Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Especificação e Verificação Formais de Boa-Formação em Linha de Produtos de Processo de Negócio

Giselle Barbosa Gomes Machado

Brasília
2012

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Especificação e Verificação Formais de Boa-Formação em Linha de Produtos de Processo de Negócio

Giselle Barbosa Gomes Machado

Dissertação apresentada como requisito parcial
para conclusão do Mestrado em Informática

Orientador
Prof. Dr. Vander Ramos Alves

Coorientador
Prof. Dr. Rohit Gheyi

Brasília
2012

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Mestrado em Informática

Coordenador: Prof.ª Dr.ª Mylene Christine Queiroz de Farias

Banca examinadora composta por:

Prof. Dr. Vander Ramos Alves (Orientador) — CIC/UnB
Prof.ª Dr.ª Carina Frota Alves — Centro de Informática/UFPE
Prof. Dr. Mauricio Ayala-Rincón — CIC/UnB

CIP — Catalogação Internacional na Publicação

Barbosa Gomes Machado, Giselle.

Especificação e Verificação Formais de Boa-Formação em Linha de Produtos de Processo de Negócio / Giselle Barbosa Gomes Machado. Brasília : UnB, 2012.
123 p. : il. ; 29,5 cm.

Dissertação (Mestrado) — Universidade de Brasília, Brasília, 2012.

1. Linha de Produtos de Processo de Negócio, 2. Verificação, 3. Alloy, 4. PVS, 5. Formalização, 6. Boa-formação, 7. Abordagem composicional

CDU 004

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Especificação e Verificação Formais de Boa-Formação em Linha de Produtos de Processo de Negócio

Giselle Barbosa Gomes Machado

Dissertação apresentada como requisito parcial
para conclusão do Mestrado em Informática

Prof. Dr. Vander Ramos Alves (Orientador)
CIC/UnB

Prof.ª Dr.ª Carina Frota Alves
Centro de Informática/UFPE

Prof. Dr. Mauricio Ayala-Rincón
CIC/UnB

Prof.ª Dr.ª Mylene Christine Queiroz de Farias
Coordenador do Mestrado em Informática

Brasília, 10 de dezembro de 2012

Agradecimentos

Este trabalho foi realizado com muita dedicação. Aprendi muito. Diversas pessoas contribuiram para que fosse possível realizá-lo.

Agradeço a Deus por tudo que tenho. À minha família pelo apoio e atenção. Aos meus pais que contribuiram e contribuem diretamente no meu aprendizado, na minha vida.

Ao meu marido que dedicou-se dia após dia para que eu conseguisse alcançar meu objetivo. Sempre companheiro e atencioso.

Aos meus filhos sempre calmos e compreensivos.

Ao meu orientador, professor Vander Alves, que no momento de menos esperança que tive soube escutar-me e mostrou-me um novo caminho.

Ao meu coorientador, professor Rohit Gheyi, que participou ativamente do meu trabalho sempre demonstrando muito conhecimento e paciência, além de enriquecer meu trabalho.

Aos colegas que conheci durante o mestrado da UnB, em especial aos colegas Lucinéia e Renato de Paula, que no momento de quase desistência contribuiram dando-me novos caminhos e muita esperança.

Aos colegas de orientação, Idarlan, Paula e Vinícius.

As professoras Carla Koike e Alba Cristina Magalhães que mostraram o que eu precisava para conseguir realizar meu mestrado em momentos de dificuldade.

Abstract

Quality assurance is a key challenge in product lines (PLs). Given the exponential growth of the number of products depending on the number of features, ensuring that all products meet given properties is a non-trivial issue. In particular, this holds for well-formedness in PLs. Although this has been explored at some levels of abstraction (e.g., implementation), this remains unexplored for business process PLs developed using a compositional approach, which has better modularity. Accordingly, in this dissertation we treat the problem of verification of well-formedness Business Process in Product Line, including the definition of well-formedness rules of business process, formal specification of compositional transformations, the analysis of examples and counterexamples generated by model checking, and the proof that transformations preserve the properties of all products without having to instantiate all variants. We performed a formalization of business process product line with the Alloy tool *Alloy Analyzer*, which has a formal specification language and a mechanism for verifying properties (*model checking*). Also formalization of business process product line and proofs of theorems were provided in the Prototype Verification System, which has a formal specification language and a proof assistant.

Keywords: Business Process Product Line, Verification, Alloy, PVS, Formalization, Well-formedness, Compositional Approach

Resumo

Um grande desafio em linha de produtos (LPs) é garantir as propriedades dos produtos gerados. Com o crescimento exponencial do número de produtos em função do número de características, assegurar que qualquer produto cumpre uma dada propriedade é um problema não trivial. Em particular, este é um problema válido para verificar a boa-formação em LPs. Embora tenha sido explorado em alguns níveis de abstração (por exemplo, a implementação), este problema permanece inexplorado para a linha de produtos de processo de negócios que foi desenvolvida usando uma abordagem composicional e possui melhor modularidade. Assim, nesta dissertação, tratamos do problema de verificação da boa-formação em Linha de Produtos de Processo de Negócio, incluindo a definição de regras de boa-formação de processos de negócio, especificação formal de transformações composicionais, a análise de exemplos e contraexemplos gerados pelo *model checking*, e a prova de que transformações preservam as propriedades de todos os produtos sem realmente ter que instanciá-los. Foi realizada uma formalização de linha de produtos de processo de negócio em Alloy com a ferramenta *Alloy Analyzer*, que possui uma linguagem de especificação formal e um mecanismo de verificação de propriedades (*model checking*). Também foram realizadas formalização de linha de produtos de processo de negócio e provas de teoremas no Prototype Verification System (PVS), que tem uma linguagem de especificação formal e um assistente de provas.

Palavras-chave: Linha de Produtos de Processo de Negócio, Verificação, Alloy, PVS, Formalização, Boa-formação, Abordagem composicional

Sumário

Lista de Figuras

Capítulo 1

Introdução

Os processos de negócio são utilizados para gerenciar trabalho e recursos (pessoas, equipamentos, informações, entre outros) em organizações [Dumas et al. (2005)]. Estes processos especificam atividades chave, documentos e artefatos produzidos de maneira específica por uma organização. Algumas atividades são realizadas manualmente e outras podem ser automatizadas pelo desenvolvimento de um ou mais sistemas. Tais processos são essenciais para alcançar metas de negócio e são importantes para maturidade organizacional [Jeston e Nelis (2006)].

No mapeamento de processos de negócio, ocorrem casos de replicação de atividades ou até processos inteiros e o fracasso em identificar tal replicação resulta em custos organizacionais desnecessários. Por exemplo, em uma organização onde a contratação de pessoas é regida pela Lei 8.112/90 no Brasil, em processos de solicitação de auxílios, sempre há a entrega de documentos e a validação destes documentos pela seção de Recursos Humanos. Quando os processos são mapeados estas atividades são repetidas em todos os processos de solicitação de auxílios (por exemplo, auxílio natalidade e auxílio pré-escolar). Esta replicação é claramente contra metas de negócio, por exemplo, eficiência, maximização de lucro, entre outras.

A fim de reduzir este risco e aperfeiçoar a distribuição de recursos organizacionais, é importante saber da existência de replicação e então lidar com tal comunalidade e variabilidade. Assim, linha de produtos de processo de negócio surgiu para alavancar conceitos e técnicas de Linha de Produtos de Software (LPS) [Clements e Northrop (2001)] para processo de negócio: uma linha de produtos de processo de negócio (LPPN) é uma linha de produtos cujos produtos são um conjunto de processos de negócios para uma determinada organização [Machado et al. (2011)].

Garantia de propriedades é um desafio chave em linhas de produtos. Por outro lado, o crescimento exponencial de variabilidade de um produto em função do número de características é um problema não trivial. Em particular, neste trabalho, tratamos do problema de verificação da boa-formação em LPPN. Um processo de negócio válido é definido pelas regras de boa-formação (*well-formedness*). Embora tenha sido explorado em alguns níveis de abstração (por exemplo, a implementação), este permanece na maior parte inexplorado para LPPN. Embora um trabalho recente investigue esta questão formal para processos de negócios [Rosa et al. (2011)], ele só foi feito para uma abordagem anotativa e a prova formal não foi feita em um assistente de provas, o que pode aumentar a probabilidade de erro. Abordagens anotativas têm limitações de modularidade e de legibilidade, ao passo

que as abordagens composicionais atenuam estes problemas e também têm sido amplamente utilizadas para o gerenciamento de variabilidade em linhas de produtos [Aleixo et al. (2012)]. Além disso, do lado da formalização, uma prova feita com um assistente de provas é suficientemente detalhada para fins de replicação e de certificação.

Neste contexto, Machado et al. (2011) caracterizaram a variabilidade no processo de negócio e apresentaram uma abordagem composicional para administrar tal variabilidade utilizando conceitos de Linha de Produtos de Software (LPS), sem, no entanto, se preocupar em garantir a boa-formação das instâncias geradas pela LPS.

Em particular, a ideia central deste trabalho consiste em garantir que as instâncias de uma LPPN sejam bem-formadas, que satisfaçam propriedades de boa-formação, sem a necessidade de verificar a boa-formação individualmente. Para isto, foram realizadas uma especificação e uma verificação formais que estão mais detalhadas nas seções a seguir.

1.1 Problema

A geração de produtos da LPPN é feita pela seleção de características e pela composição dos artefatos associados as características do modelo de características [Clements e Northrop (2001)]. Para garantir que o resultado das combinações de características gera produtos bem-formados, é preciso verificar se todas as instâncias de processos gerados por uma LPPN são bem-formadas.

O problema é exatamente verificar, de modo escalável, se cada instância gerada pela LPPN é válida, pois a medida que o modelo de características cresce, a quantidade de produtos cresce exponencialmente com o número de características.

1.1.1 Exemplo Motivante

Nesta seção, descrevemos como o acréscimo de características no modelo de características pode aumentar exponencialmente a quantidade de produtos gerados pela LPPN. Para `n` igual ao número de características, temos a função 2^{n-1} configurações possíveis.

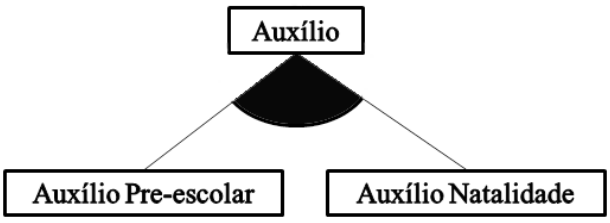

Figura 1.1: Modelo parcial de características com três características

Na Figura 1.1 e Figura 1.4 são apresentados modelos parciais de características de LPPN. Um modelo de características contém um conjunto de configurações possíveis que podem ser selecionadas para gerar um produto. Este artefato descreve as possíveis configurações para linha de produtos, apresenta especificações variantes e comuns entre os produtos e é usado para descrever e para selecionar produtos com base nas características que eles suportam.

No exemplo da Figura 1.1 temos um modelo com três características onde a característica pai `Auxílio` está relacionada com duas características `Auxílio Pre-escolar` e

`Auxílio Natalidade` representado no modelo pelo arco preto preenchido, onde pelo menos uma característica deve ser selecionada.

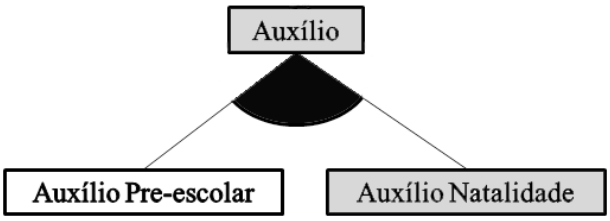

Figura 1.2: Modelo parcial de características com três características com uma configuração selecionada

Na Figura 1.1 temos um modelo de característica com apenas três características. Quando selecionarmos um auxílio, podemos escolher três configurações possíveis. São elas: `c1`={Auxílio, Auxílio Pre-escolar, Auxílio Natalidade}, `c2`={Auxílio, Auxílio Pre-escolar} e `c3`={Auxílio, Auxílio Natalidade}.

Associada à configuração da Figura 1.2 marcada em cinza, temos como produto o processo de negócio Auxílio Natalidade que pode ser visto na Figura 1.3. Para saber se ele é bem-formado deve ser feita uma análise para verificar se está de acordo com as regras de boa-formação descritas na linguagem de processo de negócio especificada neste trabalho. As regras de boa-formação são propriedades criadas neste trabalho que estão baseadas nas propriedades descritas informalmente no manual BPMN [OMG (2009)]. No caso da Figura 1.3, o processo de negócio é bem-formado.

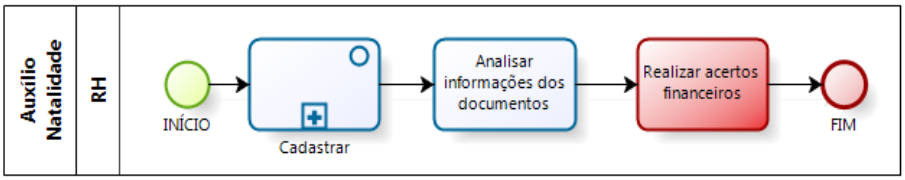

Figura 1.3: Exemplo de processo de negócio: Auxílio Natalidade

Ao adicionarmos uma característica, conforme Figura 1.4, passamos a ter sete configurações possíveis: `c1`={Auxílio, Auxílio Pre-escolar, Auxílio Natalidade, Auxílio Moradia}, `c2`={Auxílio, Auxílio Pre-escolar, Auxílio Natalidade}, `c3`={Auxílio, Auxílio Pre-escolar, Auxílio Moradia}, `c4`={Auxílio, Auxílio Natalidade, Auxílio Moradia}, `c5`={Auxílio, Auxílio Pre-escolar}, `c6`={Auxílio, Auxílio Natalidade} e `c7`={Auxílio, Auxílio Moradia}.

De modo geral, quanto mais características houver no modelo de características maior vai ser a quantidade de produtos na linha de produtos para serem verificados individualmente e esta função é exponencial.

1.2 Solução Proposta

Visando garantir de forma escalável que a LPPN gera produtos bem-formados, foram realizadas especificações e verificações formais da LPPN.

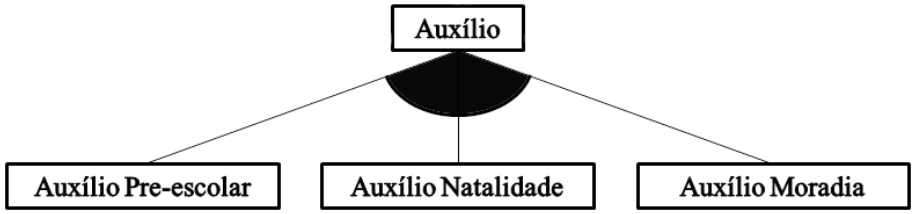

Figura 1.4: Modelo parcial de características com quatro características

Inicialmente, considerando que especificação formal não é trivial, utilizamos Alloy com o intuito de refinar incrementalmente a especificação com auxílio do mecanismo de verificação de propriedades *model checking* (ferramenta *Alloy Analyzer*). A especificação em Alloy e a utilização da ferramenta auxiliaram diretamente na criação das regras de boa-formação para um modelo de processo de negócio e facilitaram o entendimento da linguagem da LPPN [OMG (2009)].

Em seguida, com a especificação refinada e estável formalizamos tipos, regras de boa-formação e duas transformações da LPPN na linguagem PVS. Não fizemos uma especificação gradual de tipos e nem das regras de boa-formação no PVS, somente em Alloy. A evolução da especificação em Alloy auxiliou na especificação da linguagem e das transformações no *PVS* de forma clara.

No PVS foi utilizado o provador de teoremas para provar lemas e teoremas mais gerais, buscando garantir para a LPPN a boa-formação de todas as instâncias. Uma das vantagens em utilizar o PVS com relação ao Alloy foi de obter a prova de que as transformações preservam a boa-formação de todos os produtos com a utilização do provador de teoremas, enquanto que Alloy só faz verificação em um escopo limitado. A especificação formal e a verificação com o auxílio do assistente de provas no PVS foram utilizadas para garantir a boa-formação dos produtos da LPPN. Neste processo, a simplificação da linguagem e abordagem incremental foram utilizadas para tratar a complexidade.

Trabalhar com métodos formais possibilitou a precisão na especificação da LPPN deixando-a mais coerente, aumentando a confiança na especificação. Como consequência, foi possível corrigir erros de especificação e diminuir custos provenientes de falhas, não deixando nenhuma dúvida em relação ao comportamento da LPPN para cada configuração possível.

1.3 Contribuição

Esta dissertação apresenta as seguintes contribuições:

- **Verificação escalável Geral** (Capítulos 3 e 4): a verificação é formal e garante a boa-formação de todos os produtos sem verificar cada um individualmente.
- **Especificação formal em Alloy** (Capítulo 3): a formalização em Alloy facilitou o entendimento da linguagem e auxiliou diretamente na criação das regras de boa-formação para um modelo de processo de negócio.
- **Especificação formal em PVS** (Capítulo 4): a formalização no PVS especifica precisamente uma linguagem da LPPN, as propriedades de boa-formação e a

composição de transformações. Ela se concentra em como essas transformações contribuem para boa-formação dos produtos da LPPN.

- **Especificação formal de processo de negócio** (Capítulos 3 e 4): a especificação de processos de negócio em Alloy e em PVS criando sintaxe formal para processos de negócio.
- **Relato de experiência** (Capítulo 5): experiência na utilização do assistente de provas, do *model check*, da combinação do *model check* com o assistente de provas e em criar estratégias de provas. Esta experiência pode ser útil para fins semelhantes.

Como resultado deste trabalho, a autora foi autora e co-autora nos seguintes artigos:

- Giselle Machado, Vander Alves, e Rohit Gheyi. Formal specification and verification of well-formedness in business process product lines. In *6th Latin American Workshop no Aspect-Oriented Software Development: Advanced Modularization Techniques*, LAWASP'12, 2012.
- Idarlan Machado, Rodrigo Bonifácio, Vander Alves, Lucinéia Turnes, e Giselle Machado. Managing variability in business processes: an aspect-oriented approach. In *Proceedings of the 2011 international workshop on Early aspects*, EA'11, pages 25-30, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0645-4. doi: 10.1145/1960502.1960508. URL http://doi.acm.org/10.1145/1960502.1960508.

1.4 Estrutura

O restante desta dissertação está organizada da seguinte forma:

- O Capítulo 2 apresenta a fundamentação teórica com os principais conceitos para o entendimento deste trabalho. Definimos o que é Linha de Produtos de Software (LPS), Processo de Negócio, Linha de Produtos de Processo de Negócio e explicamos as linguagens de especificação formal PVS e Alloy.
- O Capítulo 3 corresponde à especificação formal e verificação escalável na LPPN em Alloy.
- O Capítulo 4 corresponde à especificação formal e verificação escalável na LPPN em PVS.
- O Capítulo 5 aborda a experiência adquirida com formalização, comenta sobre as contribuições deste trabalho, discute trabalhos relacionados, apresenta as conclusões desta pesquisa e discute propostas de trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Neste capítulo, são apresentados temas relevantes que fundamentam o nosso trabalho: Linha de Produtos de Software (Seção 2.1), Processo de Negócio (Seção 2.2), Linha de Produtos de Processo de Negócio (Seção 2.3), Alloy (Seção 2.4) e PVS (Seção 2.5).

2.1 Linha de Produtos de Software

Uma Linha de Produtos de Software é um conjunto de produtos de software relacionados que compartilham funcionalidades comuns em um mesmo domínio e que são gerados a partir de componentes reusáveis.

Os produtos ou instâncias da *LPS* são gerados pela composição e adaptação dos artefatos genéricos com artefatos específicos a cada produto. Os benefícios potenciais de *LPS* incluem redução de tempo e custos de desenvolvimento e aumento da produtividade e qualidade dos produtos. Em particular, isto implica que a qualidade dos artefatos genéricos e específicos esteja combinada de forma eficiente para garantir a qualidade de produtos na LPS.

Na abordagem de *LPS*, os principais desafios são o gerenciamento de variabilidade dos artefatos e as estratégias de adoção [Krueger (2001)]. Os principais elementos são: *Feature Model (FM)*, *Configuration Knowledge (CK)* e *Asset Base (AB)*. Um *FM* apresenta especificações variantes e comuns entre os produtos e é usado para descrever e selecionar produtos com base nas características que eles suportam. Uma característica é proeminente ou distinta de aspecto visível ao usuário, qualidade ou característica de um sistema de software [Kang et al. (1990)]. *Asset Base* corresponde aos componentes, classes, arquivos de propriedades e outros artefatos, que são compostos de diferentes maneiras para especificar ou construir os diferentes produtos [Clements e Northrop (2001)]. O *CK* relaciona características e recursos, especificando quais as combinações de recursos possíveis os *AB* devem implementar. Assim, um *CK* pode ser usado para realmente construir um determinado produto com características escolhidas para este produto [Czarnecki e Eisenecker (2000)].

Em uma *LPS* o esforço inicial para desenvolvimento de produtos é maior, tanto em tempo quanto em custo, do que para construção de um produto que não utiliza *LPS*, devido à construção dos artefatos da *LPS*. Mas após a construção da *LPS*, o esforço de desenvolvimento de novos produtos tende a ser reduzido, uma vez que os ciclos de

desenvolvimento são mais curtos, pois novos produtos são desenvolvidos com o reuso estratégico e sistemático de artefatos existentes.

2.2 Processo de Negócio

Um processo de negócio compreende um conjunto de informações associadas a atividades, uma atividade pode ser uma tarefa ou um subprocesso. A tarefa é uma atividade que não pode mais ser decomposta. O subprocesso é um conjunto de atividades que podem ser decompostas com mais detalhes. As atividades manipulam e utilizam os recursos da empresa, com a finalidade de agregar valor e, ao final, gerar produtos, serviços ou informações. As atividades podem ser compostas (subprocessos) ou atômicas (tarefas) [Jeston e Nelis (2006)].

Uma visão histórica para processo tem suas origens na Revolução Industrial, com o estabelecimento das fábricas. A fim de permitir um maior volume e a qualidade da produção, instituiu-se a noção de trabalho em série, feita passo a passo. Assim, verifica-se a ideia de funcionamento de uma sequência de passos para a obtenção de um determinado produto.

Depois de quase dois séculos, um processo é apresentado como "um conjunto definido de atividades ou comportamentos executados por humanos ou máquinas para alcançar um ou mais objetivos", de acordo com o Guia do Conjunto de Gestão de Processos de Negócios Comum do Conhecimento (BPM CBOK) [ABPMP (2009)].

O Guia de Simplificação [SEGEP] é um documento elaborado pelo Ministério do Planejamento. Ele indica como um dos principais objetivos no exercício de atividades de um processo de agregação de valor deve ser. Assim, um processo pode ser definido como a transformação de entradas pelas atividades de valores para os clientes ou cidadãos.

Neste contexto, a Gestão de Processos de Negócios (BPM) atua como uma abordagem organizada e instrumental, a fim de identificar, documentar, desenhar, operar, monitorar e melhorar os processos para alcançar resultados que são consistentes e alinhados com os objetivos organizacionais. Conceitos de BPM permitem a uma organização, não só refletir sobre os resultados esperados, mas também modelá-los e executá-los.

O principal objetivo do BPMN é fornecer uma notação que é facilmente compreensível por todos os usuários de negócio, desde os analistas de negócio que criam os rascunhos iniciais dos processos, até as pessoas de negócio que irão gerenciar e monitorar estes processos, passando pelos desenvolvedores técnicos responsáveis pela implementação da tecnologia e pelos que irão executar os processos [OMG (2009)].

Um processo na notação *BPMN* pode ser privativo (é utilizado quando não há interesse em verificar a interação entre processos), abstrato (representa a interação entre um processo principal e outro processo participante, sendo que não se preocupa com o conteúdo do processo participante) ou colaborativo (descreve a interação entre duas entidades do negócio ou mais, sendo que o conteúdo do fluxo é especificado em todas as entidades). Neste trabalho, por simplificação, tratamos apenas de processos privativos.

