de produto gerado como resultado final do processo de geração de uma linha de produtos de software. Para este produto foram selecionadas as seguintes *features*: selectComponents, selectUseCases e createBuildEntries.

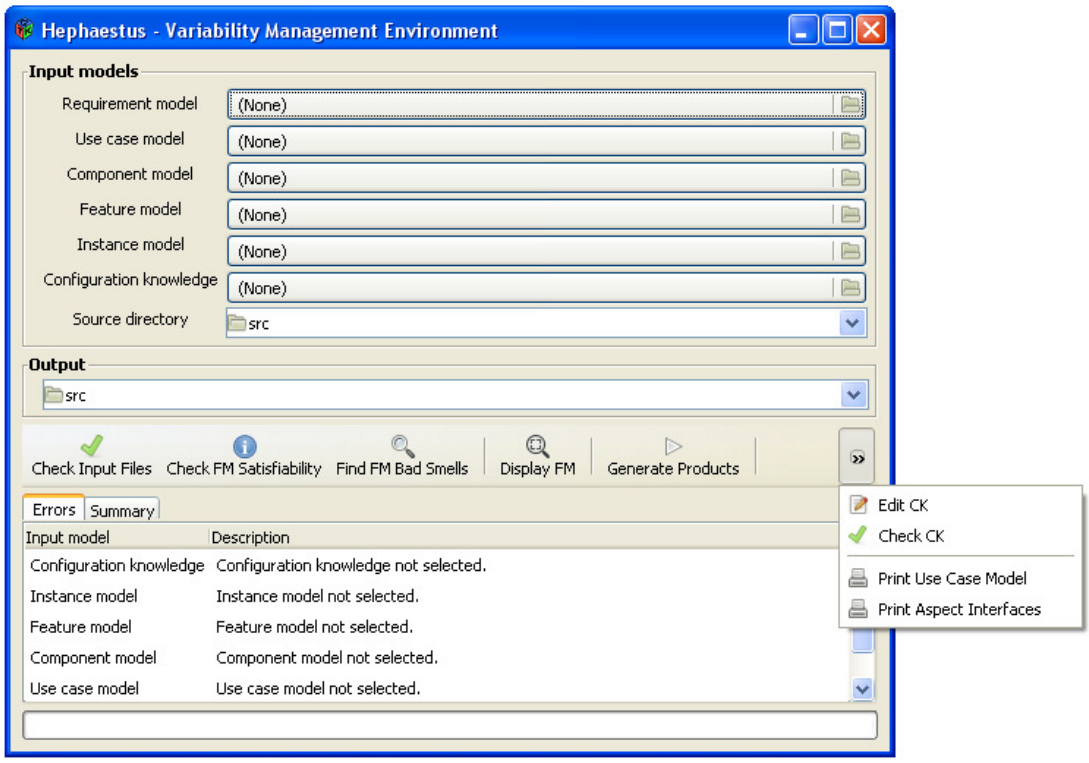


Figura 6.11 Instancia de um produto da SPL Hephaestus

7. Conclusões

Neste trabalho apresentamos algumas técnicas existentes em Haskell que nos permitem a metaprogramação para geração de variações em software. No entanto, elas não estavam apropriadas ao sistema do Hephaestus que analisamos.

Observamos assim, todo o processo de criação e implementação de uma linguagem de domínio específico como solução do problema. Extraímos também todas as *features* do Hephaestus, dessa forma o transformamos em uma linha de produtos, pois podemos gerar diferentes instâncias selecionando as *features* desejadas. Também verificamos que os problemas encontrados no código, como espalhamento e *croscutting* concerns foram resolvidos.

Ao final, geramos um *plugin* para o eclipse que permite ao desenvolvedor escrever novas transformações de uma forma mais simples, sem a necessidade de escrever muito código. Além de gerar o código de um novo produto de forma bastante prática.Por exemplo, tivemos uma redução de 55% de código escrito para definição de interface gráfica e até 85% no *parser* do XML do *ConfigurationKnowledge* para chamada da transformação apropriada.

A curva de aprendizado do Stratego/XT foi um pouco lenta, uma vez que a documentação é bastante extensa. O uso do plugin Spoofax para o eclipse, no entanto, facilitou bastante a criação da DSL e das transformações, devido ao seu editor e por abstrair o uso de várias ferramentas do Stratego/XT para geração de estruturas intermediárias.

7.1 Trabalhos Futuros

Como trabalho futuro, poderíamos incluir a SDF da linguagem Haskell dentro da SDF de Transkell. Dessa forma todo o código escrito nas seções `Code` seria validado sintaticamente. Outra proposta seria implementar a função de auto completar no editor de Transkell, bem como criar *templates* de seções, visando facilitar a criação de novas transformações.

Por fim poderíamos utilizar a solução apresentada para criar novas transformações para o Hephaestus. Por exemplo, o usuário poderia fornecer um documento de requisitos para a linha de produtos completa, a partir da interface gráfica, e realizar o mapeamento de

features e requisitos. Dessa forma, poderíamos gerar um documento de requisitos específico para cada produto da linha.

REFERÊNCIAS BIBLIOGRÁFICAS

[1]. ANASTASOPOULES, M.; GACEK, C. ***Implementing Product Line Variabilities***.

[2]. LÄMMEL, Ralf; VISSER, Joost. **A Strafunski Application Letter**.

[3]. **Applications and libraries/Generic programming/Strafunski**. Disponível em: <http://www.haskell.org/haskellwiki/Applications_and_libraries/Generic_programming/Strafunski> Último acesso em: 02/06/2010

[4]. GRABMÜLLER, Martin; KLEEBLATT, Dirk. **Harpy Tutorial**. Abril 2007

[5]. **Gramáticas livres de contexto**. Disponível em < http://www.dca.fee.unicamp.br/cursos/EA876/apostila/HTML/node43.html> . Último acesso em 16 de março 2010.

[6]. SWIERSTRA, Wouter. **Why Attribute Grammars Matter**. Julho 2005

[7]. Knuth E. Dolnald. **The genesis of attribute grammars. Proceedings of the international conference on Attribute grammars and their applications**. 1990, vol. 461, 1–12.

[8]. Type classes and overloading. Disponível em <http://www.haskell.org/tutorial/classes.html> Último acesso em 02 de junho de 2010.

[9]. SULZMANN, Martin; WANG, Meng. **Aspect-Oriented Programming with Type Classes**.

[10]. ANDRADE, Carlos A. R.; SANTOS, André L. M.; BORBA, Paulo H. M. **AspectH: Uma Extensão Orientada a Aspectos de Haskell.** Recife PE.

[11]. **The art of metaprogramming, Part 1: Introduction to metaprogramming**. Disponível em:< http://www.ibm.com/developerworks/linux/library/l-metaprog1.html >

[12]. JONES, Simon P.; SHEARD, T. **Template metaprogramming for Haskell**. In: CHAKRAVARTY, Manuel M. (Ed.). ACM SIGPLAN Haskell Workshop 02. EUA: out. 2002. P. 1-16.

[13]. **Template Haskell**. Disponível em: <http://www.haskell.org/haskellwiki/Template_Haskell>

[14]. **Low-level TH**. Disponível em <http://www.haskell.org/bz/th3.htm>. Último acesso em 03 de junho 2010.

[15]. **Software Product Lines.** Disponível em < http://www.sei.cmu.edu/productlines/ >. Último acesso em 10 de junho 2010.

[16]. **What are the benefits of Software Product Lines?** Disponível em < http://www.software-acumen.com/about-product-lines/product-line-benefits// > Último acesso em 10 de junho 2010.

[17]. **Ferramentas do SBCARS 2009**.Disponível em: < http://www.ines.org.br/?tag=linhas-de-produtos-de-software > Último acesso 10 de junho de 2010.

[18]. **Domain Specific Languages**. Disponível em: <http://www.martinfowler.com/bliki/DomainSpecificLanguage.html > Último acesso 13 de junho 2010.

[19]. **Stratego/XT** Disponível em: < http://strategoxt.org/> Último acesso 13 de junho 2010.

[20]. Software transformation Systems. Disponível em <http://hydra.nixos.org/build/436750/download/1/manual/chunk-chapter/tutorial-software-transformation-systems.html#id1125717> Último acesso em 13 de junho 2010.

[21]. VISSER, Joost; SCHEERDER, Jeroen. **A Quick Introduction to SDF**. Abril 2005. Disponível em < ftp://ftp.stratego-language.org/pub/stratego/docs/sdfintro.pdf>

[22]. **Chapter 6. Syntax definition in SDF**. Disponível em <http://hydra.nixos.org/build/436750/download/1/manual/chunk-chapter/tutorial-sdf.html> Último acesso em 13 de junho 2010.