Em BPMN temos nove elementos, são eles: artefatos, conectores, atividades, subprocessos, gateways, eventos de início, eventos intermediários, eventos de fim e objetos. Os artefatos podem ser *pool* ou piscina, *lane* ou raia e milestone ou etapa. Uma *pool* representa um processo ou uma entidade (Figura 2.1). Uma *lane* é uma subpartição dentro

da *pool* e é utilizada para organizar e categorizar a *pool* (Figura 2.2). Uma *milestone* é uma subpartição dentro do processo utilizada para organizar o processo em etapas (Figura 2.3). Neste trabalho por simplificação especificamos apenas *pool*.

Figura 2.1: Pool

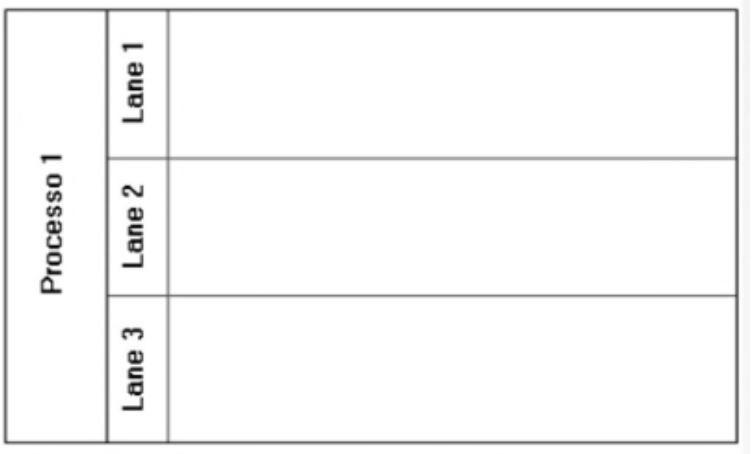

Figura 2.2: Lane

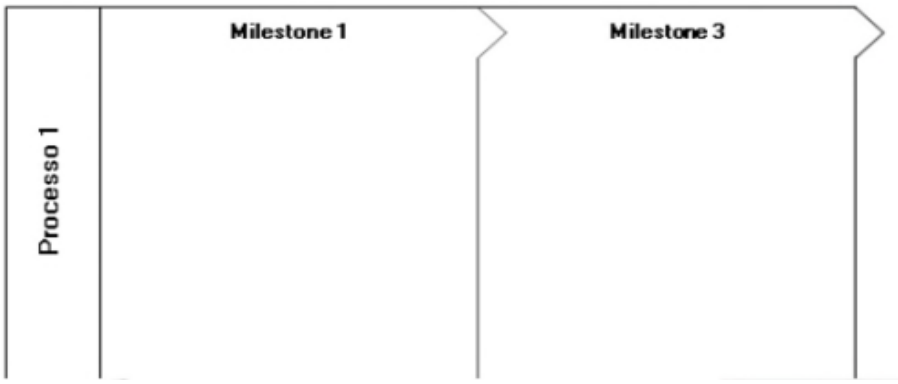

Figura 2.3: Milestone

Existem três tipos de conectores: fluxo de sequência, fluxo de mensagem e associação. Neste trabalho especificamos apenas o fluxo de sequência (representado na Figura 2.4)

que é usado para mostrar a ordem em que as atividades serão executadas. Cada fluxo tem só uma origem e um destino.

Figura 2.4: Conector do tipo fluxo de sequência

Neste trabalho foi utilizada apenas a atividade padrão, que é o tipo mais frequentemente usado durante os estágios iniciais do desenvolvimento do processo. A Figura 2.5 ilustra uma atividade padrão.

Figura 2.5: Atividade do tipo padrão

Assim como as atividades, existem diversos tipos de eventos de início e eventos de fim. Porém, especificamos apenas o tipo padrão de evento de início e fim. O tipo padrão de evento de início, ilustrado na Figura 2.6, é usado para iniciar o processo em cada processo só pode ter um único início. O evento de início nunca terá um fluxo de sequência chegando nele, pois só pode ter fluxo de sequência saindo dele.

Figura 2.6: Evento de início do tipo padrão

Já o evento de fim do tipo padrão, ilustrado na Figura 2.7, é usado para terminar o processo, sendo que um processo pode ter um ou mais eventos de fim. Este tipo de evento só pode ter fluxo de sequência chegando nele, nunca pode haver fluxo de sequência saindo dele.

Figura 2.7: Evento de fim do tipo padrão

A Figura 2.8 ilustra um subprocesso do tipo padrão. Quando não se tem uma atividade que não se encerra em si mesma, ou seja, não é uma atividade atômica, ela pode ser transformada em subprocesso. O subprocesso é parte do processo pai e não pode ser utilizado em outro processo.

Neste trabalho, por simplificação, não foram utilizados *gateways*, eventos intermediários e nem objetos.

Na Figura 1.3 é ilustrado um exemplo de processo de negócio que mapeia o fluxo para um servidor público regido pela Lei 8.112/1990 que dispõe sobre o regime público dos servidores públicos civis da União, das autarquias e das fundações públicas federais

Figura 2.8: Subprocesso do tipo padrão

(adquirir o auxílio natalidade - artigo 196)[Nacional (1990)]. Este processo é automatizado pelo sistema SIAPE (Sistema Integrado de Administração de Recursos Humanos que pertence à Secretaria de Gestão Pública do Ministério do Planejamento). O processo inicia-se com um cadastro no RH (recursos humanos); em seguida, é realizada uma análise da documentação do servidor e, por fim, são feitos os acertos financeiros.

A identificação e a análise de processos devem ser compatíveis com os objetivos de negócio da organização. Muitas empresas e instituições buscam metodologias para modelagem de processos de negócio devido aos muitos problemas relacionados à falta de qualidade de um sistema de informação têm como causa principal a falta de qualidade da modelagem de processos de negócio. O trabalho proposto em Machado et al. está baseado no problema da necessidade de caracterização da variabilidade de processo de negócio [Machado et al. (2011)].

O manual da notação BPMN [OMG (2009)] especifica informalmente como um processo de negócio deve ser modelado. A partir deste manual foi possível criar parte das regras de boa formação para processo de negócio utilizadas neste trabalho. Na Figura 2.9, é possível visualizar um exemplo de processo mal-formado (com um "X" a direita) e outro bem-formado.

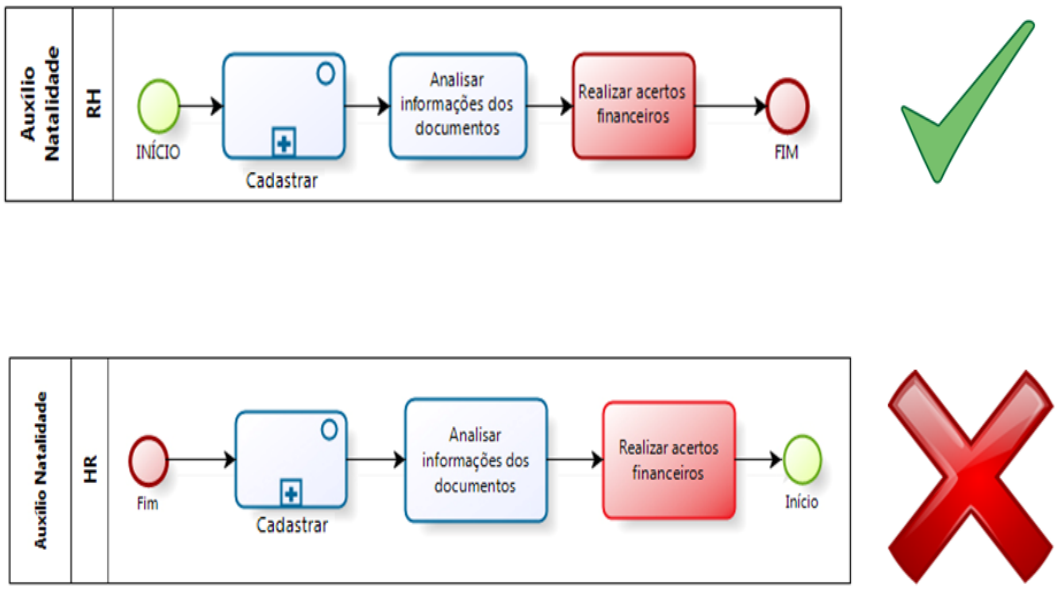

Figura 2.9: Exemplos de processos de negócio bem-formado e mal-formado

Na Figura 2.10, temos um trecho retirado do manual de BPMN [OMG (2009)] que determina que um fluxo de sequência não pode conectar um evento de início, ou seja, não pode haver uma seta que liga a atividade "Realizar acertos financeiros" ao evento "Início".

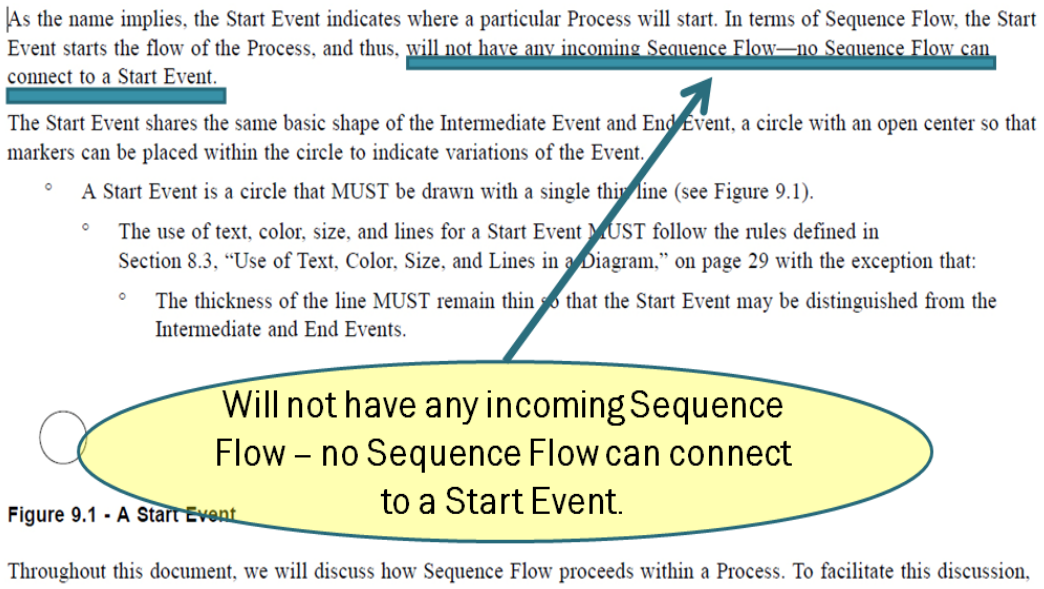

Figura 2.10: Trecho retirado do manual BPMN [OMG (2009)]

2.3 Linha de Produtos de Processo de Negócio

O conceito de processos de negócio abrange elementos tais como atividades, tarefas, processos, recursos, dentre outros, que são comuns em uma organização. Durante a execução de um processo de negócio é comum identificar replicação de certas atividades que podem comprometer a eficiência do processo resultando em perdas em relação ao custo e à qualidade.

Como exemplo de replicação, comparando-se as Figuras 1.3 e 2.11, é possível verificar que há a replicação de atividades em processos de negócio distintos. Em ambos os processos há a entrega de documentos que é realizada no cadastro e a validação destes documentos pela seção de Recursos Humanos.

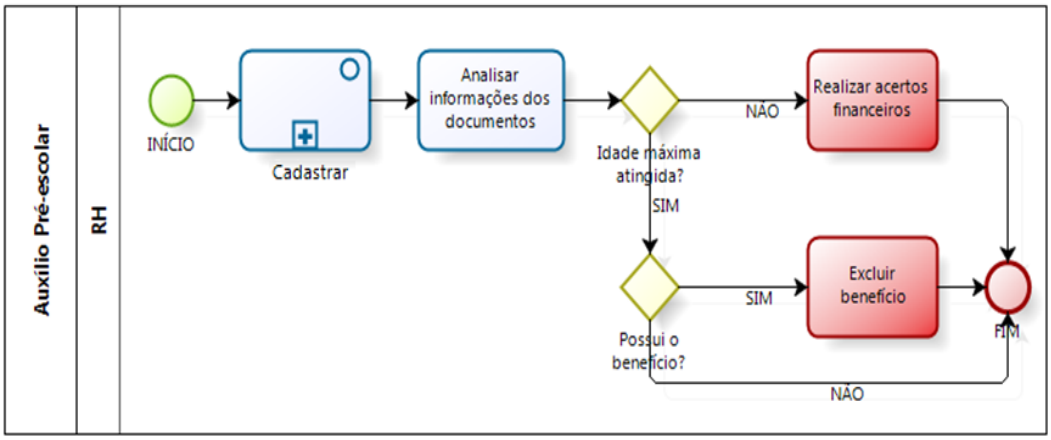

Figura 2.11: Exemplo de processo de negócio: Auxílio Pré-escolar

Uma vez que processos de negócio podem ser abordados no contexto de Linhas de Produtos de Software, parte do problema de replicação pode ser solucionado quando for considerado neste cenário o gerenciamento da variabilidade [Machado et al. (2011)].

Uma Linha de Produtos de Processo de Negócio (LPPN) é um conjunto de instâncias de processos relacionados que são gerados a partir da seleção e combinação válida de características.

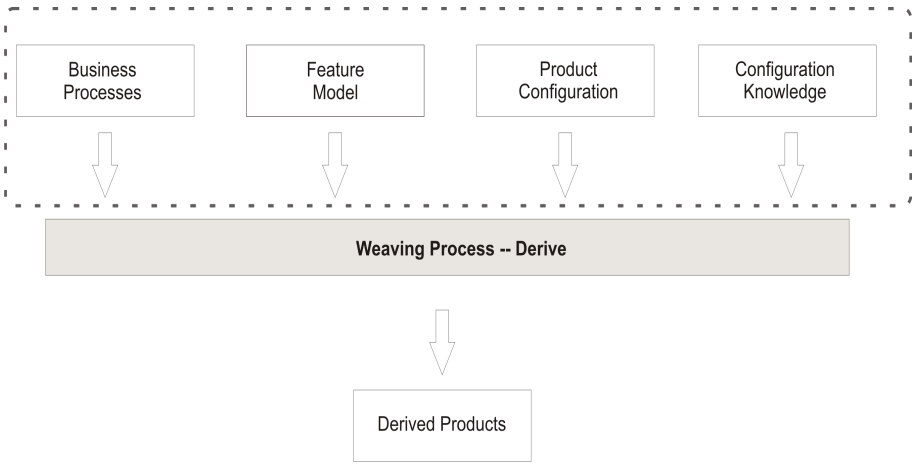

Figura 2.12: Linha de Produtos de Processo de Negócio [Machado et al. (2011)]

Neste trabalho, utilizamos uma abordagem composicional para gerenciar a variabilidade em linhas de produto de processo de negócio [Machado et al. (2011)], que é mostrada na Figura 2.12. Esta abordagem foi utilizada devido a diminuição do espalhamento e do entrelaçamento das variabilidades e comunalidades, pois as variabilidades ficam separadas das comunalidades [Machado et al. (2011)].

- **Processo de Negócio**: especifica atividades que existem, por exemplo em uma organização. O gerenciamento dessas atividades que compõe um processo, permite tratar características comuns e variantes que irão então formar o *FM*;
- ***Feature Model (FM)***: descreve características relevantes de um dado domínio e detalha restrições que existem entre as mesmas. No contexto de processos de negócio, a partir da análise de processos existentes, são extraídas características comuns e variantes que irão compor os membros (instâncias de processos) os quais devem conter configurações válidas de uma LPPN;
- ***Product Configuration***: conjunto de características descrevendo um membro específico de uma LPPN. Uma importante propriedade é que cada configuração do produto de processo deve obedecer as relações e restrições especificadas no *FM* correspondente;
- ***Configuration knowledge (CK)***: corresponde a uma lista de itens de configuração, que relaciona características que são expressas para as tarefas (ou transformações) usadas para gerar automaticamente um membro de uma LPPN. A Figura 2.13 ilustra a intuição do funcionamento do *CK*;

- ***Weaving Process***: é uma função que realiza sucessivos refinamentos até obter a instância de um processo (o produto). Forma composicional de trabalho usando transformações. As transformações são funções implementadas em Haskell e nelas estão implementadas a lógica referente ao tratamento de variabilidades no cenário de processos de negócio. Portanto, o passo `Weaving Process` representa a execução dessas funções.
- ***Derived Products***: são os produtos gerados pela LPPN.

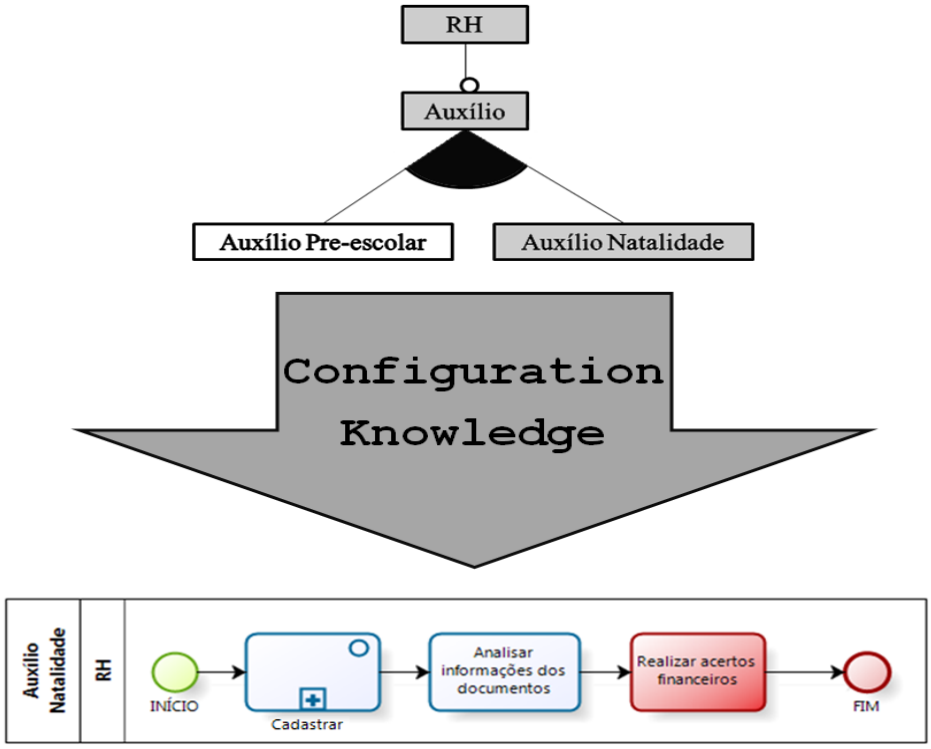

Figura 2.13: Exemplo do funcionamento do *CK*

O trabalho Machado et al. propõe a extensão dos conceitos apresentados no trabalho de Bonifácio [Bonifácio (2010)]. No trabalho de Machado et al. inicialmente é apresentado, usando abordagem extrativa, o gerenciamento da variabilidade no domínio de Recursos Humanos em uma dada organização, sem, no entanto, se preocupar em garantir a boa-formação das instâncias geradas pela LPPN [Machado et al. (2011)].

A implementação da LPPN existente foi feita na linguagem de programação funcional Haskell, que possui apoio da formalização limitada. Na especificação original em Haskell as regras de boa-formação de processos de negócio foram incorporadas ao *typechecker*, elas refletem principalmente a especificação BPMN [OMG (2009)].

Fundamental no processo de derivação é a aplicação das transformações (especificado em CK). Uma condição suficiente para garantir uma boa-formação do produto resultante é garantir que cada transformação conserva a boa-formação. Neste trabalho, estamos interessados em verificar se tais transformações preservam esta propriedade.

2.4 Alloy

Alloy é uma linguagem de modelagem formal, com base em lógica de primeira ordem, que dá uma notação matemática para especificar objetos e seus relacionamentos. Alloy tem sintaxe, sistema de tipos e semântica simples, sendo projetada para a análise automática. Além disso, é uma linguagem totalmente declarativa [Jackson (2011)]. Neste trabalho utilizamos *Alloy 4.1*.

Um modelo de Alloy pode conter três tipos de construções de especificação: de assinatura (introduzir novos tipos - *signatures*) (Seção 2.4.1), de restrição (restrições de registro e expressões - *facts*, *predicates* e *functions*) (Seções 2.4.2 e 2.4.3) e de análise (são utilizados para realizar análise - *assertion*, *run* e *check*) (Seção 2.4.4). As construções de especificação de modelos descrevem domínios e definem propriedades sobre estes domínios.

2.4.1 Assinaturas

Uma assinatura introduz um conjunto de objetos. A declaração

```
sig FlowObject {}
```

introduz um conjunto nomeado `FlowObject`. A assinatura é realmente mais do que apenas um conjunto porque pode incluir declarações de relações e pode introduzir um novo tipo de forma implícita. Mas é conveniente usar o termo "assinatura" livremente para se referir a esta estrutura maior e ao conjunto associado a uma determinada assinatura, por isso é possível falar, por exemplo, dos "elementos da assinatura", ou seja, os átomos contidos na definição [Jackson (2011)]. A seguir temos mais duas declarações de assinaturas

```
sig BusinessProcess{}
sig BusinessProcessModel {
    processes: set BusinessProcess
}
```

que introduzem uma assintura nomeada `BusinessProcess` e outra assinatura nomeada `BusinessProcessModel`. Neste caso, temos que a assinatura `BusinessProcessModel` é composta por um conjunto de `BusinessProcess`.

Um conjunto pode ser introduzido como um subconjunto de outro conjunto, assim

```
sig Start extends FlowObject {}
```

introduz um conjunto chamado `Start` que é um subconjunto de `FlowObject`. A assinatura `Start` é uma extensão ou uma subassinatura de `FlowObject`. Uma assinatura, tal como `FlowObject`, que é declarada de forma independente de qualquer outra é uma assinatura de nível superior. As extensões de uma assinatura são mutuamente disjuntas, como são assinaturas de nível superior. Assim, dada as declarações

```
sig FlowObject {}
sig Transition {}
sig Start extends FlowObject {}
sig End extends FlowObject {}
```

pode-se inferir que `FlowObject` e `Transition` são disjuntos e `Start` e `End` são disjuntos (mas não isso `FlowObject = Start + End`).

Uma assinatura abstrata não possui elementos exceto os pertencentes às suas extensões. Então ao escrevermos

```
abstract sig FlowObject {}
sig Start extends FlowObject {}
sig End extends FlowObject {}
```

por exemplo, introduzimos três conjuntos com restrições implícitas

```
Start in FlowObject
End in FlowObject
```

porque `Start` e `End` extendem `FlowObject`, e

```
FlowObject in Start + End
```

porque `FlowObject` é abstrato. Então

```
FlowObject = Start + End
```

e `Start` e `End` são partições de `FlowObject`.

Na especificação a seguir temos a assinatura `Transition` que possui um mapeamento feito pelos `startObject` e `endObject`, que são assinaturas `FlowObject`, associando `Transition` à `FlowObject`.

```
sig Transition {
    startObject: FlowObject,
    endObject: FlowObject
 }
```

Na especificação

```
one sig Start, End extends FlowObject {}
```

`one` representa uma enumeração, existem apenas dois elementos que fazem parte do conjunto `FlowObject`, `Start` e `End`.

2.4.2 Fatos

Restrições que são assumidas para serem mantidas são registradas como fatos. Um modelo pode ter qualquer número de fatos. A ordem em que os fatos aparecem, e a ordem de restrições dentro de um fato, é irrelevante. Pode-se dar a um fato um nome mnemônico exclusivo [Jackson (2011)].

A expressão `t.startObject` denota o mapeamento de `FlowObjects` para o conjunto de `transitions t`. A seguir é especificado um fato que define que o conjunto de `transitions` de `BusinessProcess` não pode possuir `Transition` que possua o `startObject` pertencente ao subconjunto da assinatura `End` (`!` representa a negação e `in` representa um operador de conjunto em Alloy). Além disto, o `startObject` não pode ser vazio (`none` representa o operador constante de vazio).

```
fact{
    all t: BusinessProcess.transitions |
           t.startObject !in End and
           t.startObject != none
}
```

No exemplo especificado a seguir, cada `startObject` pertencente ao conjunto de `transitions` de `BusinessProcess` é acessível a partir do `FlowObject Start` (o símbolo * representa o operador relacional de fecho reflexivo-transitivo).

```
fact{
    Start in BusinessProcess.*transitions.startObject
}
```

2.4.3 Funções e Predicados

Muitas vezes, há limitações que não devem ser gravadas como fatos. Pode-se querer analisar o modelo com uma restrição incluída e excluída; verificar se uma restrição segue a partir de algumas outras restrições, ou declarar uma restrição para que possa ser reutilizada em diferentes contextos. Expressões de predicados são para esses fins. Expressões de funções são para reutilização [Jackson (2011)].

Uma função é um nome para uma expressão, com zero ou mais declarações para argumentos, e uma expressão de declaração para o resultado. Quando a função é usada, uma expressão deve ser fornecida para cada argumento, o seu significado é apenas uma expressão da função, com cada argumento substituído por sua expressão instanciada.

Um predicado é um nome para uma restrição, com zero ou mais declarações para argumentos. Quando o predicado é usado, uma expressão deve ser fornecida para cada argumento, o seu significado é apenas restrição do predicado com cada argumento substituído por sua expressão instanciada.

Por exemplo, a função `SimpleSelect` que seleciona processos de negócio que fazem parte de um modelo recebido pela função.

```
sig BusinessProcess{}
sig BusinessProcessModel{processes: set BusinessProcess}
```

```
fun SimpleSelect[spl: BusinessProcessModel]:
    BusinessProcess{
        bp: spl.processes
    }
```

O predicado `wellFormedModel` verifica se um `BusinessProcessModel` possui processos que foram selecionados pela função `SimpleSelect` que satisfazem o predicado `wellFormed`.

```
pred wellFormedModel[p:BusinessProcessModel] {
    all bp: SimpleSelect[p] | wellFormed[bp]
}
```

2.4.4 Análise

Alloy tem também os pontos de análise, onde há um mecanismo de verificação das propriedades (*model check*), utilizados para realizar a análise com a ferramenta *Alloy Analyzer*. Esta ferramenta traduz uma especificação em uma fórmula booleana. A ferramenta pode ser usada para encontrar uma solução para uma especificação ou para verificar se alguma propriedade é válida para um escopo pré-definido. Um escopo define o número máximo de objetos permitidos para cada assinatura. A análise, em seguida, consiste na ligação de instâncias com as assinaturas e as relações, em busca de uma combinação de valores que compõem a fórmula booleana verdadeira traduzida.

As simulações realizadas pela ferramenta *Alloy Analyzer* são corretas e completas até um determinado escopo. Isto significa que, se a ferramenta não encontrar nenhuma solução, só é possível saber que a propriedade se manteve sobre este escopo. Não se pode concluir que as fórmulas declaradas na afirmação são válidas para um escopo maior já que a ferramenta não é um provador de teoremas, ou seja, de forma geral, a análise em Alloy é incompleta. No entanto, em alguns domínios específicos, é possível saber o número exato de instâncias envolvidas de cada assinatura. Nestas situações, é possível realizar uma análise completa. Como exemplo, no artigo Gheyi et al. (2006), eles propõe uma teoria de *Feature Models* em Alloy que foi utilizada para verificar um certo número de propriedades. Eles mostraram, por exemplo, que é possível verificar as propriedades gerais da relação de refatoração de *FM*, como reflexividade e transitividade.

2.5 PVS

O *Prototype Verification System (PVS)* [Owre et al. (2001)] fornece suporte mecanizado para especificação formal e verificação. Ele suporta uma ampla gama de atividades envolvidas na criação, análise, modificação, gerenciamento e documentação das teorias e provas. Entre outras coisas, o sistema *PVS* consiste de uma linguagem de especificação e de um provador de teoremas. A especificação consiste em um conjunto de teorias. Cada teoria pode introduzir axiomas, lemas, definições de tipos, constantes e teoremas. As especificações são fortemente tipadas, o que significa que cada expressão tem um tipo associado [Owre et al. (2001)].

Na Seção 2.5.1, são introduzidos conceitos de tipos e valores. A Seção 2.5.2, demonstra dois exemplos de especificação de fórmulas. A Seção 2.5.3, explica como as especificações em PVS são construídas a partir de teorias. A Seção 2.5.4, explica como um `datatype` é especificado. A Seção 2.5.5 explica definição indutiva. Por fim, na Seção 2.5.6, são fornecidos conceitos de provador de teoremas em PVS.

2.5.1 Tipos e Valores

A seguir é mostrado um exemplo de parte da LPPN, foi declarado um tipo *uninterpreted type* representando um identificador nomeado de `Id`. A única suposição feita em um tipo `Id` é que é disjunto de todos os outros tipos, exceto para os subtipos dele.

```
Id:  TYPE
```

Pode-se também especificar um *uninterpreted subtype*. Por exemplo,

```
idFO: TYPE FROM Id
```

onde `idFO` é um subtipo de `Id`. Não são feitas suposições sobre subtipos não interpretados, em particular, ele pode estar ou não estar vazio.

Além disto há o tipo *record type*, conforme a declaração de `FlowObject` que possui um identificador do tipo `idFO`.

```
FlowObject: TYPE = [# id: idFO #]
```

Record type impõe uma suposição de que está vazio, se qualquer de seus tipos de componentes está vazio, já que o tipo resultante é dado pelo produto cartesiano dos seus constituintes. O tipo `id` é declarado usando o tipo interpretado `idFO`.

Também há a declaração de predicados. A seguir temos o predicado `EQ_FO` que recebe dois parâmetros (`fo1` e `fo2` do tipo `FlowObject`) e retorna `bool`. Quando o `id` de `fo1` for igual ao `id` de `fo2` é retornado verdadeiro e caso contrário é retornado falso.

```
EQ_FO(fo1,fo2:  FlowObject):  bool =
  id(fo1) = id(fo2)
```

2.5.2 Fórmulas

Baseados em tipos e funções é possível declarar fórmulas. Declarações de fórmulas introduzem axiomas, suposições, teoremas e obrigações. Por exemplo, o fragmento a seguir

```
verificaIgualdade:  THEOREM
  ∀(f1, f2:  FlowObject):  id(f1) = id(f2) => EQ_FO(f1,f2)
```

temos a declaração do teorema `verificaIgualdade` que afirma que para quaisquer dois `FlowObject` se eles possuem o mesmo identificador, então eles satisfazem o predicado `EQ_FO`.

Um outro exemplo é de declaração de fórmula são as obrigações de prova. Ao verificar a especificação da declaração da função `fatorial` foi gerada uma terminação TCC (*type correctness condition*).

```
fatorial_TCC: OBLIGATION
  ∀(x:  nat):  (NOT x=0) IMPLIES (x - 1 < x)
```

Um certo número de TCCs é gerado em funções recursivas. Essas TCCs verificam se o valor da função de medição aplicada ao argumento recebido como um parâmetro *(x)* é maior do que o valor da função de medida aplicada ao parâmetro utilizado na chamada recursiva *(x-1)*, para garantir a terminação. O usuário deverá cumprir essas obrigações de prova com a ajuda do provador PVS. Uma TCC é gerada de modo a se certificar de que é um atribuído de valor válido. Às vezes, o provador de teoremas consegue provar automaticamente as TCCs.

Funções, tuplas e registros podem ser "modificados", por meio da expressão de substituição `WITH`. O resultado de uma expressão de substituição é uma função, tupla ou registro que é exatamente o mesmo que o original, exceto que os argumentos que especificados levam os novos valores. Por exemplo,

```
identity WITH [(0) := 1, (1) := 2]
```

é a mesma função que a função de identidade (definida no prelúdio), exceto para valores argumento 0 e 1. O prelúdio consiste em teorias pré-definidas que são construídas no sistema PVS. Por exemplo, a função `list2set` que está no prelúdio e compõe uma lista de elementos formando um conjunto, conforme especificação a seguir,

```
list2set[Transition]((:tt2,tt3:))
```

o resultado é um conjunto com dois elementos `tt2` e `tt3` do tipo `Transition`. O símbolo "(::)" é utilizado para expressões de coerção.

2.5.3 Teoria

Especificações em PVS são construídas a partir de teorias, que fornecem a reutilização e estruturação. Cada teoria pode declarar alguns tipos, constantes e fórmulas. Além disso, cada teoria pode importar outras teorias usando a cláusula de importação. Por padrão, todas as teorias importam a biblioteca de prelúdio, que prevê, entre outras coisas, os operadores booleanos, conjuntos, igualdade e os tipos de números reais, racionais, inteiros e naturais e suas propriedades associadas [Owre et al. (2001)].

```
LPPN: THEORY
    BEGIN
       IMPORTING ... ...
    END LPPN
```

2.5.4 Datatypes

Um *datatype* em PVS é especificado fornecendo um conjunto de construtores, juntamente com assessores e reconhecedores. Quando se verifica um *datatype* com relação a tipos, uma nova teoria é criada. Esta teoria fornece os axiomas e os princípios de indução necessários para assegurar que o *datatype* é a álgebra inicial definida pelos construtores [Owre et al. (2001)].

Por exemplo, suponha que se gostaria de especificar a sintaxe de comandos do programa de uma linguagem imperativa. Em seguida, declara-se o *datatype* de comando abstrato. Para simplificar, vamos apenas especificar dois tipos de comandos: skip e composição sequencial.

```
comando: DATATYPE
    skip: skip?: comando
    sequencial(primeiro:comando, segundo:comando):
               sequencial?: comando
    ...
END comando
```

Quando verificamos o tipo, o *uninterpreted type* `comando` é criado.

O *datatype* anterior declara dois construtores, `skip` e `sequencial`, que permitem a construção de `comando`. O termo `comando` no final de cada construtor é opcional e refere-se a um elemento que é um subtipo de `comando`.

Os reconhecedores `skip?` e `sequencial?` são predicados do *datatype* `comando` que são verdadeiros quando seus argumentos são construídos usando o construtor correspondente. Por exemplo, `sequencial?(c)` é verdadeiro quando `c` for do tipo `comando sequencial`.

Suponha que haja o `comando(c,skip)`, onde `c` é o comando. Os assessores `primeiro` e `segundo` podem ser usados para acessar os comandos `primeiro` e `segundo`.

2.5.5 Definição Indutiva

PVS fornece suporte para a construção definições indutivas. Definições indutivas são normalmente apresentadas para especificar regras para a geração de um conjunto de elementos e, em seguida, indicando que um objeto está no conjunto apenas se tiver sido gerado de acordo com as regras, portanto, é o menor conjunto fechado sob as regras.

A especificação de `even` é um exemplo simples de definição indutiva, conforme especificação a seguir.

```
even(n:nat): INDUCTIVE bool =
n = 0 OR (n > 1 AND even(n - 2))
```

A função `even` recebe um `n` do tipo `nat` (`n` é um número natural) e retorna `bool`. O retorno é verdadeiro quando `n` for zero ou quando for maior que 1 e `even(n -2)`, ou seja, quando for um número par.

2.5.6 Provador de Teoremas

Além da linguagem, o *PVS* possui um provador de teoremas que fornece um conjunto de procedimentos de inferência primitivas poderosas que são aplicadas de forma interativa sob a orientação do usuário dentro de um quadro de cálculo de sequentes. As inferências primitivas incluem: regras proposicionais e quantificadores, indução, reescrita, simplificação usando procedimentos de decisão (para a igualdade e aritmética linear, de dados e abstração de predicado) e verificação do modelo simbólico.

Um teorema nada mais é do que uma proposição formalmente dedutível (uma afirmação que pode ser provada), ou seja, a partir de certas proposições (premissas) é possível formar uma sequência de proposições que levam ao teorema criado, resultando na prova ou demonstração do teorema [Daghlian (1995)].

Uma forma de se calcular a validade de argumentos em lógica proposicional clássica é o chamado cálculo de sequentes, que foi proposto por Gentzen na década de trinta. O cálculo de sequentes de Gentzen constitui um estilo dedutivo. Mas em PVS a lógica não é clássica, é de ordem superior. Para provar que o cálculo de sequentes é consistente temos que mostrar que o sequente é válido. Ao se iniciar uma prova no PVS, aparece uma única fórmula sob uma linha tracejada que é um subsequente. Fórmulas acima das linhas tracejadas são chamadas de antecedentes e as abaixo são consequentes. A interpretação de um sequente é que a conjunção dos antecedentes implica a disjunção dos consequentes, um ou ambos os antecedentes e consequentes pode estar vazio. É fácil perceber que um sequente é verdadeiro se qualquer antecedente for o mesmo que qualquer consequente, se qualquer antecedente for falso e se qualquer consequente for verdadeiro. Cada prova em PVS começa com um consequente único [Owre et al. (1998)].

O objetivo básico da prova é gerar uma árvore de prova de sequentes em que todas as folhas são trivialmente verdadeiras. Os nós da árvore de prova são sequentes, e quando no provador foca-se em uma folha da árvore sem comprovação, obtém-se os atuais subsequentes. Quando um determinado ramo é completo (ou seja, termina em uma folha provada), a prova automaticamente passa para a próxima folha não comprovada, ou, se não houver mais ramos não comprovados, a prova está completa.

A árvore de prova auxilia na visualização da prova como um todo. Em cada nó da árvore existe um comando de prova já utilizado, e ao clicar no nó é possível visualizar quais as fórmulas existentes naquele momento da prova, o que facilita a visualização e correção da prova (como exemplo temos a Figura 4.10).

Capítulo 3

Teoria de Linha de Produtos de Processo de Negócio em Alloy

Este capítulo apresenta a especificação formal da LPPN em Alloy (a especificação completa está no Apêndice A), que busca garantir de forma escalável que a LPPN gera produtos bem-formados.

Para fins de legibilidade, neste trabalho utilizamos alguns símbolos de Alloy como símbolos matemáticos usuais. A estrutura geral da especificação é apresentada na Figura 3.1. As setas indicam as relações de dependência entre os módulos, onde cada um dos retângulos é um módulo em Alloy (grupo de definições associadas, por exemplo, funções, tipos de dados, teoremas). A especificação é dividida em quatro módulos:

- *Linguagem*: define a sintaxe e as regras de boa-formação da LPPN (Seção 3.1);
- *Exemplo*: possui exemplos concretos da LPPN, que exemplificam casos reais (Seção 3.1.3);
- *Transformações*: define as transformações sobre a LPPN (Seção 3.2);
- *Propriedades*: são afirmações que geram contraexemplos (por exemplo `WFM_SBP` - Section 3.2) sobre a boa-formação dos produtos resultantes de transformações da LPPN.

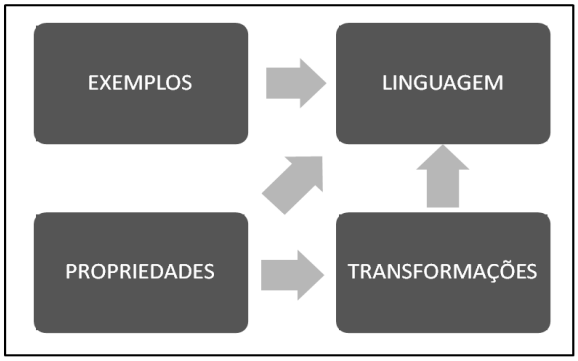

Figura 3.1: Estrutura global de especificação formal de LPPN

3.1 Linguagem Núcleo

Nesta seção será detalhada a linguagem núcleo especificada em Alloy. A linguagem núcleo é composta pela sintaxe e pelas regras de boa-formação da LPPN. A linguagem núcleo para LPPN inclui objetos do tipo início, atividade (que podem ter anotações) e fim, transições (que podem ter condições) e regras de boa-formação.

Neste trabalho, os processos de negócio foram tratados de forma linear e sem quantificação (só existe um ponto de junção) para simplificar o problema. A semântica não faz parte do escopo. Com esta simplificação conseguimos obter resultados mais simples, o que facilitou o entendimento dos exemplos e contraexemplos gerados pelo *model check*. Simplificações semelhantes são feitas em outros trabalhos [Apel et al. (2010)].

3.1.1 Sintaxe Abstrata

Na sintaxe abstrata temos os tipos especificados em Alloy para LPPN. A Figura 3.2, ilustra os tipos envolvidos. Nesta imagem, cada retângulo representa um tipo que em Alloy é tratado como uma assinatura (`sig`). As setas representam os relacionamentos entre os tipos, por exemplo: o tipo `BusinessProcessModel` possui um conjunto de processos do tipo `BusinessProcess`, ele representa a LPPN. A legenda no canto superior esquerdo mostra todos os tipos representados e a quantidade de tipos que são extensões de outras assinaturas, no caso temos quatorze assinaturas extendidas.

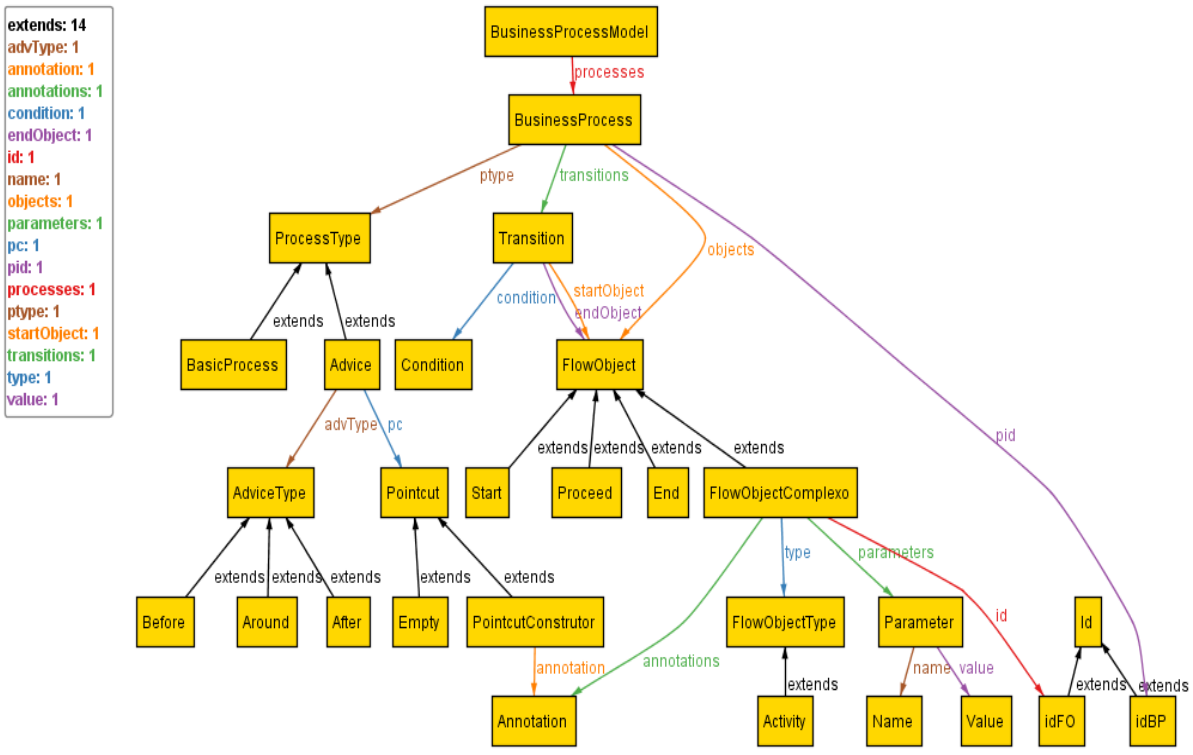

Figura 3.2: Modelo de objetos

O trecho da especificação abaixo é uma parte da declaração de assinaturas da Figura 3.2. O tipo `BusinessProcess` especificado no código abaixo possui quatro cons-

trutores: pid (que é do tipo idBP), ptype (que é do tipo ProcessType), objects (que é um conjunto de FlowObject) e transitions (que é um conjunto de transições do tipo Transition).

```
sig BusinessProcessModel{
    processes: set BusinessProcess
}
abstract sig ProcessType {}
one sig BasicProcess extends ProcessType{}
sig BusinessProcess {
    pid: idBP,
    ptype: ProcessType,
    objects: set FlowObject,
    transitions: set Transition
}
```

Uma Transition representa um conector do tipo fluxo de sequência. Ela é composta por três construtores, conforme especificação a seguir. Os construtores startObject e endObject são do tipo FlowObject e o construtor condition é do tipo Condition.

```
sig Transition {
    startObject: FlowObject,
    endObject: FlowObject,
    condition: Condition
}
```

Na especificação,

```
one sig Advice extends ProcessType {
    advType: AdviceType,
    pc: Pointcut
}
```

temos a assinatura Advice que é extensão do tipo ProcessType, ou seja, é um tipo de processo. Advice é constituído por um advType e um pc. O pc é do tipo Pointcut. Quando temos casamento de um Advice com um FlowObject temos um ponto de junção, que faz parte da abordagem composicional.

Um FlowObject representa uma atividade do tipo padrão, um evento de início do tipo padrão ou um evento de fim do tipo padrão de um processo de negócio.