[23]. **The Syntax Definition Formalism SDF**. Disponível em <http://homepages.cwi.nl/~daybuild/daily-books/syntax/2-sdf/sdf.html> Último acesso em 13 de junho 2010.

[24]. **Analisador sintático LR.** Disponível em <http://pt.wikipedia.org/wiki/Analisador_sint%C3%A1tico_LR> Último acesso em 14 de junho 2010.

[25]. **Seção Context-free Syntax.** Disponível em <http://pt.wikipedia.org/wiki/Syntax_Definition_Formalism> Último acesso em 14 de junho 2010.

[26]. **The art of metaprogramming, Part 1: Introduction to metaprogramming.** Disponível em:< http://www.ibm.com/developerworks/linux/library/l-metaprog1.html **>** Acesso em 16 de março 2010

[27]. **AspectJ crosscutting object for better modularity.** Dsiponível em <www.eclipse.org/aspectj> Acesso em 16 de junho 2010

[28]. **Árvore sintática abstrata.** Disponível em <pt.wikipedia.org/wiki/Arvore_sintatica_abstrata> Acesso em 16 de junho 2010

[29]. **HEERING, J.; KLINT P.; OLIVIER, P. A.; VAN DEN BRAND, M. G. J. Compiling Language Definitions: The ASF+SDF Compiler.** P. 27. Disponível em <http://portal.acm.org/citation.cfm?id=567099&dl=GUIDE&coll=GUIDE&CFID=94389916&CFTOKEN=89227610>

[30]. **The Spoofax Language Workbench**. Disponível em < http://strategoxt.org/Spoofax> Ultimo acesso em 21 de junho 2010.

[31]. **Productivity and Cost Benefits of Software Product Lines**. Disponível em <http://www.softwareproductlines.com/benefits/productivity.html> Último acesso em 21 de junho 2010.

[32]. BONIFÁCIO, Rodrigo; TEIXEIRA, Leopoldo; BORBA, Paulo H. M. **Hephaestus A Tool for Managing SPL Variabilities.**

APÊNDICE A – Extração da transformação PreprocessFiles

```
    Transformation PreprocessFiles{

     BasicType{
        Path = String
     }

     Input{
       type = [String]
     }

     Output{
           Code{
                            preprocessFiles :: String -> [String] -> String -> IO()
                            preprocessFiles _ [] _ = return()
                            preprocessFiles flags files outputFolder = do
                               pid <- runCommand ("java -jar preprocessor.jar \"" ++ flags ++ "\""
++ (concatPaths files outputFolder))
                               waitForProcess pid >>= exitWith

                            concatPaths :: [String] -> String -> String
                            concatPaths [] _ = ""
                               concatPaths (str:strs) outputFolder = " \"" ++ outputFolder </> str
++ "\"" ++ (concatPaths strs outputFolder)
          }

           ExportCode{
                    preprocessFiles (outputPath ++ "/build.lst") (preProcessFiles splInstance)
outputPath
            }

     }
     SPL{
       Type {
         preprocessFiles = [String]
        }

        Initialize {
              []
        }
     }

     Code{
            data PreProcessor = PreProcessor
            {
                paths :: [Path]
            }

            instance Transformation PreProcessor where
            (<+>) (PreProcessor e) spl product =
                        let es = preProcessFiles product
                        in product { preProcessFiles = es ++ e }
     }
}
```

APÊNDICE B – Extração da transformação SelectUseCases

```
Transformation SelectUseCases{

        BasicType{
           Id = String
           Name = String
        }

        Input{

           Import("ImportsToImput.hs")
            Type = [String]
        }

        Output{

                Import("ImportsToExportProduct.hs")

                Code{
                exportUcmToLatex f ucm =
                 bracket (openFile f WriteMode)
                         hClose
                         (\h -> hPutStr h (show (ucmToLatex ucm)))
                exportUcmToXML f ucm =
                 bracket (openFile f WriteMode)
                         hClose
                         (\h -> hPutStr h (show (ucmToXML ucm)))
                }

                ExportCode{
                   exportUcmToLatex (outputPath ++ "/use-cases.tex") (ucm splInstance)
                   exportUcmToXML (outputPath ++ "/use-cases.xml") (ucm splInstance)
                }
        }

        SPL{
             Import("ImportsToInterpreterAndTypes.hs")

                Code{
                  ucmodel       = splUCM spl
                  emptyUCM      = ucmodel { useCases = [] , aspects = [] }
                }

                 Type {
                        ucm = UseCaseModel
                 }

                Initialize {
                 emptyUCM
                }
        }
        Gui{
             Import ("ImportsToGui.hs")
             Type = fileChooser
             InputName = "Use case model"
             ParserFunction =  "parseUseCaseFile"
    }

    Import("Hephastus\configuration-knowledge\src\ConfigurationKnowledge\PrettyPrinter\Latex.hs")
    Import("Hephastus\transormations\src\Transformations\UseCaseModel.hs")
    Import("Hephastus\uc-model\src\UseCaseModel\Parsers\BNFC\AbsUCM.hs")
    Import("Hephastus\uc-model\src\UseCaseModel\Parsers\BNFC\ErrM.hs")
    Import("Hephastus\uc-model\src\UseCaseModel\Parsers\BNFC\PrintUCM.hs")
}
```

APÊNDICE C – Extração da transformação SelectAndMoveComponent

```
Transformation SelectAndMoveComponent{

        Input{

         Import("TransforSelectAndMoveComponents.hs")
         Type = [String,String]

        }

        Output{

        ExportCode{

          print "\n Copying source files to output directory \n"
          print $ map (++ "\n") [fst c | c <- components p]

        }
   }


        SPL{
              Type{
               components = [(Id, Id)]
              }

                  Initialize{
                   []
                }
        }

        Gui{
                  Import ("ImportsToGui.hs")
                  Type = fileChooser
                  InputName = "Component model"
                  ParserFunction = "parseComponentModel"
   }

    Import("Hephastus\transormations\src\Transformations\ComponentModel.hs")
Import("Hephastus\component-model\src\ComponentModel\Parsers\ParserComponentModel.hs")
Import("Hephastus\component-model\src\ComponentModel\Parsers\AbsComponentModel.hs")
Import("Hephastus\component-model\src\ComponentModel\Parsers\SkelComponentModel.hs")
Import("Hephastus\component-model\src\ComponentModel\Parsers\ErrM.hs")
Import("Hephastus\component-model\src\ComponentModel\Parsers\LexComponentModel.hs")
Import("Hephastus\component-model\src\ComponentModel\Parsers\ParComponentModel.hs")

}
```

APÊNDICE D – Extração da transformação BindParameter

```
Transformation BindParameter {

        Input {
         Type = [String,String]
        }

        Code{

        module Trasformations.BindParameter
        where
        import Transformations.UseCaseModel

        -- | Transformation that binds scenario parameters to a selection of
        --   features. It deals with the kind of transformation named
        --   'variability in data'.

        data BindParameter = BindParameter {
         pmtId :: Id,      -- ^ formal parameter of scenarios
         featureId :: Id   -- ^ feature identifier
        }

        instance Transformation BindParameter where
                (<+>) (BindParameter pid fid) spl product = bindParameter steps (parenthesize
        options) pid product
          where
           steps = [s | s <- ucmSteps (ucm product), s `refers` pid]
                    options = concat (map featureOptionsValues [f | f <- flatten (fcTree (fc
        product)), fId (fnode f) == fid])
           bindParameter [] o pid p = p
                   bindParameter (s:ss) o pid p = bindParameter ss o pid (gReplaceParameterInScenario
        (sId s) pid o p)
        }

        -- this is the generic function for replacing a string into a step (identified
        -- by 'sid'). It follows the SYB pattern.

        gReplaceParameterInScenario :: Id -> String -> String -> InstanceModel -> InstanceModel
                gReplaceParameterInScenario sid fp ap = everywhere (mkT (replaceParameterInScenario
        sid fp ap))

        replaceParameterInScenario :: Id -> String -> String -> Scenario -> Scenario
        replaceParameterInScenario sid fp ap scn =
         let sts = steps scn
         in scn { steps = map (replaceStringInStep sid fp ap) sts

         }
}
```

APÊNDICE E – Extração da transformação SelectComponents

```
Transformation SelectComponents{

	Input{
	Type = [String]
	}

	Code{
		module Transformations. SelectComponents
		where

		import BasicTypes
		import ConfigurationKnowledge.Types

		-- | This datatype represents a transformation that retrieves
		--   a component name from a component id. The component name
		--   is retrieved from a file that maps an id to the name of
		--   the component.

		data SelectComponents = SelectComponents {
		 componentIds :: [Id]
		}

		instance Transformation SelectComponents where
		 (<+>) (SelectComponents ids) spl product =
			let
			 scs = [(snd x, snd x) | x <- splMappings spl, fst x `elem` ids]
			 ics = components product
			in case scs of
			 [] -> product
			 otherwise -> product { components = ics ++ scs }

		instance Show SelectComponents where
		 show (SelectComponents ids) = "selectComponents " ++ (show ids)
	}
	}
```