3.1.2 Regras de boa-formação

As regras em Alloy foram definidas como predicados que compõem as relações entre as assinaturas. Primeiro, uma LPPN bem formada é descrita pelo predicado wellFormedModel, conforme especificação a seguir, que compreende um conjunto de processos bem-formados.

```
pred wellFormedModel[p:BusinessProcessModel] {
```

```
  ∀ bp:  p.processes | wellFormed[bp]
}
```

Um processo de negócio bem-formado é descrito pelo predicado `wellFormed` e satisfaz as seguintes regras:

```
pred wellFormed[p:  BusinessProcess]{
  WF1[p] ∧ WF2[p] ∧ WF3[p] ∧ WF4[p] ∧ WF5[p] ∧ WF6[p] ∧
  WF7[p] ∧ WF8[p] ∧ WF9[p]
}
```

Figura 3.3: Ilustração da regra WF1

A regra `WF1` define que todos os processos de negócio devem ter um início e um fim, ou seja, o `Start` e o `End` devem fazer parte do conjunto de objetos de um `BusinessProcess` (Figura 3.3).

```
pred WF1[p:  BusinessProcess]{
  Start+End in p.objects
}
```

A regra `WF2` define que o objeto `Start` não pode finalizar um processo de negócio, ou seja, o `Start` não pode ser o `endObject` de uma `Transition`. Na Figura 3.4, é ilustrada a intuição da regra `WF2`, onde o `Start` e o `End` são `FlowObjects` e o "X" representa que eles não devem estar nas posições de início, no caso do `End`, e fim, no caso do `Start`.

```
pred WF2[p:  BusinessProcess]{
  ∀ t:  p.transitions | t.endObject !in Start ∧
  t.endObject != none
}
```

Figura 3.4: Ilustração das regras WF2 e WF3

A regra `WF3` define que o objeto `End` não pode iniciar um processo de negócio, ou seja, não pode ser o `startObject` de uma `Transition` (Figura 3.4).

```
pred WF3[p:  BusinessProcess]{
  ∀ t:  p.transitions | t.startObject !in End ∧
  t.startObject != none
}
```

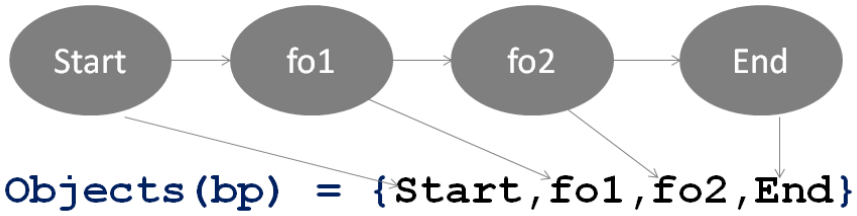

Figura 3.5: Ilustração da regra WF4

A regra `WF4` define que os `FlowObjects` de uma `Transition` de `p` devem fazer parte do conjunto de objetos do `BusinessProcess` (Figura 3.5).

```
pred WF4[p:  BusinessProcess]{
  p.transitions.(startObject+endObject) = p.objects
}
```

A regra `WF5` refere-se a propriedade de alcançabilidade a partir do objeto `Start`, onde qualquer objeto deve ser alcançável a partir do objeto `Start` (Figura 3.6).

```
pred WF5[p:  BusinessProcess]{
  Start in p.*transitions.startObject
}
```

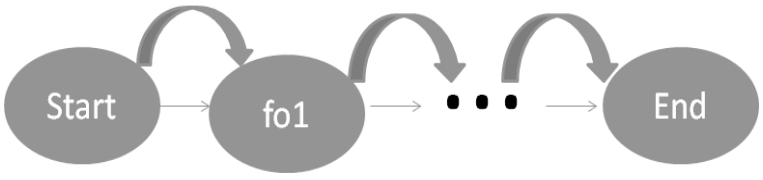

Figura 3.6: Ilustração das regras WF5 e WF6

A regra `WF6` é similar a regra `WF5`, sendo que esta estabelece que qualquer `FlowObject` deve alcançar o objeto `End` (Figura 3.6).

```
pred WF6[p:  BusinessProcess]{
  End in p.*transitions.endObject
}
```

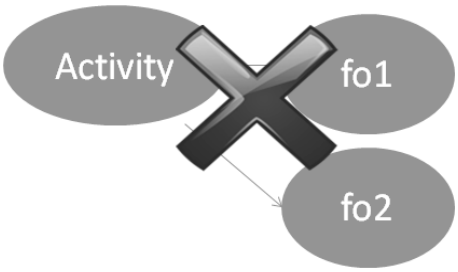

Figura 3.7: Ilustração da regra WF7

A regra `WF7` estabelece que não pode haver mais de uma transição saindo de um `FlowObject` do tipo `Activity`; se houver dois objetos saindo de uma transição, então eles são estruturalmente equivalentes (chamamos de `eq_tr`). Uma relação de equivalência estrutural de transições ocorre quando duas transições do tipo `Transition` possuem o mesmo `startObject`, o mesmo `endObject` e a mesma `condition` (ou seja, duas transições com as mesmas características) (Figura 3.7).

```
pred WF7[p:  BusinessProcess]{
  ∀ f:  p.objects | f.type = Activity ∧
                    f = FlowObjectComplexo ∧
  ∀ t1, t2:  p.transitions | t1.startObject = f ∧
                             t2.startObject = f
  ⇒ eq_tr[t1,t2]
}
```

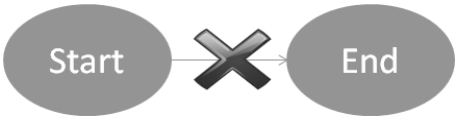

Figura 3.8: Ilustração da regra WF8

Já a regra `WF8` foi especificada para prevenir que haja uma transformação em um `Advice` vazio, que pode ser reduzido a nenhuma transformação. Esta regra também foi motivada para simplificar o esforço da verificação. Ela estabelece que não pode haver uma `Transition` do `Start` para o `End` quando o `BusinessProcess` for um `Advice` (Figura 3.8).

```
pred WF8[p:  BusinessProcess]{
  p.ptype = Advice ⇒
  ¬ (some t:  p.transitions | t.startObject = Start ∧
                              t.endObject = End)
}
```

A regra `WF9` é similar a regra `WF7`. Ela estabelece que um `FlowObject` do tipo `Start` deve ter apenas uma `Transition` saindo dele, caso haja duas então elas são equivalentes estruturalmente (Figura 3.9).

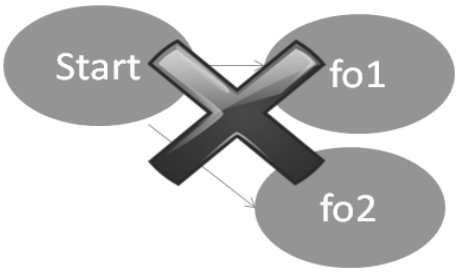

Figura 3.9: Ilustração da regra WF9

```
pred WF9[p:  BusinessProcess]{
  ∀ f:  p.objects | f = Start ∧
  ∀ t1, t2:  p.transitions | t1.startObject = f ∧
                             t2.startObject = f
  ⇒ eq_tr[t1,t2]
}
```

Os predicados em Alloy podem ser analisados através do *Alloy Analyzer* de forma automática pelo comando `run`. Ao utilizar este comando é possível gerar exemplos para o predicado especificado. Na especificação a seguir temos o exemplo de comando para o predicado `wellFormedModel`.

```
run wellFormedModel
```

Um exemplo da execução do comando `run` para o predicado `wellFormedModel` é ilustrado na Figura 3.10.

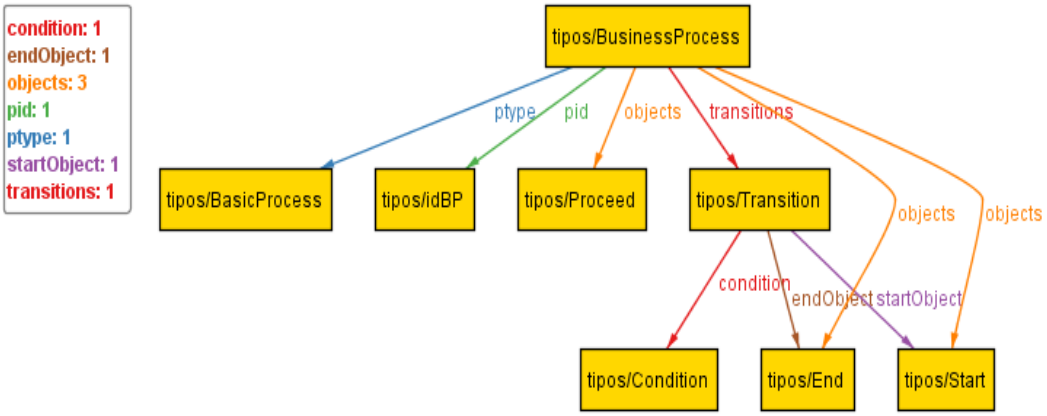

Figura 3.10: Resultado da execução do comando `run` para o predicado `wellFormedModel`

3.1.3 Exemplo

Esta seção possui um exemplo de LPPN. Estes são valores para os tipos definidos na sintaxe abstrata. O objetivo do exemplo é ilustrar a definição de linguagem e auxiliar na definição e verificação da especificação de propriedades específicas antes das gerais.

Para especificar um exemplo de LPPN na teoria criada neste trabalho em Alloy, deve-se criar um `BusinessProcessModel`, depois devem ser criados os processos que fazem parte do conjunto `processes` do tipo `BusinessProcess`.

O exemplo seguinte especifica um `BusinessProcessModel` chamado de `Modelo` que possui apenas um `BusinessProcess`, o processo de negócio Aposentadoria (Figura 3.15). A especificação

```
one sig Modelo in BusinessProcessModel {}
one sig Aposentadoria in BusinessProcess {}
one sig Analisar in FlowObjectComplexo {}
one sig transition0, transition1 in Transition {}
one sig id0 in idBP {}
one sig id1 in idFO {}
one sig condition0, condition1 in Condition {}
one sig annotation0, annotation1 in Annotation {}
one sig parameter0 in Parameter {}
one sig name0 in Name{}
one sig value0 in Value{}
```

define assinaturas específicas, ou seja, valores específicos para determinados tipos. Em seguida, foram definidos fatos (`fact`) que definem as relações entre os tipos. Na especificação a seguir, temos um fato que determina que o conjunto de processos do `Modelo` possui apenas um `BusinessProcess` que é o `Aposentadoria`.

```
fact{
    Modelo.processes = Aposentadoria
}
```

A seguir, temos o fato que especifica as relações entre os tipos que compõe o `BusinessProcess` `Aposentadoria`. Este processo é definido com o `ptype` `Advice` (o `Advice` possui dois `fields` o `advType` que é do tipo `After` e o `pc` que é do tipo `PointcutConstrutor`, sendo que este possui uma anotação `annotation1`). O identificador do processo é o `id0`. Seu conjunto de objetos é composto por três `FlowObjects`: `Start`, `End` e `Analisar` (este objeto refere-se a atividade "Analisar requisitos legais do processo"). O conjunto de transições possui duas transições: `transition0` e `transition1`.

```
fact{
    Aposentadoria.ptype = Advice
    Advice.advType = After
    Advice.pc = PointcutConstrutor
    PointcutConstrutor.annotation = annotation1
```

```
    Aposentadoria.pid = id0
    Aposentadoria.objects = Start+End+Analisar
    Aposentadoria.transitions = transition0+transition1
}
```

A especificação a seguir define as relações de tipos para o `FlowObject Analisar`, que possui o identificador `id1`, a anotação `annotation0`, o tipo `Activity` e o parâmetro `parameter0`. Sendo que o `parameter0` possui o `name name0` e o `value value0`.

```
fact{
    Analisar.type = Activity
    Analisar.id = id1
    Analisar.annotations = annotation0
    parameter0.name = name0
    parameter0.value = value0
    Analisar.parameters = parameter0
}
```

As transições são especificadas a seguir sendo que a `transition0` possui a condição `condition0`, o `startObject Start` e o `endObject Analisar`. Já a `transition1` possui a condição `condition1`, o `startObject Analisar` e o `endObject End`.

```
fact{
    transition0.condition = condition0
    transition1.condition = condition1
    transition0.startObject = Start
    transition0.endObject = Analisar
    transition1.startObject = Analisar
    transition1.endObject = End
}
```

Após a especificação do `BusinessProcessModel Modelo` utilizou-se o predicado `show`

```
pred show[]{}
run show for 8
```

para gerar automaticamente o exemplo. Este resultado do comando `run` é importante para verificarmos se os tipos especificados estão de acordo com os processos de negócio, ou seja, para identificar falhas na especificação. O resultado é um `BusinessProcessModel` ilustrado na Figura 3.11.

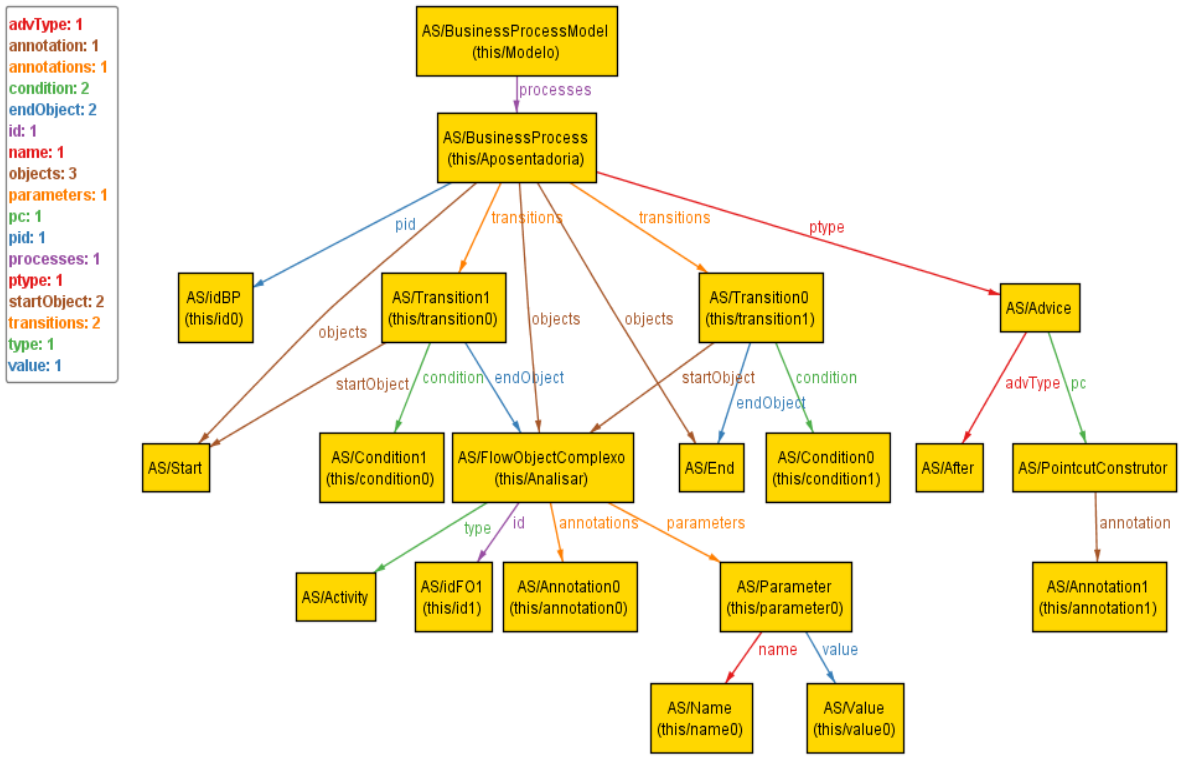

Figura 3.11: Resultado da execução do comando `run`

3.2 Transformações, Propriedades e Verificações

Nesta seção, tratamos sobre as transformações que fazem a composição dos produtos da LPPN, verificamos propriedades e, também, o resultado das transformações.

As transformações são aplicadas a fim de gerenciar a variabilidade em LPPN: selecionando e compondo processos de negócio, alguns dos quais são `Advices` (processos de negócio específicos são construídos de acordo com a função `Weaving Process` - Seção 2.3). Na abordagem composicional investigada, formalizamos duas transformações descritas a seguir.

3.2.1 Transformação SelectBusinessProcess

Para especificar transformações em Alloy foram utilizadas funções (`fun`). Basicamente, coloca-se a palavra `fun`, após o nome da função. Em seguida, devem ser descritos quais parâmetros que vão ser recebidos, que são opcionais, entre colchetes, colocando primeiramente o nome do parâmetro e após a assinatura do parâmetro. Por fim, coloca-se o tipo (no caso do Alloy, a assinatura) que será retornado pela função.

Na transformação `SelectBusinessProcess` é selecionado um `BusinessProcess` de uma LPPN conforme o identificador de um `BusinessProcess`, o tipo `idBP`. A Figura 3.12 ilustra um modelo `SPL` onde é selecionado o `BusinessProcess` com o `idBP` igual a `Id0`. Após este processo ser selecionado ele é copiado para a saída da transformação, o modelo `Produto`.

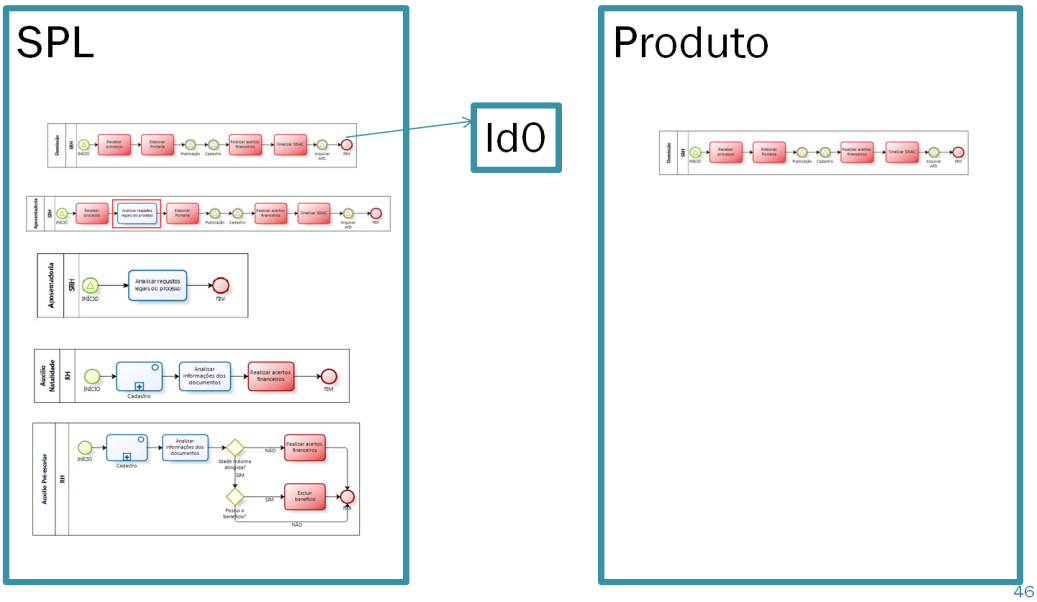

Figura 3.12: Exemplo da transformação *SelectBusinessProcess*

A transformação recebe dois parâmetros (`Id` e `BusinessProcessModel`) e retorna um `BusinessProcessModel` com apenas um `BusinessProcess`, sendo que a transformação exige que:

- o conjunto de processos resultante do `BusinessProcessModel` deve possuir apenas um elemento;
- o `BusinessProcess` deve possuir o mesmo `Id` que o `bpId` (parâmetro de entrada);
- o `BusinessProcess` com o mesmo `Id` que o `bpId` deve ser equivalente estruturalmente ao processo que está sendo selecionado (este passo foi necessário para gerar uma cópia do processo para o modelo que é retornado pela transformação);
- a LPPN que é parâmetro de entrada deve ser bem formada.

A especificação abaixo refere-se à função `SelectBusinessProcess` descrita acima.

```
fun SelectBusinessProcess[bpId:  Id,
                          spl:  BusinessProcessModel]:
 BusinessProcessModel{
  {
   bps:BusinessProcessModel | #bps.processes = 1 ∧
   eq_bp[bps.processes,pid.bpId & spl.processes] ∧
   wellFormedModel[spl]
  }
 }
```

Em Alloy, é possível fazer verificações automáticas(`check`). As verificações permitem verificar propriedades e assinaturas que foram especificadas. Além disto, elas permitem

encontrar automaticamente falhas dentro de um determinado escopo e auxiliam na criação de novos predicados. Um exemplo de criação de novo predicado devido a uma verificação realizada foi a criação da regra de boa formação `WF4`. Ao executar uma verificação foi encontrado um contraexemplo que dizia que um processo de negócio poderia ter objetos do tipo `Flowobject` que não fazem parte do conjunto de objetos de um `BusinessProcess`.

É realizada uma afirmação (`assert`) e, em seguida, é feita a verificação desta afirmação com o comando `check`. Na especificação

```
assert Unicidade_SBP{
    all i:idBP, spl: BusinessProcessModel |
                     one SelectBusinessProcess[i,spl].processes
}
check Unicidade_SBP for 6
```

há um exemplo de um `check` para verificar se a transformação `SelectBusinessProcess` retorna um `BusinessProcessModel` com um único `BusinessProcess` (um `BusinessProcessModel` é um modelo que contém um ou mais `BusinessProcess`). Na execução do `check` podem ser encontrados ou não contraexemplos, são eles que auxiliam na procura por falhas. Quando não há um contraexemplo, significa que a propriedade testada pode ser correta, mas não é sempre válida pois varia conforme o escopo. Quando há um contraexemplo, sabe-se com certeza que a propriedade não está correta, facilitando a visualização das falhas.

A afirmação `Unicidade_SBP` gerou contraexemplos nas primeiras verificações. Esta falha foi devido à especificação incorreta de tipos. Antes de chegarmos a especificação de tipos final, tivemos diversas alterações, conforme as falhas encontradas, com auxílio dos contraexemplos.

Por exemplo, uma das falhas encontrada na especificação com esta verificação foi para o tipo `id`. Inicialmente, não tínhamos especificado o `idBP` e o `idFO`, mas apenas `id`. O que acontecia era que a função `SelectBusinessProcess`, que recebe um `id` como parâmetro de entrada, poderia gerar um conjunto vazio na sua saída se recebesse um `id` que não identificasse um `BusinessProcess` e sim um `FlowObject`. Isto fez com que criássemos identificadores distintos para `BusinessProcess` e para `FlowObject`(o `idBP` e o `idFO`).

A afirmação a seguir

```
assert WFM_SBP{
 ∀ i:  idBP | ∀ spl:  BusinessProcessModel |
 wellFormedModel[spl] ∧ i in spl.processes.pid
  ⇒ wellFormedModel[SelectBusinessProcess[i,spl]]
}
check WFM_SBP
```

diz que, para todo `i` do tipo `idBP` e para todo `spl` do tipo `BusinessProcessModel`, se `spl` é bem-formado e `i` é um `id` de um `BusinessProcess` que faz parte do conjunto de processos do modelo `spl`, então o resultado da transformação `SelectBusinessProcess(i,spl)` é um modelo bem-formado. Nesta afirmação não

foi gerado contraexemplos, notou-se que ao aumentar o escopo o tempo de execução aumentou significativamente. Assim, existe possibilidade de ser verdadeiro.

3.2.2 Transformação EvaluateAfterAdvice

A outra transformação especificada foi a `evaluateAfterAdvice`. Esta transformação faz a composição de um `BusinessProcess` do tipo `Advice` com um outro `BusinessProcess` para formar um terceiro `BusinessProcess`.

Por exemplo, comparando as Figuras 3.13 e 3.14 é possível notar que o único `FlowObject` que varia é "Analisar requisitos legais do processo" que está contornado em vermelho no processo "Aposentadoria"; o restante dos `FlowObjects` são comuns a ambos os processos.

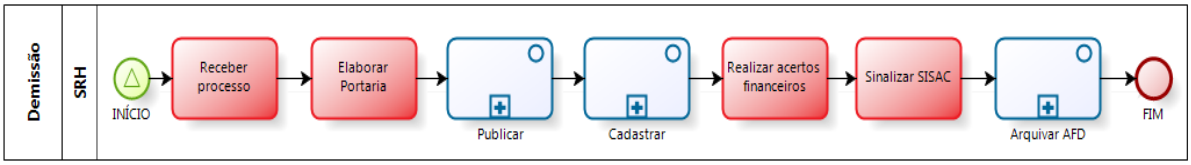

Figura 3.13: Processo Demissão (Lei 8.112/90)

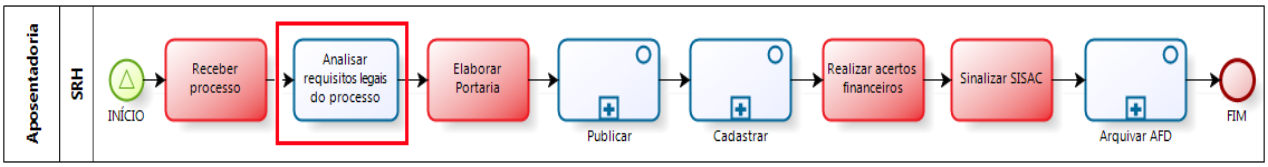

Figura 3.14: Processo Aposentadoria (Lei 8.112/90)

Tratando esta variabilidade de forma composicional, na Figura 3.15, foi criado um processo do tipo `Advice` que possui apenas a parte que varia entre os dois processos citados anteriormente.

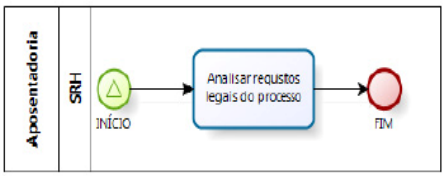

Figura 3.15: Exemplo de *Advice*

Assim, para formar o processo "Aposentadoria", a transformação `evaluateAfterAdvice` recebe o processo do tipo `Advice` e o processo "Demissão" e, via composição de aspectos, cria um terceiro processo que é o processo "Aposentadoria". Três `transitions` são retiradas e são criadas duas novas `transitions` que ligam um processo ao outro, conforme ilustrado na Figura 3.16.

Em `Alloy`, a transformação `evaluateAfterAdvice` está especificada conforme a função a seguir.