APÊNDICE F – Extração da transformação CreateBuildEntries

```
Transformation CreateBuildEntries{

        Output{

                Import ("ImportModule.hs")

                Code{
                exportBuildFile f es =
                 bracket (openFile f WriteMode)
                         hClose
                         (\h -> hPutStr h (concat [e ++ "\n" | e <- es]))

                createOutDir out c =
                 do
                  print $ "Selected output dir: " ++ out
                  print $ "Selected output component" ++ (snd c)
                  let new = out </> (snd c)
                  let newDir = dropFileName new
                  print new
                  print newDir
                  createDirectoryIfMissing True newDir
                }

                ExportCode{
                exportBuildFile  (o ++ "/build.lst") (buildEntries p)
                }
        }

        Input{
                Type = [String]
        }

        SPL{
             Type{
              buildEntries = [String]
             }

                Initialize{
                  []
              }
       }

    Code{

        module Transformation.CreateBuildEntries
        where
        import BasicTypes
        import ConfigurationKnowledge.Types

        data CreateBuildEntries = CreateBuildEntries {
          entries :: [String]
        }

        instance Transformation CreateBuildEntries where
         (<+>) (CreateBuildEntries e) spl product =
                let es = buildEntries product
                in product { buildEntries = es ++ e }

        instance Show CreateBuildEntries where
         show (CreateBuildEntries e) = "createBuildEntries " ++ (show e)

        }

}
```

APÊNDICE G – Definição da sintaxe léxica da SDF

```
module Common
exports
  sorts Identifier  Extension String Keyword Type

  lexical syntax
    [\ \t\n]  -> LAYOUT
    [A-Za-z][A-Za-z0-9]* -> Identifier
    [0-9] -> Digit
    Digit+ -> Decimal
    Digit+ "." Digit+ -> Float
    ".hs"  -> Extension
    ".lhs" -> Extension
    "True" -> Boolean
       "False"-> Boolean
    StrChar*   -> String
    ~[\"\n\r] -> StrChar
    ~[\"\n\r\ ]+ -> NonEmptyString
     HWord* -> HaskellWord
    ~[\21] -> HWord  %% Just rejects an unknown character

     "String" -> TypeString
     "Bool" -> TypeBool
     "Char" -> TypeChar
     "Int"-> TypeInt
     "Char" ("," "Char")* -> TypeCharSequence
     "String" ("," "String")* -> TypeStringSequence

     "[String]" -> SingleListString
     "[String,String]" -> ListPairString

     "textBox" -> TextBox
     "fileChooser" -> FileChooser

    "outputSrc" -> Keyword
    "inputSrc"    -> Keyword
    "splInstance" -> Keyword

    Boolean  -> Identifier {reject}
    Keyword -> Identifier {reject}
    FileChooser -> String {reject}
    TextBox -> String {reject}
    TextBox -> Identifier {reject}
    FileChooser -> Identifier {reject}

  context-free restrictions
    LAYOUT? -/- [\ \t\n\r]
    LAYOUT? -/- [\/].[\*]

  lexical restrictions
     String -/- [\ ]
```

APÊNDICE H – Definição da sintaxe livre de contexto de Transkell

```
module Transkell

imports Common

exports
  sorts BasicType Input Output SPLInstance Gui Imports PrimitiveType Types
  context-free start-symbols
    Transf
  context-free syntax

     "Transformation" TransfName "{" Body "}" -> Transf {cons("Transformation"),prefer}
     Identifier -> TransfName {cons("TransformationName")}
     BasicType? Output? Input SPLInstance? Gui? Imports HaskellCode? ->  Body {cons("Body"),prefer}

    "Output" "{" OutputBody "}"-> Output {cons("Output")}
     Imports HaskellCode? ExportCode -> OutputBody {cons("OutputBody")}
     Import* -> Imports {cons("ExtraImports")}

    "Code" "{" Code "}" -> HaskellCode {cons("HaskellCode")}
    "ExportCode" "{" Code "}" -> ExportCode {cons("ExportCode")}

    "Import" "(" FileExp ")"  ->  Import {cons("Import"),prefer}

     "\"" File "\"" -> FileExp  {cons("FileExp")}

     StringPath ExtensionCons -> File {cons("FilePath")}

     Extension -> ExtensionCons {cons("Extension")}

     String -> StringPath {cons("StringPath")}

     "BasicType" "{" Types "}"-> BasicType {cons("BasicType"),prefer}
     Declaration+ -> Types {cons("Declarations")}
     IdDeclaration "=" Type -> Declaration {cons("Declaration")}
     Identifier -> IdDeclaration {cons("TypeName")}
     "Input" "{" InputBody "}" -> Input {cons("Input"),prefer}
     Imports TypeDeclaration -> InputBody {cons("InputBody")}
        "Type" "=" TypeInput -> TypeDeclaration {cons("TypeDeclaration")}

        SingleListString -> TypeInput {cons("TypeInputSingle")}
        ListPairString -> TypeInput {cons("TypeInputPair")}

     "Gui" "{" Imports ComponentType InputName ValidateFunction  HaskellCode? "}"-> Gui {cons("Gui")}

     "Type" "=" Component -> ComponentType  {cons("ComponentType")}

     "InputName" "=" "\"" String "\""-> InputName {cons("InputName")}

     "ParserFunction" "=" "\"" Identifier "\"" -> ValidateFunction {cons("ValidateFunction")}

     "SPL" "{" SPLBody "}" -> SPLInstance {cons("SPLInstance")}

       Imports HaskellCode? SPLType SPLInitialize -> SPLBody {cons("SPLBody")}

      "Type" "{" SPLIdentifier "=" IdentifierType "}" -> SPLType {cons("SPLType")}
```

```
      Identifier -> SPLIdentifier {cons("SPLIdentifier")}
       String -> IdentifierType {cons("IdentifierType")}
      "Initialize" "{" InitializationBody "}" -> SPLInitialize {cons("SPLInitialize")}

       Variable -> InitializationBody {cons("InitializationBody"),prefer}
       String -> Variable {cons("Variable")}

       List -> SimpleType  {cons("SimpleType"),prefer}
       Tuple -> SimpleType  {cons("SimpleType")}
       StringExp -> SimpleType  {cons("SimpleType")}
       Character -> SimpleType  {cons("SimpleType")}
       NumberExp -> SimpleType  {cons("SimpleType")}
       Boolean   -> SimpleType  {cons("SimpleType")}

      "/'" StrChar "/'"-> Character  {cons("Character")}
      "\"" String "\"" -> StringExp {cons("String")}

      Number -> NumberExp  {cons("Number")}
      Decimal -> Number {cons("Decimal")}
      Float -> Number {cons("Float")}

      "(" TypeList ")" -> Tuple {cons("Tuple")}

      NonEmptyList  -> List  {cons("List")}
      EmptyList  -> List  {cons("List"),prefer}

      "[" "]" -> EmptyList {cons("EmptyList")}
      "[" TypeList "]" -> NonEmptyList {cons("NonEmptyList")}

       StringSequence -> TypeList {cons("TypeList")}
       BoolSequence   -> TypeList {cons("TypeList")}
       CharacterSequence -> TypeList {cons("TypeList")}
       NumberSequence   -> TypeList {cons("TypeList")}

       StringExp ("," StringSequence)? -> StringSequence {cons("StringSequence")}
       Bool ("," BoolSequence)? -> BoolSequence {cons("BoolSequence")}

       Character ("," CharacterSequence)? -> CharacterSequence {cons("CharacterSequence")}
       Number ("," NumberSequence)? -> NumberSequence {cons("NumberSequence")}
       Boolean -> Bool {cons("Boolean")}
       HaskellWord  -> Code

       FileChooser -> Component {cons("FileChooser")}
       TextBox -> Component {cons("Textbox")}

        PrimitiveType -> Type {cons("Type")}
        TypeList -> Type {cons("TypeList")}
        TypeTuple -> Type {cons("TypeTuple")}
        TypeString  -> PrimitiveType {cons("PrimitiveType")}
        TypeBool -> PrimitiveType {cons("PrimitiveType")}
        TypeChar -> PrimitiveType {cons("PrimitiveType")}
        TypeInt -> PrimitiveType {cons("PrimitiveType")}

    "[" TypeStringSequence "]" -> TypeList {cons("TypeListString")}
    "[" TypeCharSequence "]" -> TypeList {cons("TypeListChar")}
    "(" TypeStringSequence ")" -> TypeTuple {cons("TypeTupleString")}
    "(" TypeCharSequence ")" -> TypeTuple {cons("TypeTupleChar")}

  context-free restrictions
  HaskellWord -/- [\ \n\t]
```