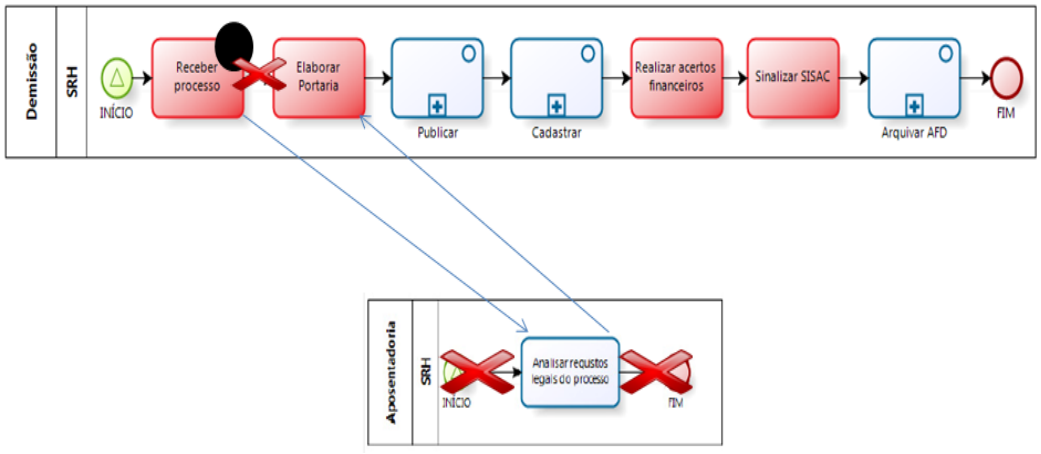

Figura 3.16: Exemplo da transformação *evaluateAfterAdvice*

```
fun evaluateAfterAdvice[adv, bp:  BusinessProcess]:
 BusinessProcess{
  {
   p:BusinessProcess | p.pid = bp.pid ∧ p.ptype = bp.ptype ∧
   p.objects = {bp.objects} ++
   {b:adv.objects | b != End ∧ b != Start}
   ∧ p.transitions = montaT(adv,bp)
  }
 }
```

A transformação `evaluateAfterAdvice` recebe dois `BusinessProcess` como parâmetros de entrada e retorna um `BusinessProcess` que possui o mesmo identificador e o mesmo tipo do parâmetro `bp`. O conjunto de objetos do `BusinessProcess` resultante é composto pela união de todos os objetos de `bp` com os objetos de `adv` sem o `Start` e sem o `End` (o símbolo `++` em Alloy representa um operador relacional, ele representa a união de conjuntos, se houver a repetição de elementos entre os conjuntos que estão sendo unidos esta repetição será descartada). O conjunto de transições do `BusinessProcess` resultante da transformação `evaluateAfterAdvice` é composto pela função `montaT`.

A função `montaT` possui quatro partes que definem o conjunto de transições do `BusinessProcess` resultante do `evaluateAfterAdvice`. A primeira parte é composta por transições que fazem parte de `bp`. A segunda parte é constituída por transições que fazem parte do `adv`. A terceira parte possui uma transição que liga o `FlowObject` anotado de `bp`, ao primeiro `FlowObject` de `adv`, diferente do `Start`. E, a quarta parte, é composta por uma transição que liga o último `FlowObject`, diferente do `End`, do `adv` até o `FlowObject` seguinte ao `FlowObject` anotado em `bp`.

A verificação da transformação `evaluateAfterAdvice` foi através da propriedade de que o `BusinessProcess` retornado da transformação `evaluateAfterAdvice` deve estar de acordo com as regras de boa-formação. Esta verificação está especificada a seguir.

```
assert WFM_EVAL{
 ∀ adv, bp:BusinessProcess |
 ∀ BPM : BusinessProcessModel |
      adv.ptype in Advice ∧
      adv in BPM.processes ∧
      bp in BPM.processes ∧
      wellFormedModel[BPM] ∧
      adv.objects not in bp.objects ∧
      evaluateAfterAdvice[adv,bp] != none ⇒
 wellFormed[evaluateAfterAdvice[adv,bp]]
}
check WFM_EVAL
```

A afirmação `WFM_EVAL` especifica que para quaisquer `BusinessProcess` `adv` e `bp`, e para todo modelo `BPM`, se `adv` do tipo `Advice`, `adv` e `bp` fazem parte do conjunto de processos de `BPM`. `BPM` é um modelo bem-formado, o conjunto de objetos de `adv` é disjunto do conjunto de objetos de `bp`, e o retorno da transformação `evaluateAfterAdvice[adv,bp]` é diferente de vazio, isto implica que o resultado de `evaluateAfterAdvice[adv,bp]` é bem-formado. Nesta afirmação não foi gerado contraexemplos, considerando um escopo para dez instâncias para cada assinatura. Assim, existe possibilidade de ser verdadeiro.

Capítulo 4

Teoria de Linha de Produtos de Processo de Negócio em PVS

Este capítulo apresenta a especificação e a verificação formais de LPPN em PVS (a especificação completa pode ser encontrada no Apêndice B). A formalização é baseada na especificação feita anteriormente em Alloy (Capítulo 3). Decidiu-se realizar a formalização em PVS porque este contém uma linguagem de especificação suficientemente expressiva para o propósito presente neste trabalho.

Para fins de legibilidade, neste trabalho, utilizamos alguns símbolos de PVS com símbolos matemáticos usuais. A estrutura geral da especificação é apresentada na Figura 3.1, de forma análoga a especificação em Alloy. Em particular, as contribuições são:

- **Especificação formal** (Section 4.1): a formalização no PVS especifica precisamente uma linguagem de LPPN (Seção 4.1.1), propriedades de boa-formação (Seção 4.1.2) e composição de transformações (Seção 4.2). Ela se concentra em como essas transformações contribuem para boa-formação.
- **Verificação geral** (Seção 4.2): a verificação é formal e garante a boa-formação de todos os produtos sem verificar cada um individualmente. É composta de provas de que as transformações representadas preservam a boa-formação em qualquer LPPN.

4.1 Linguagem Núcleo

Na linguagem especificada em PVS, de forma análoga ao Capítulo 3 os processos são tratados de forma linear. Atualmente, a linguagem não aborda outros processos de negócio construídos com decisões, *swimlanes* e eventos. No entanto, uma abordagem similar pode ser aplicada para tratar estes processos e é considerada como trabalho futuro. Esta linguagem simplificada de BPMN facilitou a prova dos teoremas, pois diminuiu o tempo gasto com as provas devido à quantidade de tipos especificados que foi reduzida. Notamos que a LPPN neste trabalho está limitada a boa-formação de propriedades.

4.1.1 Sintaxe Abstrata

A sintaxe da LPPN consiste de tipos de dados algébricos hierarquicamente relacionados que definem conceitos chave no domínio LPPN como modelo de processos de negócios

e processos de negócio. Traduzimos a implementação em Alloy para PVS mapeando-se construções correspondentes entre as linguagens: as assinaturas em Alloy foram mapeadas em PVS como datatype; as afirmações em Alloy (`assert`) foram mapeados como teoremas em PVS; as transformações foram mapeadas com funções tanto em Alloy como PVS.

Abaixo está especificado em PVS o tipo `BusinessProcessModel`, o tipo raiz da sintaxe abstrata; ele consiste em um conjunto de processos do tipo `BusinessProcess`. `BPM` é um tipo construtor e `BusinessProcessModel?` um predicado que reconhece valores do tipo `BPM`.

```
BusinessProcessModel: DATATYPE
 BEGIN
  BPM(processes: set[BusinessProcess]): BusinessProcessModel?
 END BusinessProcessModel
```

`BusinessProcess` é um tipo com quatro campos (um identificador (`pid`), um tipo (`ptype`), um conjunto de objetos (`objects`) e um conjunto de transições (`transitions`)), cada um dos quais é modelado como um datatype em PVS. Conforme especificação a seguir.

```
BusinessProcess: DATATYPE
 BEGIN
  BusinessProcess(pid: idBP, ptype: ProcessType,
                  objects: set[FlowObject],
                  transitions: set[Transition]): BusinessProcess?
 END BusinessProcess
```

O `ptype` de um `BusinessProcess` é um `ProcessType` que possui dois predicados `BasicProcess?` (que reconhece valores do tipo `BasicProcess`) e `Advice` (que reconhece valores do tipo `Advice`).

```
ProcessType: DATATYPE
 BEGIN
  BasicProcess: BasicProcess?
  Advice(advType: AdviceType, pc: Pointcut): Advice?
 END ProcessType
```

O conjunto `transitions` de um `BusinessProcess` é composto por transições do tipo `Transition`. Cada `Transition` é um `datatype` que possui um predicado `MkTransition?` que reconhece valores do tipo `MkTransition`. O valor `MkTransition` possui três campos: `startObject` e `endObject` do tipo `FlowObject`; e, `condition` do tipo `Condition`.

```
Transition: DATATYPE
 BEGIN
```

```
    MkTransition(startObject: FlowObject, endObject: FlowObject,
                 condition: Condition): MkTransition?
  END Transition
```

4.1.2 Regras de boa-formação

As regras de boa-formação especificadas em PVS foram baseadas nas especificações da Seção 3.1.2 e são explicadas a seguir. Primeiro, uma LPPN bem-formada, descrita pelo predicado `wellFormedModel`, compreende um conjunto de processos de negócio bem-formados:

```
wellFormedModel(p: BusinessProcessModel): bool =
  ∀ (bp: {p_1: BusinessProcess | (p_1 ∈ processes(p))}): wellFormed(bp)
```

Um produto bem-formado é definido pelo predicado `wf` e satisfaz as seguintes regras:

```
wellFormed(p: BusinessProcess): bool =
  WF1(p) ∧ WF2(p) ∧ WF3(p) ∧ WF4(p) ∧ WF5(p) ∧ WF6(p) ∧
  WF7(p) ∧ WF8(p) ∧ WF9(p)
```

A seguir, ilustramos todas as regras de boa-formação de um processo de negócio. A regra `WF1` define que todo processo deve ter um início e um fim, ou seja, os objetos `Start` e `End` devem fazer parte do conjunto de objetos de `BusinessProcess`.

```
WF1(p: BusinessProcess): bool =
  (Start0 ∈ objects(p)) ∧ (End0 ∈ objects(p))
```

Qualquer `FlowObject` de `bp` deve ser alcançável a partir do `Start` é definido pela regra `WF5`. Para formalizar esta regra, primeiro definimos o predicado `extends`, que relaciona dois objetos iguais ou compreende uma transição no processo de negócio:

```
extends(bp: BusinessProcess, origem: FlowObject)
       (fim: FlowObject): bool =
  (origem ∈ (objects(bp))) ∧ (fim ∈ (objects(bp))) ∧
  ((origem = fim) ∨ (∃ (t: Transition): (t ∈ transitions(bp)) ∧
  startObject(t) = origem ∧ endObject(t) = fim))
```

Em seguida, consideramos o fecho transitivo reflexivo da relação `extends`. Em PVS, a definição indutiva `extendsClosure` especifica que há uma relação direta `extends` entre dois objetos ou há um objeto intermediário entre eles. O objeto intermediário deve ser alcançável a partir da origem e deve estar diretamente relacionado com o `extends`:

```
extendsClosure(bp: BusinessProcess, origem: FlowObject)
              (fim: FlowObject): INDUCTIVE boolean =
  ((origem ∈ (objects(bp))) ∧ (fim ∈ (objects(bp)))) ∧
```

((extends(bp, origem)(fim)) ∨
(∃ (meio: FlowObject): (meio ∈ objects(bp)) ∧
extendsClosure(bp, origem)(meio) ∧
extends(bp, meio)(fim)))

Para formalizar a regra `WF5`, contamos com `extendsClosure` da seguinte forma:

WF5(p: BusinessProcess): bool =
(∀ (fo: FlowObject): (fo ∈ objects(p)) ⇒ extendsClosure(p, Start0)(fo))

Outras regras de boa-formação referem-se a propriedades dos objetos `Start` e `End`, como: o `Start` não pode finalizar uma transição, o `End` não pode inicializar uma transição, o `End` é alcançável a partir de qualquer objeto, e propriedades de transições como transições de um processo de negócio compreendem objetos dentro do processo e só há uma transição a partir de um determinado objeto. As regras são basicamente as mesmas regras especificadas em Alloy, a seguir estão as especificações das regras `WF2`, `WF3`, `WF4`, `WF6`, `WF7`, `WF8` e `WF9`.

WF2(p: BusinessProcess): bool =
(∀ (t: Transition): (t ∈ transitions(p)) ⇒ endObject(t) ≠ Start0)

WF3(p: BusinessProcess): bool =
(∀ (t: Transition): (t ∈ transitions(p)) ⇒ startObject(t) ≠ End0)

WF4(p: BusinessProcess): bool =
(∀ (t: Transition): (t ∈ transitions(p)) ⇒ (startObject(t) ∈ objects(p)) ∧
(endObject(t) ∈ objects(p)))

WF6(p: BusinessProcess): bool =
(∀ (fo: FlowObject): (fo ∈ objects(p)) ⇒ extendsClosure(p, fo)(End0))

WF7(p: BusinessProcess): bool =
(∀ (fo: FlowObject): (fo ∈ objects(p)) ∧ CFO?(fo) ∧
type0(fo) = Activity ⇒
(∀ (tt1, tt2: Transition): (tt1 ∈ transitions(p)) ∧ (tt2 ∈ transitions(p)) ∧
startObject(tt1) = fo ∧ startObject(tt2) = fo
⇒ EQ_TR(tt1, tt2)))

Uma relação de equivalência estrutural de transições (`EQ_TR`) ocorre quando duas transições do tipo `Transition` possuem o mesmo `startObject`, o mesmo `endObject` e a mesma `condition` (ou seja, duas transições com as mesmas características).

WF8(p: BusinessProcess): bool =
Advice?(ptype(p)) $\Rightarrow$ $\neg$ extends(p, Start0)(End0)

WF9(p: BusinessProcess): bool =
($\forall$ (fo: FlowObject): (fo $\in$ objects(p)) $\wedge$ Start0?(fo) $\Rightarrow$
($\forall$ (tt1, tt2: Transition): (tt1 $\in$ transitions(p)) $\wedge$
(tt2 $\in$ transitions(p)) $\wedge$ startObject(tt1) = fo $\wedge$
startObject(tt2) = fo $\Rightarrow$ EQ_TR(tt1, tt2)))

4.1.3 Exemplo

A seguir um exemplo de LPPN, nomeado de `M`, compreendendo um único processo (Figura 3.15). Uma vez que o escopo de nomes em PVS é válido a partir do ponto de definição para a especificação, a definição tem um estilo construtivo.

Os objetos de fluxo do processo são `Start`, "*Analisar os requisitos legais do processo*" (nomeado `C`) e `End`. `Start` e `End` são objetos únicos. `C` é um tipo de `FlowObject` e tem quatro campos: um identificador (`id1`), um tipo (`Activity`), um conjunto de anotações (`a1`) e um conjunto de parâmetros (`p1`):

```
id1:  idFO
a1:  Annotation
p1:  Parameter
C: FlowObject = CFO(id1,Activity,a1,p1)
```

As setas são chamadas de transições. Uma transição conecta dois `FlowObject`. A primeira transição é representada por `t1`, ela conecta o `Start` ao `C`. Esta transição tem a condição `c1`. A segunda transição (segunda seta) é `t2`, ela conecta o objeto `C` ao `End` e possui a condição `c2`:

```
c1,c2: Condition
t1: Transition = MkTransition(Start,C,c1)
t2: Transition = MkTransition(C,End,c2)
```

O processo de negócio "*Aposentadoria*" (`bp1`) é composto por um identificador (`id`), um tipo (`Advice`), um conjunto de objetos (`objs`) e um conjunto de transições (`ts`). `MkTransition` é o único construtor de uma `Transition`.

```
id:  idBP
objs = {Start, C, End}
ts = {t1, t2}
bp1:  BusinessProcess = BusinessProcess(id,Advice,objs,ts)
```

Finalmente, esta simples LPPN compreende um valor de tipo `BusinessProcessModel`, neste caso um conjunto de processos.

```
M: BusinessProcessModel = BPM({bp1,bp2})
```

De maneira geral, para especificar um exemplo de LPPN na teoria criada neste trabalho em PVS, deve-se criar um `BusinessProcessModel`, depois devem ser criados os processos que fazem parte do conjunto `processes` do `BusinessProcessModel` criado do tipo `BusinessProcess`, sendo que cada `BusinessProcess` deve possuir quatro campos, conforme a especificação do `datatype BusinessProcess`.

4.2 Transformações, Propriedades e Verificações

No processo de derivação composicional de produtos de LPPN, transformações desempenham um papel central através da composição de processos de negócios, alguns dos quais são *advices*, para construir um produto para qualquer configuração selecionada. Uma condição suficiente para boa-formação do produto resultante é garantir que cada transformação conserva a boa-formação.

Neste trabalho, abordamos três transformações, duas iguais às especificadas em Alloy (Capítulo 3) e mais uma que se mostrou útil para especificação e verificação da transformação `evaluateAfterAdvice` que é mais complexa.

4.2.1 Transformação SelectBusinessProcess

De forma análoga a Seção 3.2, `SelectBusinessProcess` simplesmente seleciona um `BusinessProcessModel`, identificado por um id, a partir dos processos de negócio dentro da LPPN. A transformação possui dois parâmetros (`bpId` e `pl`) e retorna um `BusinessProcessModel` com somente um processo de negócio. Além disso, a transformação requer que o `BusinessProcessModel` retornado tenha o mesmo `Id` que `bpId` e que a LPPN, que é um parâmetro de entrada (`pl`), seja bem-formada, como descrito a seguir.

SelectBusinessProcess
  (spl: {sp: BusinessProcessModel | wellFormedModel(sp)},
 bpId: {id: Id | ∃ (p: BusinessProcess):
                 (p ∈ processes(spl)) ∧ pid(p) = id}):
 BusinessProcessModel =

A transformação `SelectBusinessProcess` retorna um `BusinessProcessModel` compreendendo um processo `pl` e com o mesmo identificador como `bpId`.

spl WITH [processes := {bp: BusinessProcess |
                        (bp ∈ processes(spl)) ∧
                        pid(bp) = bpId}]

O teorema `WFM_SBP` afirma que a transformação `SelectBusinessProcess` preserva a boa-formação, ou seja, `SelectBusinessProcess(i,pl)` sempre gera uma LPPN

bem-formada, uma vez que `pl` é bem-formado e `i` refere-se ao processo de negócio em `pl`:

```
WFM_SBP: THEOREM
 ∀ (spl: {sp: BusinessProcessModel | wellFormedModel(sp)},
   bpId: {id: Id | ∃ (p: BusinessProcess): (p ∈ processes(spl)) ∧
                                            pid(p) = id}):
     wellFormedModel(SelectBusinessProcess(spl, bpId))
```

Prova

A estratégia utilizada para provar[1] o teorema `WFM_SBP` foi primeiro realizar a skolemização sobre o quantificador universal do primeiro sequente (este sequente é composto por uma única fórmula que é o teorema em si, conforme Figura 4.1) chegando ao sequente da Figura 4.2.

```
WFM_SBP :

  |-------
{1}   FORALL (spl: {sp: BusinessProcessModel | wellFormedModel(sp)},
              bpId:
                {id: Id |
                         EXISTS (p: BusinessProcess):
                           member(p, processes(spl)) AND pid(p) = id}):
        wellFormedModel(SelectBusinessProcess(spl, bpId))
```

Figura 4.1: Primeiro sequente da prova do teorema WFM_SBP

```
WFM_SBP :

{-1}  wellFormedModel(spl!1)
{-2}  EXISTS (p: BusinessProcess):
        member(p, processes(spl!1)) AND pid(p) = bpId!1
  |-------
{1}   wellFormedModel(SelectBusinessProcess(spl!1, bpId!1))
```

Figura 4.2: Segundo sequente da prova do teorema WFM_SBP

Após isso, skolemizamos o quantificador existencial chegando ao sequente da Figura 4.3.

```
WFM_SBP :

[-1]  wellFormedModel(spl!1)
{-2}  member(p!1, processes(spl!1)) AND pid(p!1) = bpId!1
  |-------
[1]   wellFormedModel(SelectBusinessProcess(spl!1, bpId!1))
```

Figura 4.3: Terceiro sequente da prova do teorema WFM_SBP

Em seguida, expandimos a definição `wellFormedModel` o que resultou no sequente da Figura 4.4.

[1] As provas podem ser encontradas no site http://www.gisellemachado.com/mestrado/pvs/arquivos/proof.zip

```
WFM_SBP :

{-1}  FORALL (bp: {p1: BusinessProcess | member(p1, processes(spl!1))}):
        wellFormed(bp)
[-2]  member(p!1, processes(spl!1)) AND pid(p!1) = bpId!1
  |-------
{1}   FORALL (bp:
                {p1: BusinessProcess |
                        member(p1,
                               processes(SelectBusinessProcess
                                         (spl!1, bpId!1))))}):
        wellFormed(bp)
```

Figura 4.4: Quarto sequente da prova do teorema WFM_SBP

Skolemizamos a fórmula `{1}` o que resultou no sequente da Figura 4.5.

```
WFM_SBP :

{-1}  member(bp!1, processes(SelectBusinessProcess(spl!1, bpId!1)))
[-2]  FORALL (bp: {p1: BusinessProcess | member(p1, processes(spl!1))}):
        wellFormed(bp)
[-3]  member(p!1, processes(spl!1)) AND pid(p!1) = bpId!1
  |-------
{1}   wellFormed(bp!1)
```

Figura 4.5: Quinto sequente da prova do teorema WFM_SBP

Expandimos a definição `SelectBusinessProcess`, resultando no sequente da Figura 4.6.

```
WFM_SBP :

{-1}  member(bp!1,
             {bp: BusinessProcess |
               member(bp, processes(spl!1)) AND pid(bp) = bpId!1})
[-2]  FORALL (bp: {p1: BusinessProcess | member(p1, processes(spl!1))}):
        wellFormed(bp)
[-3]  member(p!1, processes(spl!1)) AND pid(p!1) = bpId!1
  |-------
[1]   wellFormed(bp!1)
```

Figura 4.6: Sexto sequente da prova do teorema WFM_SBP

Expandimos a definição de `member` resultando no sequente da Figura 4.7.

```
WFM_SBP :

{-1}  processes(spl!1)(bp!1) AND pid(bp!1) = bpId!1
[-2]  FORALL (bp: {p1: BusinessProcess | member(p1, processes(spl!1))}):
        wellFormed(bp)
{-3}  processes(spl!1)(p!1) AND pid(p!1) = bpId!1
  |-------
[1]   wellFormed(bp!1)
```

Figura 4.7: Sétimo sequente da prova do teorema WFM_SBP

Em seguida, instanciamos a fórmula -2 com a variável `bp!1`, o que resultou em dois subobjetivos (Figuras 4.8 e 4.9).

Os subobjetivos possuem condições que estão no antecedente e no consequente, o que prova os dois subobjetivos e o teorema `WFM_SBP`.

```
WFM_SBP.1 :

[-1]  processes(spl!1)(bp!1) AND pid(bp!1) = bpId!1
{-2}  wellFormed(bp!1)
[-3]  processes(spl!1)(p!1) AND pid(p!1) = bpId!1
  |-------
[1]   wellFormed(bp!1)
```

Figura 4.8: Primeiro subobjetivo da prova do teorema WFM_SBP

```
WFM_SBP.2 (TCC):

[-1]  processes(spl!1)(bp!1) AND pid(bp!1) = bpId!1
[-2]  processes(spl!1)(p!1) AND pid(p!1) = bpId!1
  |-------
{1}   member[BusinessProcess](bp!1, processes(spl!1))
[2]   wellFormed(bp!1)
```

Figura 4.9: Segundo subobjetivo da prova do teorema WFM_SBP

Na árvore de prova, ilustrada na Figura 4.10, é possível visualizar os comandos de provas utilizados (por exemplo, `skolem-typepred`) e os sequentes (cada nó da árvore é um sequente). O primeiro sequente (Figura 4.1) é o nó raiz. Do primeiro para o segundo sequente (Figura 4.2) chegou-se utilizando o comando de prova `skolem-typepred`.

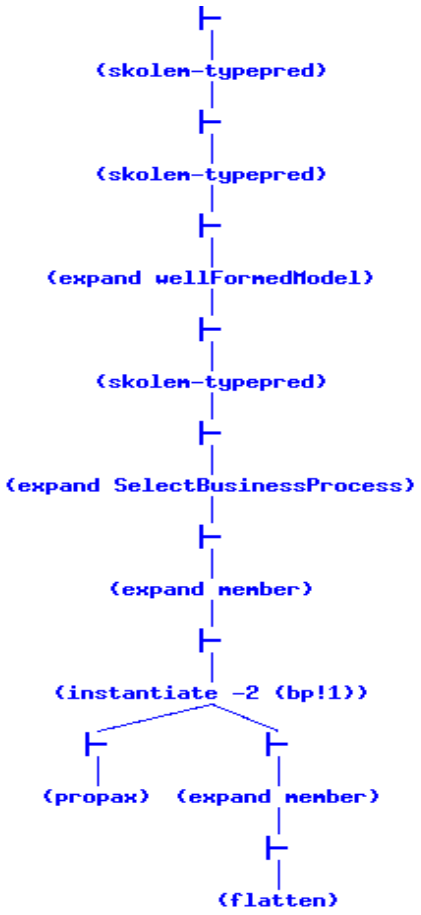

Figura 4.10: Árvore de prova do teorema WFM_SBP

4.2.2 Transformação SimpleEvaluateAfterAdvice

A especificação da transformação `SimpleEvaluateAfterAdvice` (`simpleEA`) foi criada para simplificar a transformação `evaluateAfterAdvice`, ou seja, fomos criando as transformações de forma incremental. A transformação `simpleEA` é mais simples que `evaluateAfterAdvice`, pois adiciona apenas um `FlowObject` a um `BusinessProcess`, enquanto a transformação `evaluateAfterAdvice` adiciona um processo de negócio inteiro do tipo `Advice` a outro processo, gerando um terceiro processo. Esta simplificação é útil, pois auxilia na prova, diminuindo o escopo de prova. Esta transformação não foi especificada em Alloy, pois não houve prova de teorema em Alloy.

Queremos traduzir a ideia de que `simpleEA` representa um template, uma relação entre dois processos de negócio, conforme ilustrado na Figura 4.11. A partir do processo `BP1` o processo `BP2` é construído. Em `BP1` introduzimos um `FlowObject(fo2)`, retiramos uma `Transition(t1)` e inserimos duas novas `Transitions` (`t2` e `t3`), criando `BP2`. As reticências indicam a parte comum entre os processos `BP1` e `BP2`, que é desconhecida, as setas são as transições e os retângulos são FlowObjects.

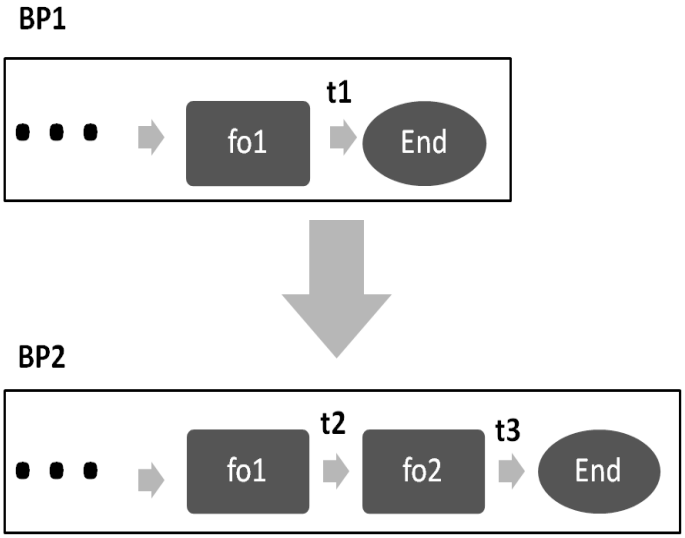

Figura 4.11: Ilustração de simpleEA

A `simpleEA` especifica que no processo `BP1` conhecemos apenas o final dele, ou seja, `fo1`, `t1` e `End`. O processo `BP2` possui apenas uma diferença do processo `BP1`. Se acrescentarmos um objeto (`fo2`) entre `fo1` e `End` em `BP1` obteremos `BP2`. A seguir temos a especificação de `simpleEA`.

```
simpleEA(bp11, bp22:  BusinessProcess,
             fo1, fo2:  FlowObject,
         tt1, tt2, tt3:  Transition):  bool =
  CFO?(fo2) ∧ CFO?(fo1) ∧
  BasicProcess?(ptype(bp22)) ∧ Advice?(ptype(bp11)) ∧
  (objects(bp11)(fo1)) ∧
```

$$
\begin{array}{l}
(\text{startObject}(\text{tt1}) = \text{fo1} \wedge \text{endObject}(\text{tt1}) = \text{End0}) \wedge \\
(\text{startObject}(\text{tt2}) = \text{fo1} \wedge \text{endObject}(\text{tt2}) = \text{fo2}) \wedge \\
(\text{startObject}(\text{tt3}) = \text{fo2} \wedge \text{endObject}(\text{tt3}) = \text{End0}) \wedge \\
(\text{transitions}(\text{bp11})(\text{tt1})) \wedge (\text{transitions}(\text{bp22})(\text{tt2})) \wedge (\text{transitions}(\text{bp22})(\text{tt3})) \wedge \\
(\forall\ (\text{tt}: \{t: \text{Transition} \mid \text{transitions}(\text{bp22})(t)\}): \neg\ \text{EQ\_TR}(\text{tt1}, \text{tt})) \wedge \\
(\forall\ (\text{tt}: \{t: \text{Transition} \mid \text{transitions}(\text{bp11})(t)\}): \neg\ \text{EQ\_TR}(\text{tt2}, \text{tt})) \wedge \\
(\forall\ (\text{tt}: \{t: \text{Transition} \mid \text{transitions}(\text{bp11})(t)\}): \neg\ \text{EQ\_TR}(\text{tt3}, \text{tt})) \wedge \\
(\text{objects}(\text{bp11}) \cup \text{list2set}[\text{FlowObject}]((:\text{fo2}:))) = \text{objects}(\text{bp22}) \wedge \\
(\text{transitions}(\text{bp11}) \cup \text{list2set}[\text{Transition}]((:\text{tt2}, \text{tt3}:))) = \\
(\text{transitions}(\text{bp22}) \cup \text{list2set}[\text{Transition}]((:\text{tt1}:))) \wedge \\
(\forall\ (\text{fo}: \{f: \text{FlowObject} \mid \text{objects}(\text{bp11})(f)\}): \neg\ \text{EQ\_FO}(\text{fo2}, \text{fo}))
\end{array}
$$

Com `simpleEA` nós provamos que a partir do template de uma sintaxe específica, que possui condições específicas, e da boa-formação do processo de negócio `BP1`, o processo de negócio resultante `BP2` também é bem-formado. Criamos o teorema (Teorema10) para especificar esta propriedade. Este teorema afirma que para uma determinada sintaxe definida anteriormente (`simpleEA`) se `BP1` é bem-formado, então `BP2` também é bem-formado, conforme especificação a seguir.

$$
\begin{array}{l}
\text{Teorema10}: \quad \text{THEOREM} \\
\forall\ (\text{bp11}, \text{bp22}: \text{BusinessProcess}, \\
\qquad \text{fo1}, \text{fo2}: \text{FlowObject}, \\
\quad \text{tt1}, \text{tt2}, \text{tt3}: \text{Transition}): \\
\text{simpleEA}(\text{bp11}, \text{bp22}, \text{fo1}, \text{fo2}, \text{tt1}, \text{tt2}, \text{tt3}) \wedge \text{wellFormed}(\text{bp11}) \Rightarrow \\
\text{wellFormed}(\text{bp22})
\end{array}
$$

Prova

Para provar este teorema, criamos lemas auxiliares (conforme Figura 4.12). Dividimos a prova, conforme as regras de boa-formação e provamos cada lema.

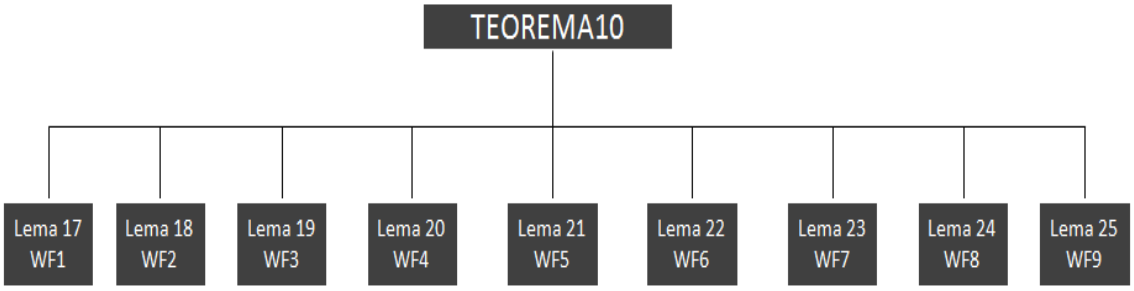

Figura 4.12: Quebra da prova do teorema Teorema10

No `Lema17` especificado a seguir, a última linha refere-se à regra `WF1`. Os demais lemas são definidos analogamente.

$$
\begin{array}{l}
\text{Lema17}: \quad \text{LEMMA}
\end{array}
$$

$\forall$ (bp11, bp22: BusinessProcess,
fo1, fo2: FlowObject,
tt1, tt2, tt3: Transition):
simpleEA(bp11, bp22, fo1, fo2, tt1, tt2, tt3) $\wedge$ wellFormed(bp11) $\Rightarrow$
WF1(bp22)

Para provar o Lema17, pela definição de simpleEA, o conjunto de objetos de BP2 é igual ao conjunto de objetos de BP1 unido com fo2. Por hipótese BP1 é bem-formado, então o Start e o End pertencem ao conjunto de objetos de BP1; como o conjunto de objetos de BP1 é um subconjunto do conjunto de objetos de BP2, então Start e End fazem parte do conjunto de objetos de BP2, o que satisfaz a regra de boa-formação WF1 e prova o Lema17.

As regras WF2 (onde o End não inicia nenhuma transição) e WF3 (onde o Start não finaliza nenhuma transição) foram provadas utilizando a definição de simpleEA. Pela igualdade de conjuntos definida em simpleEA temos que

$$(\text{transitions(bp11)} \cup \text{list2set}\big[\text{Transition}\big]((:\text{tt2}, \text{tt3}:))) =$$
$$(\text{transitions(bp22)} \cup \text{list2set}\big[\text{Transition}\big]((:\text{tt1}:))) \wedge$$

ou seja, as transições {t2,t3} são as únicas transições que fazem parte de BP2 e não fazem parte de BP1. Então, basta provarmos que estas transições satisfazem as regras WF2 e WF3. Pela simpleEA, temos que:

$$(\text{startObject(tt2)} = \text{fo1} \wedge \text{endObject(tt2)} = \text{fo2}) \wedge$$
$$(\text{startObject(tt3)} = \text{fo2} \wedge \text{endObject(tt3)} = \text{End0})$$

Logo, é possível perceber que o Start não finaliza t1 e nem t2 e o End não inicializa nenhuma das duas transições. Isto prova os lemas Lema18 e Lema19.

A regra WF4 especifica que o startObject e o endObject das transições do processo devem fazer parte do conjunto de objetos deste processo. Conforme a prova anterior, as transições que variam de um processo para outro são {t1,t2,t3}. BP2 possui {t2,t3}, então como sabemos que o startObject e o endObject destas transições são fo1, fo2 e End, então a regra WF4 é satisfeita, o que prova o Lema20.

As regras WF7 e WF9 especificam que só pode haver uma transição saindo de um objeto do tipo Activity e de um objeto do tipo Start, respectivamente, ou seja, o processo deve ser linear. Como o processo BP1 é bem-formado e BP2 manteve o final linear, então estas regras são satisfeitas. Isto prova os lemas Lema23 e Lema25.

A regra WF8 especifica que, quando um BusinessProcess for do tipo Adice, não pode haver uma Transition que saia do Start para o End diretamente. Como BP1 é bem-formado, então em BP1 não há Transition que saia do Start para o End. Em BP2, temos as mesmas Transitions de BP1 e mais duas {t2,t3}. Conforme visto anteriormente na prova das regras WF2 e WF3, as Transitions {t2,t3} não inicializam com Start, então a regra WF8 é satisfeita, o que prova o Lema24.

Prova das propriedades de alcançabilidade

As regras WF5 e WF6 estabelecem a alcançabilidade de um FlowObject. Na regra de boa-formação WF5 qualquer FlowObject deve ser alcançável a partir do Start e em WF6 qualquer FlowObject deve alcançar o End. Para prová-las foram utilizados lemas auxiliares, conforme estratégia de quebra de prova em problemas menores. Foi utilizada indução na prova devido à definição indutiva de extendsClosure (que compõe a especificação das regras WF5 e WF6).

A prova do Lema21 (referente a regra WF5) é dividida em dois passos: primeiro do Start até o FlowObject que é comum a BP1 e BP2 (Lema13); segundo do Start até o fo2, conforme ilustrado na Figura 4.13. O Lema6 determina que a partir do

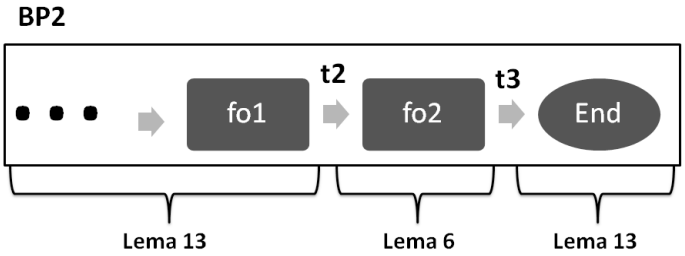

Figura 4.13: Quebra da prova do Lema21

Start chega-se ao fo2 em BP2 quando por hipótese BP1 é bem-formado e utiliza-se a transformação simpleEA, conforme especificação a seguir.

```
Lema6: LEMMA
 ∀ (bp11, bp22: BusinessProcess,
       fo1, fo2: FlowObject,
  tt1, tt2, tt3: Transition):
  simpleEA(bp11, bp22, fo1, fo2, tt1, tt2, tt3) ∧ wellFormed(bp11) ⇒
  extendsClosure(bp22, Start0)(fo2)
```

Para provar o Lema6, utilizamos a hipótese de que BP1 é bem-formado. Sabe-se que, por BP1 ser bem-formado, temos alcançabilidade do Start até fo1. Em BP2, também temos alcançabilidade do Start até fo1, pois até fo1 os processos BP1 e BP2 são iguais (possuem os mesmos FlowObjects e as mesmas Transitions).

$$(\text{startObject}(\text{tt2}) = \text{fo1} \wedge \text{endObject}(\text{tt2}) = \text{fo2})$$

Conforme especificação anterior, existe uma Transition t2 que liga o FlowObject fo1 a fo2. O FlowObject fo2 é alcançável a partir do Start em BP2, pois do Start chega-se a fo1 e de fo1 chega-se a fo2 pela Transition t2, isto prova o Lema6.

O Lema13 especifica que, para qualquer FlowObject que faça parte de BP1 e BP2, qualquer FlowObject é alcançável a partir do Start em BP2, quando por hipótese

BP1 é bem-formado e utiliza-se a transformação simpleEA, conforme especificação a seguir.

Lema13: LEMMA
∀ (bp11, bp22: BusinessProcess,
fo1, fo2: FlowObject,
tt1, tt2, tt3: Transition):
(∀ (fo: {f: FlowObject | objects(bp11)(f) ∧ objects(bp22)(f)}):
simpleEA(bp11, bp22, fo1, fo2, tt1, tt2, tt3) ∧ wellFormed(bp11) ⇒
extendsClosure(bp22, Start0)(fo))

Para provar o Lema13, tivemos dois passos. O primeiro passo é provar que a partir do Start chega-se ao FlowObject fo1. A segunda parte, é provar que a partir do Start chega-se ao FlowObject End (conforme ilustrado na Figura 4.13).

Por hipótese BP1 é bem-formado, por isto temos alcançabilidade do Start até fo1. Em BP2, temos alcançabilidade do Start até fo1, pois até fo1 os processos BP1 e BP2 são iguais (possuem os mesmos FlowObjects e as mesmas Transitions). Isto prova a primeira parte do Lema13.

Pela primeira parte da prova do Lema13, sabe-se que a partir do Start em BP2 chega-se ao FlowObject fo1. Pelo Lema6, sabe-se que a partir do Start chega-se ao fo2 em BP2. Então, para provar a segunda parte da prova do Lema13, falta provar que a partir do fo2 chega-se ao End em BP2.

Existe uma Transition que liga diretamente fo2 à End, a Transtion t3, conforme o trecho abaixo da definição de simpleEA.

(startObject(tt3) = fo2 ∧ endObject(tt3) = End0)

Com esta Transition, provamos que do FlowObject fo2 chega-se ao FlowObject End em BP2. Isto prova a segunda parte do Lema13, finalizando a prova deste lema.

Para provar que o Teorema10 satisfaz a regra WF6, Lema22, foram utilizados os lemas: Lema15 e Lema16. A Figura 4.14, ilustra a quebra de prova para o Lema22.

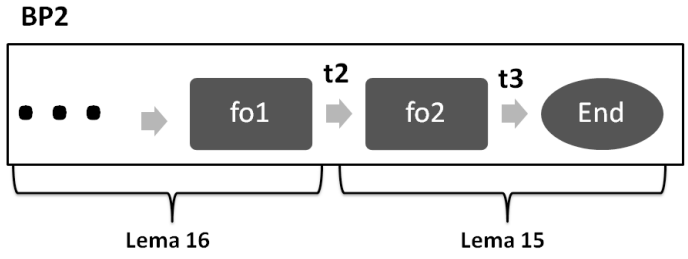

Figura 4.14: Quebra da prova do Lema22

O Lema15 determina que a partir do FlowObject fo2 chega-se ao FlowObject End em BP2, quando por hipótese BP1 é bem-formado e utiliza-se a transformação simpleEA, conforme especificação a seguir.

```
Lema15: LEMMA
 ∀ (bp11, bp22: BusinessProcess,
       fo1, fo2: FlowObject,
  tt1, tt2, tt3: Transition):
  simpleEA(bp11, bp22, fo1, fo2, tt1, tt2, tt3) ∧ wellFormed(bp11) ⇒
  extendsClosure(bp22, fo2)(End0)
```

Sabe-se pela definição de `simpleEA` que existe uma transição que liga `fo2` ao `End` em `BP2`, conforme especificação a seguir.

```
(startObject(tt3) = fo2 ∧ endObject(tt3) = End0)
```

Logo, ao expandir a definição de `extendsClosure` e instanciar com a transição `t3`, conseguimos provar que o `Lema15` é válido.

O `Lema16` especifica que, para qualquer `FlowObject` que faça parte de `BP1` e `BP2`, a partir de qualquer `FlowObject` o `End` é alcançável em `BP2`, quando por hipótese `BP1` é bem-formado e utiliza-se a transformação `simpleEA`, conforme especificação a seguir.

```
Lema16: LEMMA
 ∀ (bp11, bp22: BusinessProcess,
       fo1, fo2: FlowObject,
  tt1, tt2, tt3: Transition):
  ∀ (fo: {f: FlowObject | objects(bp11)(f) ∧ objects(bp22)(f)}):
  simpleEA(bp11, bp22, fo1, fo2, tt1, tt2, tt3) ∧ wellFormed(bp11)
  ⇒ extendsClosure(bp22, fo)(End0)
```

Em `BP2`, temos alcançabilidade de qualquer `FlowObject` até o `End`, em três passos: pelo `Lema15`, temos alcançabilidade de `fo2` até o `End`; pela `Transition t2` temos alcançabilidade de `fo1` até o `fo2`; e, pelos processos `BP1` e `BP2` serem iguais das reticências até `fo1` (possuem os mesmos `FlowObjects` e as mesmas `Transitions`), então qualquer `FlowObject` em `BP2` chega até `fo1`. Isto prova o `Lema16`.

4.2.3 Transformação EvaluateAfterAdvice

A transformação `evaluateAfterAdvice` compõe um `BusinessProcess Advice` com outro `BusinessProcess` resultando em um `BusinessProcess`. Por exemplo, uma análise simples de comunalidade e variabilidade nas Figuras 3.13 e 3.14 percebe-se que o único `FlowObject` que varia é "*Analisar requisitos legais do processo*" que está em vermelho no processo "*Aposentadoria*"; os outros `FlowObjects` são comuns para ambos os processos.

Para gerenciar a comunalidade e a variabilidade em uma abordagem composicional, na Figura 3.15, um processo do tipo `Advice` foi criado de modo análogo à Seção 3.2.2. Assim, o processo "*Aposentadoria*" é agora gerado por meio da transformação `evaluateAfterAdvice`, que compõe o processo do tipo `Advice` com o processo "*De-*

missão". Três `transitions` são removidas e duas novas `transitions` são criadas para conectar um processo ao outro, como mostra a Figura 3.16.

A transformação `evaluateAfterAdvice` no PVS é especificada tendo como parâmetros um processo de negócio `bp` e um advice `adv`, sendo que ambos são bem-formados e possuem o conjunto de objetos disjuntos. O processo de negócio resultante tem a união do conjunto de objetos de `bp` e `adv` e as transições previstas pela função `montaT`. A Figura 3.16 exemplifica como se comporta a transformação `evaluateAfterAdvice`.

```
evaluateAfterAdvice(adv: {a: BusinessProcess | wellFormed(a)},
                    bp:  {b: BusinessProcess | wellFormed(b) ∧
                             disjuntosFOS(objects(b), objects(adv))}):
  BusinessProcess = bp WITH
   [pid := pid(bp),
   ptype := ptype(bp),
   objects := (objects(bp) ∪ FO_adv?(adv)),
   transitions := montaT(adv, bp)]
```

A função `montaT` possui quatro partes que definem o conjunto de transições do `BusinessProcess` resultante do `evaluateAfterAdvice`. A primeira parte (`transitionsNotMatches`) é composta por transições que fazem parte de `bp` retirando a `Transition` que inicia com o `FlowObject` anotado. A segunda parte (`innerTransitionsAdv`) são transições que fazem parte do `adv` menos duas `Transitions` (a `Transition` que inícia com `Start` e a `Transition` que termina com `End` não fazem parte do `innerTransitionsAdv`). As duas primeiras partes são ilustradas na Figura 4.15.

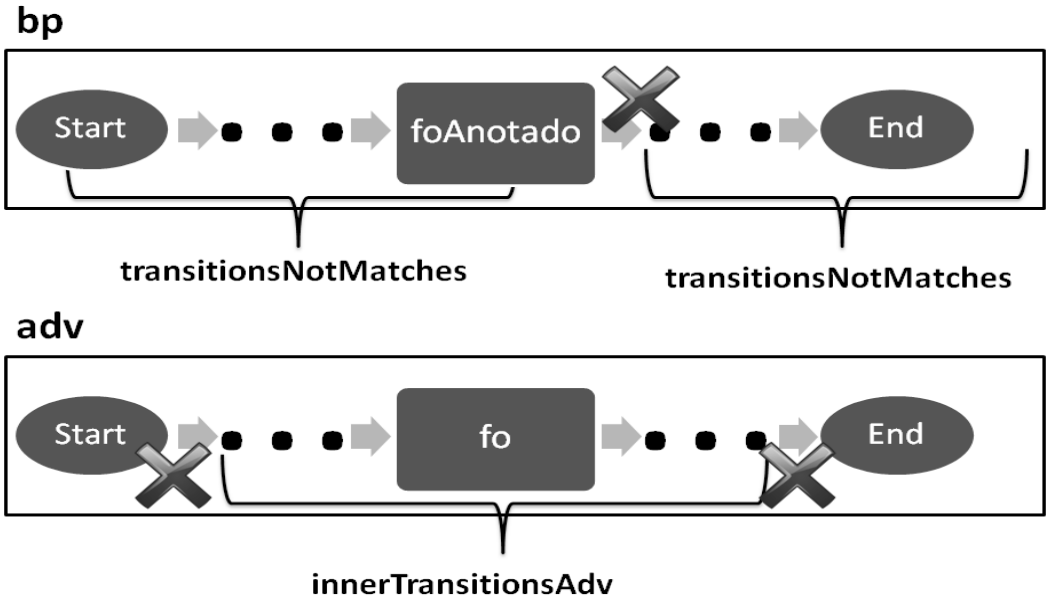

Figura 4.15: Representação de montaT para transitionsNotMatches e innerTransitionsAdv

A terceira parte (`transitionsBaseAdv`) possui uma transição que liga o `FlowObject` anotado de `bp`, que é do tipo complexo, ao primeiro `FlowObject` do tipo complexo de `adv`. A quarta parte (`transitionsAdvBase`) é composta por uma transição que liga o último `FlowObject` do tipo complexo (`CFO?`) do `adv` até o `FlowObject` seguinte ao `FlowObject` anotado em `bp`. As duas últimas partes são ilustradas na Figura 4.16.

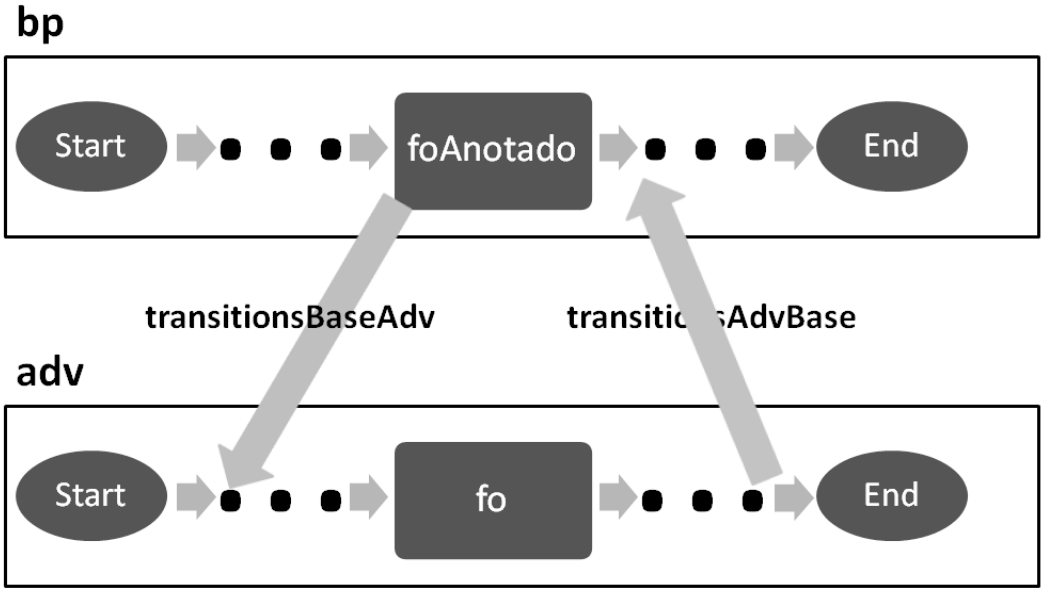

Figura 4.16: Representação de montaT para transitionsBaseAdv e transitionsAdvBase

O predicado `matches` retorna verdadeiro quando há casamento do `flowObject` anotado de `bp` com o `pointcut` do processo `adv` (ponto de junção). O predicado `BPT?` foi criado para retornar a `Transition` que tem como `startObject` o `FlowObject` anotado em `bp`. O predicado `ST?` foi criado para retornar a `Transition` que tem como `startObject` o `FlowObject Start` em `adv`. O predicado `ET?` foi criado para retornar a `Transition` que tem como `endObject` o `FlowObject End` em `adv`. A seguir, a especificação de `montaT`.

$$
\begin{aligned}
&\text{montaT}(\text{adv}: \{a: \text{BusinessProcess} \mid \text{wellFormed}(a)\},\\
&\qquad \text{bp}: \{b: \text{BusinessProcess} \mid \text{wellFormed}(b) \wedge \\
&\qquad\qquad \text{disjuntosFOS}(\text{objects}(b), \text{objects}(\text{adv}))\}):\\
&\quad \text{set}\big[\text{Transition}\big] =\\
&\quad ((\text{transitionsNotMatches} \cup \text{innerTransitionsAdv}) \cup (\text{transitionsBaseAdv} \cup \text{transitionsAdvBase}))\\
&\qquad \text{WHERE transitionsNotMatches} =\\
&\qquad\qquad \{t_1: \text{Transition} \mid \text{transitions}(\text{bp})(t_1) \wedge\\
&\qquad\qquad\qquad \text{Advice?}(\text{ptype}(\text{adv})) \wedge\\
&\qquad\qquad\qquad \text{CPC?}(\text{pc}(\text{ptype}(\text{adv}))) \wedge\\
&\qquad\qquad\qquad (\neg\ \text{matches}(\text{startObject}(t_1),\\
&\qquad\qquad\qquad \text{pc}(\text{ptype}(\text{adv}))))\},\\
&\qquad\qquad \text{innerTransitionsAdv} =\\
&\qquad\qquad \{t_2: \text{Transition} \mid \text{transitions}(\text{adv})(t_2) \wedge
\end{aligned}
$$

```
                                CFO?(startObject(t2)) ∧
                                CFO?(endObject(t2))},
transitionsBaseAdv =
  {t3: Transition | ∃ (x1: (BPT?(adv, bp)),
                      y1: (ST?(adv))):
                    startObject(t3) = startObject(x1) ∧
                    endObject(t3) = endObject(y1) ∧
                    condition(t3) = condition(x1)},
transitionsAdvBase =
  {t4: Transition | ∃ (x2: (ET?(adv)),
                      y2: (BPT?(adv, bp))):
                    startObject(t4) = startObject(x2) ∧
                    endObject(t4) = endObject(y2) ∧
                    condition(t4) = condition(x2)}
```

O teorema a seguir especifica que `evaluateAfterAdvice(adv,bp)` gera um produto bem-formado, se o processo de negócio `bp` é bem-formado e `adv` refere-se a um processo do tipo `Advice` existente, ou seja, `evaluateAfterAdvice(adv,bp)` também preserva a boa-formação:

```
WFM_EVAL: THEOREM
 ∀ (adv: {a: BusinessProcess | wellFormed(a) ∧
                               Advice?(ptype(a)) ∧
                               CPC?(pc(ptype(a)))},
    bp: {b: BusinessProcess | wellFormed(b) ∧
                              disjuntosFOS(objects(b), objects(adv))},
     f: (FOAnotado?(adv, bp))):
  wellFormed(evaluateAfterAdvice(adv, bp))
```

Prova

A estratégia utilizada para provar[2] o teorema `WFM_EVAL` foi quebrar a prova em problemas menores, criando lemas menores durante a prova, conforme Figura 4.17. Em particular, a composição é obtida através da expansão do predicado `wellFormed` e replicando o contexto, levando a lemas (Lema_EVAL1-EVAL9) correspondentes a cada uma das noves regras de boa-formação vistas na Seção 4.1.2. Os lemas são os mesmos em termos do contexto, a parte que muda é a especificação da regra de boa-formação. Por exemplo, no lema seguinte:

```
Lema_EVAL1: LEMMA
 ∀ (adv: {a: BusinessProcess | wellFormed(a) ∧
                               Advice?(ptype(a)) ∧
                               CPC?(pc(ptype(a)))},
```

[2]A prova completa em PVS pode ser encontrada no site http://www.gisellemachado.com/mestrado/pvs/arquivos/proof.zip

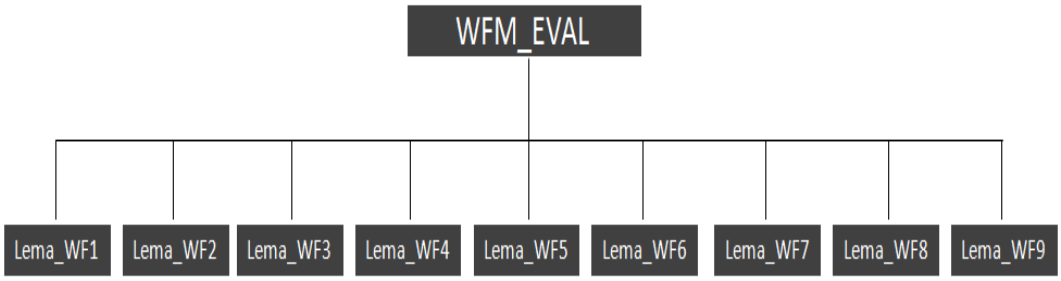

Figura 4.17: Quebra da prova do teorema WFM_EVAL

$$\begin{aligned} &\texttt{bp:}\ \{b\texttt{:}\ \text{BusinessProcess}\ \mid\ \text{wellFormed}(b)\ \wedge \\ &\qquad\qquad\qquad\qquad\qquad\qquad \text{disjuntosFOS}(\text{objects}(b),\ \text{objects}(\text{adv}))\},\\ &\quad f\texttt{:}\ \ (\text{FOAnotado?}(\text{adv},\ \text{bp}))\texttt{:}\\ &\text{WF1}(\text{evaluateAfterAdvice}(\text{adv},\ \text{bp})) \end{aligned}$$

a última linha refere-se a regra `WF1`. Outros lemas são definidos de forma análoga. A seguir, apresentamos o esboço da prova de tais lemas. Em particular a regra `WF1` estabelece que `FlowObjects` (`Start` e `End`) devem fazer parte do conjunto de objetos de `bp`. Dada a transformação `evaluateAfterAdvice`, o conjunto de objetos resultante de `evaluateAfterAdvice` é composto pela união dos objetos de `bp` com os objetos de `adv` sem o `Start` e o `End`. Como `bp` é bem-formado, então ele satisfaz `WF1` assim como o `BP` resultante de `evaluateAfterAdvice`. Isto prova `Lema_EVAL1`.

O Lema_EVAL2 busca provar a regra WF2, que diz que o End não pode ser o `startObject` de qualquer `Transition` de um processo `bp`. E Lema_EVAL3 busca provar a regra `WF3` que especifica que o Start não pode ser o `endObject` de qualquer `Transition` de um processo. Sabe-se que estas regras são válidas para `bp` e `adv`, pois são bem-formados. Sabe-se ainda que o conjunto de transições do `BusinessProcess` resultante do `evaluateAfterAdvice` é igual ao `montaT`.

As `Transitions` de `transitionsNotMatches` são compostas por transições que fazem parte de `bp`, logo satisfazem as regras de boa-formação `WF2` e `WF3`. `InnerTransitionsAdv` são transições que fazem parte do `adv`, logo, também satisfazem as regras de boa-formação `WF2` e `WF3`. `TransitionsBaseAdv` possui uma transição que liga o `FlowObject` anotado de `bp`, que é do tipo complexo, ao primeiro `FlowObject` do tipo complexo de `adv`. `TransitionsAdvBase` é composta por uma transição que liga o último `FlowObject` do tipo complexo (`CFO?`) do `adv` até o `FlowObject` seguinte ao `FlowObject` anotado em `bp`. Logo, é possível perceber que o `End` não inicia nenhuma das transições e o `Start` não finaliza nenhuma `Transition`, o que prova Lema_EVAL2 e Lema_EVAL3.

A regra `WF4` diz que o `startObject` e `endObject` de qualquer `Transition` de um processo devem fazer parte do conjunto de objetos do processo. Sabe-se que esta regra é válida para `bp` e `adv`, pois são bem-formados. Sabe-se ainda que o conjunto de transições do `BusinessProcess` resultante do `evaluateAfterAdvice` é igual ao `montaT`.

A primeira parte é composta por transições que fazem parte de `bp`, logo satisfaz a regra de boa-formação `WF4`. A segunda parte são transições que fazem parte do `adv`,

logo, também satisfaz a regra de boa-formação WF4. A terceira parte possui uma transição que liga o FlowObject anotado de bp ao primeiro FlowObject do tipo complexo de adv. A quarta parte é composta por uma transição que liga o último FlowObject do adv até o FlowObject seguinte ao FlowObject anotado em bp. Logo, é possível notar que os FlowObjects que fazem parte das quatro partes são FlowObjects que fazem parte do conjunto de FlowObjects de bp, ou do conjunto de FlowObjects de adv, que é precisamente o conjunto de FlowObjects do resultado da transformação evaluateAfterAvice. Isto finaliza a prova do lema Lema_EVAL4.

Como explicado na Seção 4.1.2, WF5 refere-se à alcançabilidade do objeto Start, ou seja, todo objeto é alcançável a partir do Start. Sabe-se que esta regra é válida para bp e adv, pois eles são bem-formados. Para provar Lema_EVAL5, contamos com lemas de granularidade mais fina correspondentes aos três grupos de objetos resultantes da partição suportada pelo objeto anotado (joinpoint) que casa com o pointcut do advice como mostrado na Figura 4.18. A primeira parte é composta por objetos do processo base bp até o objeto anotado (inclusive) e está relacionada com o lema Lema_WF5_01. A segunda parte compreende todos os objetos do advice exceto o Start e o End e está relacionada com o lema Lema_WF5_02. Finalmente, a terceira parte é composta por objetos de bp que estão depois do objeto anotado e corresponde ao lema Lema_WF5_03.

Cada um dos tais lemas refere-se à alcançabilidade do Start (de bp) até o conjunto de objetos da partição. O Lema_WF5_01 decorre da boa-formação de bp. O Lema_WF5_02 decorre a partir da definição da transformação e da boa-formação de adv. Por último, o Lema_WF5_03 decorre da definição da transformação e da boa-formação de bp. Em PVS, todas estas provas dependem de indução, uma vez que a definição de alcançabilidade é indutiva (conforme Seção 4.1.2). Isto conclui a prova do lema Lema_EVAL5.

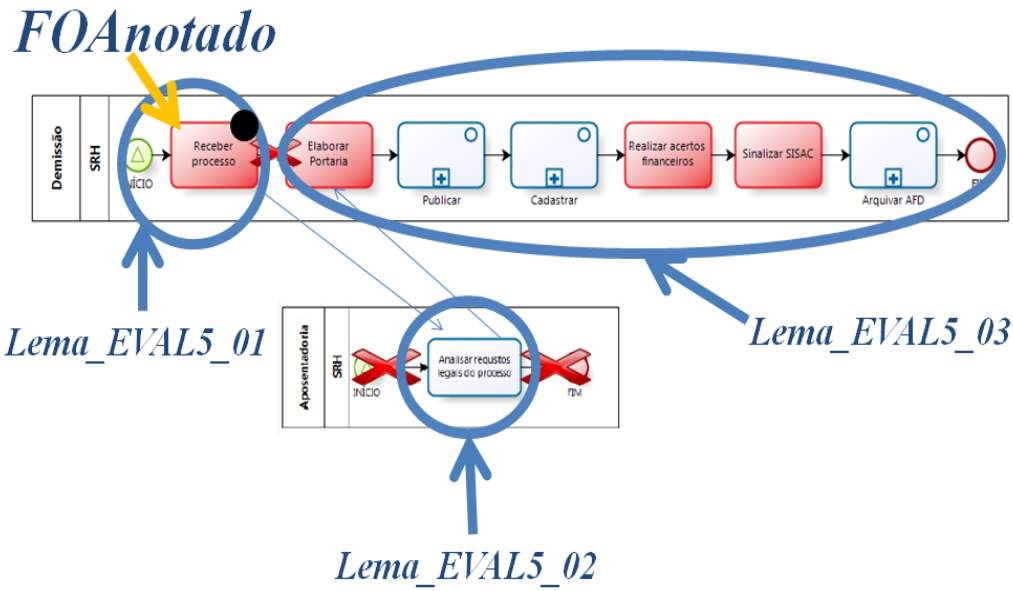

Figura 4.18: Partição de objetos para provar a alcançabilidade evaluateAdvice.

A regra WF6 especifica que a partir de qualquer FlowObject é possível chegar no End. Para provar esta regra, foi utilizada a mesma intuição utilizada na prova da regra WF5.

Por razões de tempo, os lemas auxiliares utilizados para provar as regras WF5 e WF6 do teorema WFM_EVAL (Lema_WF5_01, Lema_WF5_02, Lema_WF5_03, Lema_WF6_01, Lema_WF6_02 e Lema_WF6_03) ainda não foram provados no PVS. No entanto, para termos alguma evidência, utilizamos Alloy para verificar se são possíveis de serem provados. No Alloy fizemos afirmações e executamos com o comando check para o escopo de até 10 instâncias para cada assinatura em busca de contraexemplos, que não foram encontrados para o escopo atual do trabalho, o que sugere que sejam lemas válidos.

A regra WF7 diz que um objeto do tipo Activity deve ter apenas uma Transition saindo dele. Sabe-se que esta regra é válida para bp e adv, pois são bem-formados. Sabe-se ainda que o conjunto de transições do BusinessProcess resultante do evaluateAfterAdvice é igual ao montaT.

A primeira parte (transitionsNotMatches) é composta por transições que fazem parte de bp, logo satisfaz a regra de boa-formação WF7. A segunda parte (innerTransitionsAdv) são transições que fazem parte do adv, logo também satisfaz a regra de boa-formação WF7. A terceira parte (transitionsBaseAdv) possui uma transição que liga o FlowObject anotado de bp, ao primeiro FlowObject do tipo complexo de adv, como possui apenas uma transição resultante, então também satisfaz a regra WF7. A quarta parte (transitionsAdvBase) é composta por uma transição que liga o último FlowObject do adv até o FlowObject seguinte ao FlowObject anotado em bp, como possui apenas uma transição resultante, então também satisfaz a regra WF7. O que prova o Lema_EVAL7.

A regra WF8 diz que, quando um BusinessProcess for do tipo Advice, não pode haver uma Transition que saia do FlowObject Start para o FlowObject End e a regra WF9 diz que um objeto do tipo Start deve ter apenas uma Transition saindo dele. Sabe-se que estas regras são válidas para bp e adv, pois são bem-formados. Sabe-se ainda que o conjunto de transições do BusinessProcess resultante do evaluateAfterAdvice é igual ao montaT. A função montaT especificada possui quatro partes que definem o conjunto de transições do BusinessProcess resultante do evaluateAfterAdvice.

A primeira parte é composta por transições que fazem parte de bp, logo satisfazem ambas as regras WF8 e WF9. A segunda parte são transições que fazem parte do adv, logo, também satisfazem as regras de boa-formação WF8 e WF9. A terceira parte possui uma transição que liga o FlowObject anotado de bp, que é do tipo complexo, ao primeiro FlowObject do tipo complexo de adv. Como possui apenas uma transição resultante entre FlowObjects do tipo complexo, pois trabalhamos com a hipótese de termos apenas um ponto de junção, então também satisfazem as regras WF8 e WF9. A quarta parte é composta por uma transição, pois trabalhamos com a hipótese de termos apenas um ponto de junção, que liga o último FlowObject do adv do tipo complexo até o FlowObject seguinte ao FlowObject anotado em bp, que pode ser um End ou um FlowObject do tipo complexo, então também satisfazem as regras WF8 e WF9. O que prova Lema_EVAL8 e Lema_EVAL9.

Capítulo 5

Conclusão

Neste trabalho tratamos a especificação de LPPN em Alloy e PVS, incluindo a definição de regras de boa-formação. Nós também codificamos duas transformações e provamos que elas são corretas (preservam as regras de boa-formação) em relação a uma semântica formal em PVS. Neste Capítulo, é apresentada a conclusão dividida em quatro seções. Primeiro são apresentadas as contribuições deste trabalho. Em seguida, é feita uma análise comparativa entre Alloy e PVS. Após, são apresentados os trabalhos relacionados. E, por fim, são relacionados os possíveis trabalhos futuros.

5.1 Contribuições

Com a especificação formal em Alloy foi possível entender precisamente como funciona a LPPN e identificar tipos e regras de boa-formação. Com PVS especificamos precisamente uma linguagem da LPPN, as propriedades de boa-formação e a composição de transformações e verificamos que as transformações especificadas preservam a boa-formação dos produtos, sem ter que verificar individualmente cada instância da LPPN. Ou seja, provou-se que duas das transformações da LPPN representadas preservam a boa-formação em qualquer LPPN.

Na especificação foi utilizada uma estratégia conforme a Figura 5.1. Inicialmente foi realizada a especificação do problema e posteriormente criou-se a especificação em Alloy. Uma das atividades mais realizadas na ferramenta *Alloy Analyzer* foi checar asserções e achar contraexemplos quando as afirmações eram falsas. Neste caso, sabemos que uma afirmação é falsa quando encontramos um contraexemplo e, no caso de não acharmos contraexemplos, existe grande chance da afirmação ser verdadeira. Estes contraexemplos auxiliaram no refinamento da especificação em Alloy. Depois de ter sido obtida uma especificação mais estável em Alloy, foi realizada a especificação em PVS.

Na Figura 5.1, a seta que vai do PVS e retorna para a especificação do problema foi devido à evolução da especificação em PVS e à necessidade de especificar novas afirmações em Alloy. Usamos as novas especificações em Alloy após o início da especificação em PVS para termos mais confiança de que afirmações especificadas em teoremas no PVS eram válidas. Dessa forma, evitamos perder tempo tentando provar um teorema ou um lema no PVS com algo que não é possível ser provado.

Iteração de transformações, na Figura 5.1, é o aumento gradativo e incremental da especificação das transformações (o conceito de transformação pode ser visto

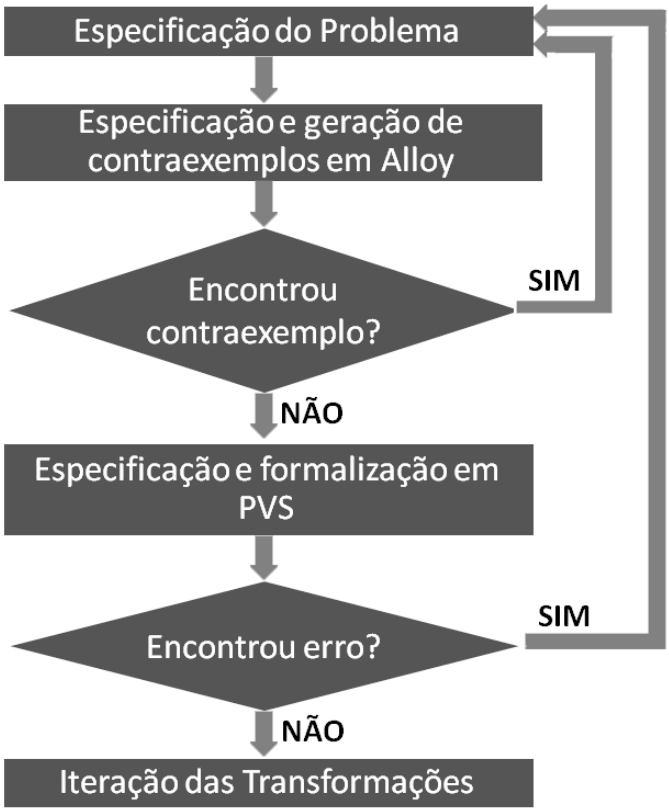

Figura 5.1: Fluxo de estratégia de especificação

na Seção 2.3) subindo o nível de forma incremental, ou seja, aumentando gradativamente a complexidade de transformações especificadas. Esta estratégia foi utilizada em PVS, quando especificamos a transformação `SimpleEA` antes da transformação `evaluateAfterAdvice`, por exemplo.

Na iteração das transformações houve mudanças devido à simplificação do problema, o que altera diretamente a especificação do problema. A simplificação do problema ocorreu para minimizar o escopo, em busca de melhor entendimento da linguagem da LPPN em Alloy. Como o escopo estava muito amplo, o Alloy gerava muitos exemplos complexos, o que dificultou a especificação precisa de exemplos simples.

A experiência com métodos formais, apesar de trabalhosa, gerou resultados significativos e precisos, o que aumenta a confiança de que os produtos da LPPN são bem-formados.

5.1.1 Discussão Alloy

Nesta seção, relata-se a experiência em trabalhar com Alloy. Em Alloy, não tivemos muitas dificuldades devido ao fato de a ferramenta possuir recursos visuais e ser de fácil usabilidade. Além disto, o livro utilizado para estudo de *Alloy* [Jackson (2011)] é bem didático, o que facilitou o aprendizado.

A primeira tática que utilizamos faz parte da estratégia de trabalho descrita na Seção 1.2. Ao utilizarmos Alloy, buscamos entender melhor o problema, trabalhando de forma gradual. Ou seja, fomos incrementando a especificação aos poucos. Como Alloy

possui uma lógica relacional bem concisa, isto facilitou o andamento e o aumento gradual da especificação.

Comparando as Figuras 3.2 e 5.2 é possível verificar a redução na especificação dos tipos. Na Figura 5.2 notamos uma representação circular (`in`) e outra retangular (`extends`). A declaração a seguir

```
sig BusinessProcess {
    objects: set FlowObject
}
abstract sig FlowObject {
    type: Type,
    transition: set FlowObject
}
one sig Start, End in FlowObject {}
abstract sig Type {}
one sig  Activity, Gateway extends Type {}
```

é a especificação que gerou o metamodelo do escopo reduzido.

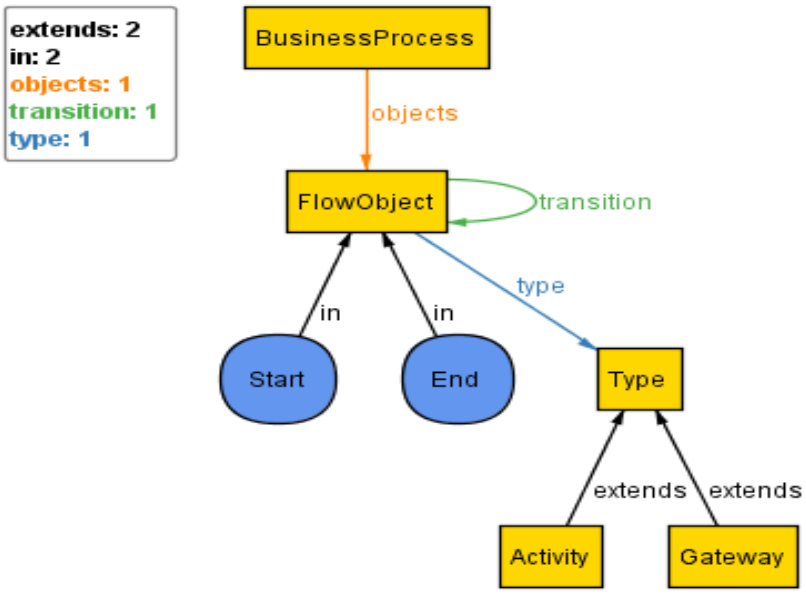

Figura 5.2: Metamodelo de tipos com escopo reduzido

Quando utilizamos o `in` dizemos que ele é um objeto do tipo FlowObject específico, quando utilizamos `extends` assumimos que existem subtipos. No exemplo do metamodelo com escopo reduzido estamos dizendo que o `Start` e o `End` são objetos específicos do tipo `FlowObject` que podem ter o subtipo chamado de `Type` `Activity` ou `Gateway`. Por simplificação, assumimos no modelo referente à Figura 3.2 que o `Start` e o `End` são um subtipo de `FlowObject`, pois não podem ser nem uma atividade, nem uma decisão. Após trabalhar com o metamodelo com escopo reduzido fomos ampliando os tipos até especificar todos.

Dando seguimento ao trabalho, foi realizada a especificação de regras de boa-formação de tipos. A primeira regra especificada foi a `WF1`. Esta regra foi criada com base no manual BPMN. Ao iniciarmos a especificação das regras tivemos o cuidado com a utilização de fatos na teoria. Temos que tomar cuidado já que se fizermos uma afirmação falsa, teremos sempre verdade, pois de falso tudo é verdade.

Mensurar o impacto de uma alteração na especificação é uma tarefa nada trivial. Então, para evitar o retrabalho e também para facilitar a manutenção da especificação a segunda tática adotada em Alloy foi o particionamento da especificação, conforme Figura 3.1.

Buscando aprimorar o desenvolvimento do trabalho, fizemos verificações de propriedades em casos específicos (exemplos com valores determinados), o mais detalhado possível, buscando por falhas na especificação, com o auxílio dos contraexemplos gerados pelo `Alloy Analyzer`, sempre do menos detalhado para o mais detalhado. Como exemplo temos a especificação da regra `WF4`.

A criação da regra "os objetos que fazem parte das transições devem ser objetos do `BusinessProcess`" (`WF4`) foi observada por exemplos gerados no Alloy e seus diagramas. O *Alloy Analyzer* gerou exemplos de objetos que faziam parte das transições de um `BusinessProcess`, mas que não pertenciam ao conjunto de objetos do `BusinessProcess`. No exemplo da Figura 5.3, temos que o `FlowObjectComplexo 1`, que está à direita do `Start`, não faz parte do conjunto de `objects` do `BusinessProcess Allowance`, pois não pertence à relação `objects`. Com a identificação desta falha no exemplo gerado, a regra `WF4` foi criada.

A especificação formal exige conhecimento e organização. Uma simples mudança na especificação pode afetar toda a especificação e comprometer a verificação da especificação já realizada.

5.1.2 Discussão PVS

Como a documentação em PVS é despersa em vários manuais, há muitos detalhes na especificação e na prova que só são aprendidos com a experiência. De fato, a primeira lição aprendida foi sobre o impacto da ordem da especificação de teoremas e tipos com relação as provas dos teoremas e lemas. Por exemplo, a especificação da função indutiva `extendsClosure` (Seção 4.1.2) é melhor descrita colocando primeiramente a base em vez da recursão, assim simplificando a prova correspondente por indução.

Além disso, vimos que uma ligeira alteração sintática na especificação pode ter um impacto nas provas, em muitos casos, que descartam a prova toda. Portanto, recomenda-se que ao provar um teorema, decomponha a prova em lemas, que podem ser reusados, minimizando a dependência sintática (por exemplo, evitando inúmeras referências no sequente) nos comandos de prova (por exemplo, skolemize, instantiate). Por exemplo, essa estratégia foi seguir na prova do teorema `WFM_EVAL` na especificação de `wf`. Como a regra `WF9` foi a última a ser especificada e alguns teoremas já haviam sido provados, então a regra de `WF9` foi colocada no início da especificação de `wf` para ela aparecer por último na hora da prova, para não perder a prova já feita.

Como parte da estratégia de prova, buscamos começar a prova usando os comandos de menor granularidade até um ponto onde a prova processa a maior granularidade com táticas embutidas. Em particular, muitas vezes visa simplificar as fórmulas embutidas

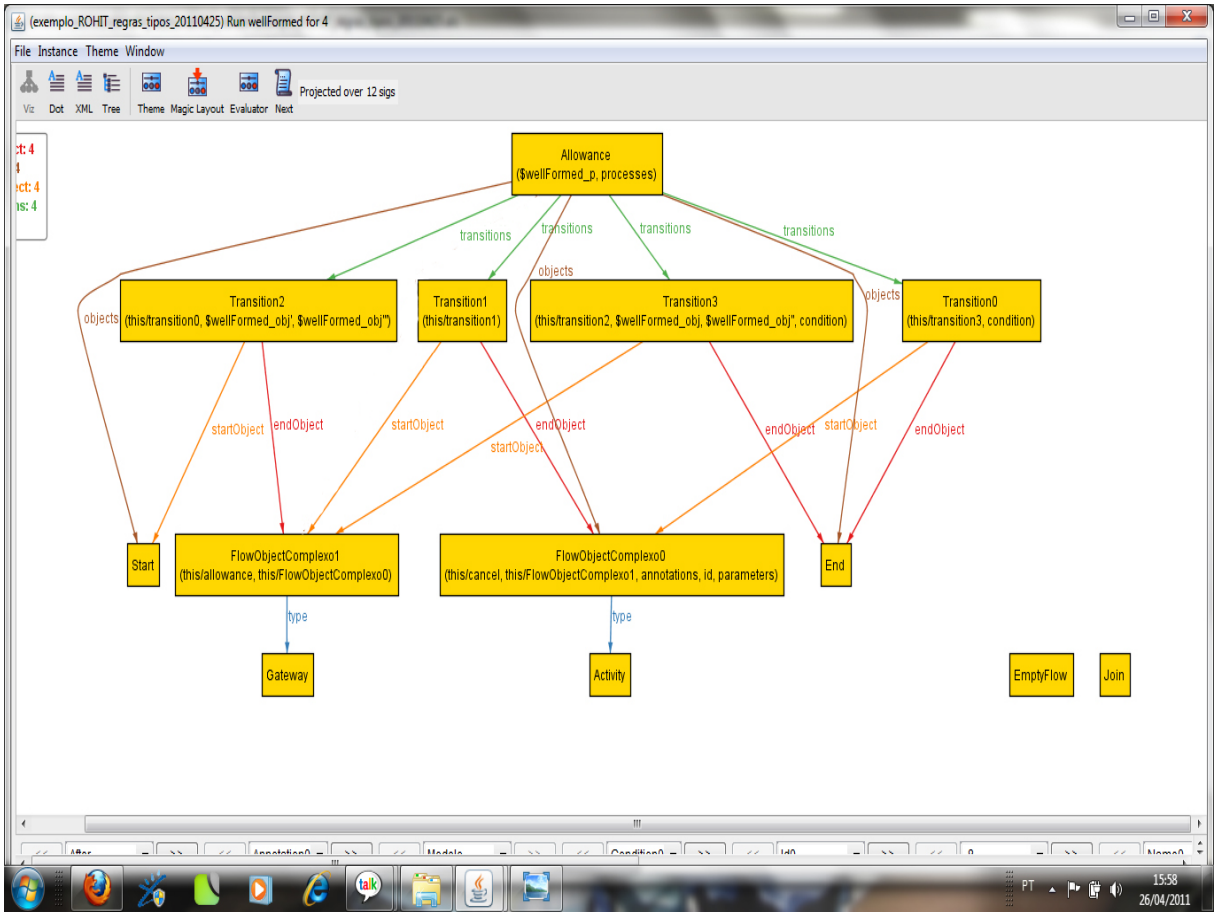

Figura 5.3: Exemplo de modelo de diagrama gerado pelo *Alloy Analyzer* pela execução do comando `run`, onde há um contraexemplo para regra `WF4`

com quantificação e conectivos, de modo a evitar a expansão das definições, o que facilita a compreensão e o progresso da prova.

Uma estratégia utilizada foi primeiro tentar provar, em um alto nível de abstração, os teoremas manualmente, antes de usar o assistente de prova. Isso foi fundamental na identificação de estratégias de decomposição, adequadas para quebrar um teorema em lemas auxiliares, buscando garantir que o teorema possa ser provado com base no lema e que cada um deles também é provado com esforço reduzido. Por exemplo, esta estratégia foi utilizada na prova do teorema `Teorema10`. Um benefício além desta decomposição é identificar lemas reutilizáveis, que de outra forma podem organizar e auxiliar o processo de prova também em termos de desempenho.

Provar teoremas não é uma atividade trivial, é necessário saber quais as premissas e se a especificação delas permite inferir o que se quer provar. Neste trabalho utilizamos algumas táticas de prova como:

- **dividir problema**: se o problema for grande, a primeira tática utilizada é dividir em pequenas partes, buscando sempre o que é mais intuitivo com uma abordagem *top-down*;

- **utilizar casos triviais**: provar casos simples primeiramente, utilizando exemplos simples. Assim conseguimos identificar falhas e corrigí-las.
- **provar casos gerais**: por fim, foram feitas as provas dos casos gerais de forma gradativa. Resolvendo primeiro as partes triviais, mais simples, e aumentando o problema parcialmente, tornando-o mais complexo e mais geral (Por exemplo, a estratégia utilizada para provar o `Teorema10`, primeiramente realizando a quebra do teorema em lemas menores e após provando os lemas mais simples até provar o teorema como um todo).

Com efeito, os esforços de especificação e prova não são triviais. No entanto, obter experiência e reutilizá-la pode acelerar o processo. Além disso, o retorno do investimento é desejável porque as provas certificam propriedades, ou seja, elas quantificam sobre o domínio da LPPN. Isto é diferente de verificar o modelo, onde o esforço de prova é menor (automático), mas pode também levar algum tempo e também não é completo, ou seja, está limitado a um certo escopo.

5.2 Comparação Alloy e PVS

Nesta seção é apresentada a comparação entre as duas especificações formais, Alloy e PVS. Primeiramente, é realizada uma análise qualitativa e, em seguida, uma análise quantitativa.

5.2.1 Análise Qualitativa

A especificação em Alloy e a utilização da ferramenta *Alloy Analyzer* auxiliaram diretamente na criação das regras de boa-formação para um modelo de processo de negócio. Uma atividade realizada com auxílio da ferramenta *Alloy Analyzer* foi realizar afirmações e achar contraexemplos, quando as afirmações eram falsas. Neste caso, sabemos que uma afirmação é falsa quando encontramos um contraexemplo e no caso de não acharmos contraexemplos suspeita-se que a afirmação seja verdadeira.

Na transição da especificação de Alloy para PVS, primeiramente foram especificados os tipos. As assinaturas em Alloy foram especificadas como tipos em PVS. Em PVS não houve a necessidade de especificar os tipos e as regras de boa-formação de forma gradual, pois a evolução da especificação em Alloy auxiliou na especificação da linguagem e das transformações no PVS de forma clara. Por exemplo, a especificação da assinatura `BusinessProcessModel` em Alloy,

```
sig BusinessProcessModel{
    processes: set BusinessProcess
}
```

e a especificação em PVS para o tipo `BusinessProcessModel`

```
 BusinessProcessModel:  DATATYPE
  BEGIN
   BPM(processes:  set[BusinessProcess]):  BusinessProcessModel?
```

```
END BusinessProcessModel
```

são correspondentes. Em ambas as especificações o tipo `BusinessProcessModel` é um conjunto de `BusinessProcess`, porém em PVS os `datatypes` possuem um axioma gerador implícito que as assinaturas em Alloy não possuem.

O mesmo ocorreu para o caso do predicado `wellFormed` que em Alloy,

```
pred wellFormed[p: BusinessProcess]{
  WF1[p] ∧ WF2[p] ∧ WF3[p] ∧ WF4[p] ∧ WF5[p] ∧ WF6[p] ∧
  WF7[p] ∧ WF8[p] ∧ WF9[p]
}
```

e em PVS,

```
wellFormed(p: BusinessProcess): bool =
 WF1(p) ∧ WF2(p) ∧ WF3(p) ∧WF4(p) ∧ WF5(p) ∧ WF6(p) ∧
 WF7(p) ∧ WF8(p) ∧ WF9(p)
```

também foi especificado da mesma forma, ou seja, é composto por nove regras de boa-formação.

No PVS foi utilizado o provador de teoremas para provar lemas e teoremas mais gerais, buscando garantir para a LPPN a boa-formação de todas as instâncias. Uma das vantagens em utilizar o PVS com relação ao Alloy foi de obter a prova de que as transformações preservam a boa-formação dos produtos com a utilização do provador de teoremas. Por outro lado, Alloy, apesar de fazer verificação automática, não oferece análise completa.

Outra distinção entre PVS e Alloy foi a utilização de definição indutiva. No PVS criamos o predicado `extendsClosure` (que possui definição indutiva) para conseguir especificar as regras de boa-formação `WF5` e `WF6` para o tipo `BusinessProcess`. Assim foi possível criar a alcançabilidade entre os objetos de um `BusinessProcess`. No Alloy não foi possível especificar de forma direta estas regras, pois não suporta definição indutiva. Em Alloy, utilizamos o operador relacional de fecho reflexivo-transitivo.

A utilização de indução foi trabalhosa e demandou tempo para aprendizado, tanto para especificar as regras de boa-formação e entendimento da especificação quanto para fazer as provas dos teoremas. A maior dificuldade foi justamente provar as regras que garantem alcançabilidade (regras de boa-formação `WF5` e `WF6`). A dificuldade de realizar a prova destas duas regras foi devido à especificação indutiva, que exigiu o uso e a experiência no uso de comandos de prova específicos. Durante a prova, houve casos de entrar em loop, por exemplo, obter o mesmo sequente várias vezes após uma instanciação. A dificuldade em identificar qual caminho seguir durante a prova, também levou ao loop. Assim, toda a prova teve que ser refeita.

5.2.2 Análise Quantitativa

Nossa meta nesta seção é medir o esforço de especificação e verificação para ajudar a prevê-lo em trabalhos semelhantes. Para isso, utilizamos o GQM (Goal Question Metric)

[Basili et al. (1994)]. Na tabela a seguir é ilustrada a meta da nossa medição.

Tabela 5.1: Meta GQM

Objetivo	Comparar
Problema	Esforço
Objeto	Especificação e verificação em Alloy e PVS
Ponto de vista	Do especificador e verificador
Contexto	Processos de negócio em Recursos Humanos na linguagem núcleo

A medição é um mecanismo para criar uma memória do trabalho realizado que ajuda a responder uma série de questões com a promulgação de qualquer processo. Ela ajuda no planejamento de projetos de apoio que permitem determinar os pontos fortes e fracos de processos e produtos. Fornece uma justificativa para adotar técnicas para avaliar a qualidade de processos e produtos específicos. Medição também ajuda na avaliação de impacto de uma determinada ação.

A abordagem de GQM baseia-se no pressuposto de que para se medir deve-se primeiro especificar metas. Embora a abordagem tenha sido originalmente usada para definir e avaliar as metas de um projeto específico em um determinado ambiente, a sua utilização foi ampliada para um contexto maior.

O resultado da aplicação de GQM é a especificação de um sistema de medição de direcionamento de um conjunto particular de problemas e um conjunto de regras para a interpretação dos dados de medição [Basili et al. (1994)]. O modelo de medição resultante tem três níveis:

- Nível conceitual (meta): uma meta é definida por um objeto. Objetos de medição são: produtos, processos e recursos.
- Operacional (pergunta): um conjunto de questões é utilizado para caracterizar a forma de avaliação. Perguntas caracterizam o objeto de medição (produto, processo, recurso) com relação a um problema de qualidade selecionado e determinam a sua qualidade do ponto de vista selecionado.
- Nível quantitativo (métrica): um conjunto de dados está associado a todas as perguntas para respondê-las de forma quantitativa. Os dados podem ser:
  - Objetivos: se eles dependem apenas do objeto que está sendo medido e não no ponto de vista a partir dos quais eles são tomados.
  - Subjetivos: se eles dependem tanto do objeto que está sendo medido, quanto do ponto de vista a partir do qual eles são tomados.

A Figura 5.4 a seguir ilustra um modelo GQM, que é uma estrutura hierárquica começando com um objetivo. O objetivo é refinado em várias questões, que normalmente quebram o problema nos seus componentes principais. Cada pergunta é então transformada em métricas, algumas delas, objetivas outras subjetivas. A mesma métrica pode ser utilizada a fim de responder a perguntas diferentes sob o mesmo objetivo.

Neste trabalho, desenvolvemos uma especificação de processos de negócio em Alloy e outra em PVS que forneceram especificações formais e no caso do PVS uma prova de corretude, boa-formação. Antes destas especificações, foi necessário aprender sobre

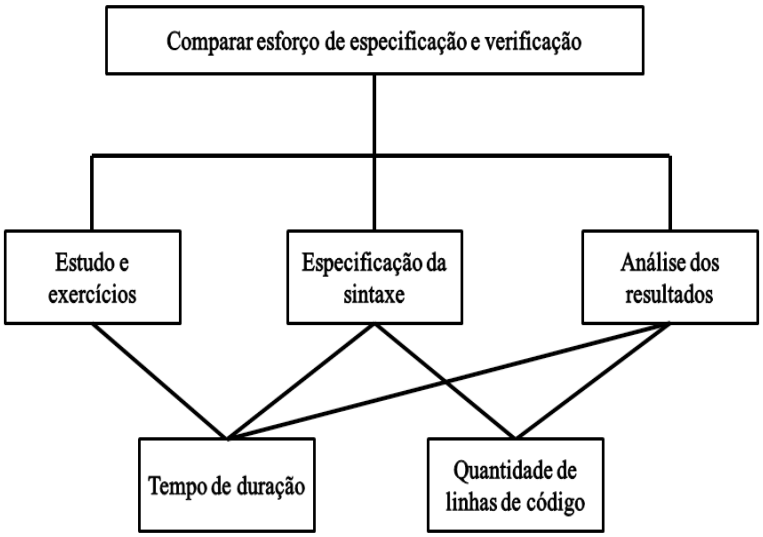

Figura 5.4: Árvore de avaliação GQM

sistemas de tipo e provas formais. Os conceitos de linhas de produtos de software, linha de produtos de processo de negócio e processo de negócio já haviam sido aprendidos anteriormente.

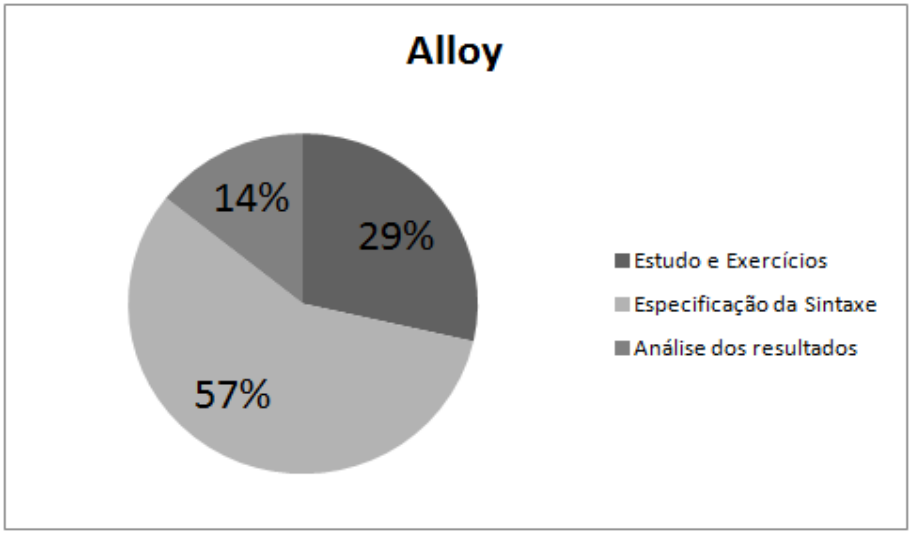

Figura 5.5: Gasto de tempo estimado para especificação Alloy

Na Figura 5.5, estimamos o tempo necessário para se familiarizar com as teorias básicas de Alloy, especificação de tipos em Alloy, especificação de regras em Alloy e análise dos resultados da ferramenta *Alloy Analyzer*. Na Figura 5.6, estimamos o tempo necessário para especificação de tipos em PVS, especificação de regras em PVS e prova de corretude em PVS. Por fim, na Figura 5.7, estimamos o tempo total do trabalho.

Ao analisar os resultados dos gráficos, é possível visualizar que o tempo dedicado ao estudo e aos exercícios foi similar nos dois gráficos (Figuras 5.5 e 5.6). Quanto à especificação, em Alloy tivemos um gasto superior ao PVS, devido ao estudo inicial da LPPN. E, também, tivemos um gasto de tempo inferior de especificação em PVS, pois

utilizamos as especificações do Alloy como base. Já o tempo de análise dos resultados em PVS foi muito superior ao tempo de análise em Alloy. Estas considerações podem ajudar a prever o esforço para trabalhos semelhantes.

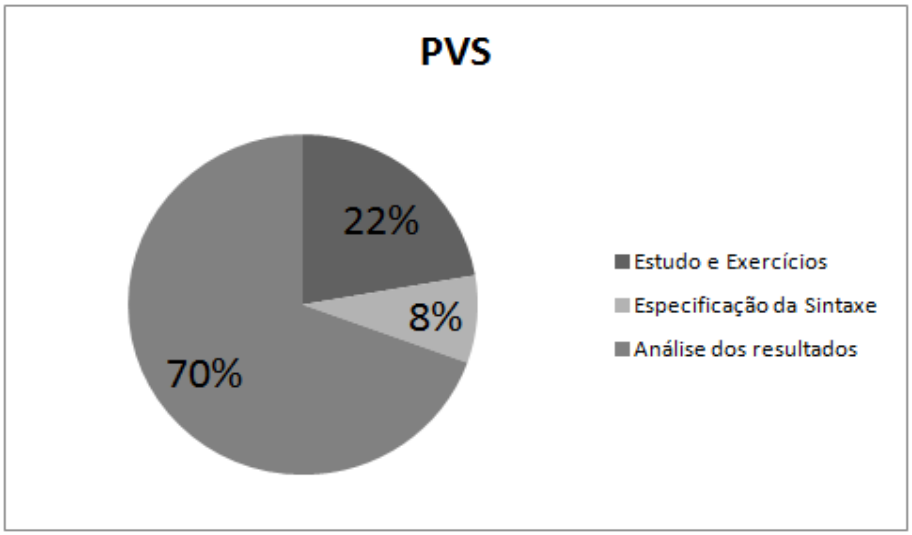

Figura 5.6: Gasto de tempo estimado para especificação e formalização PVS

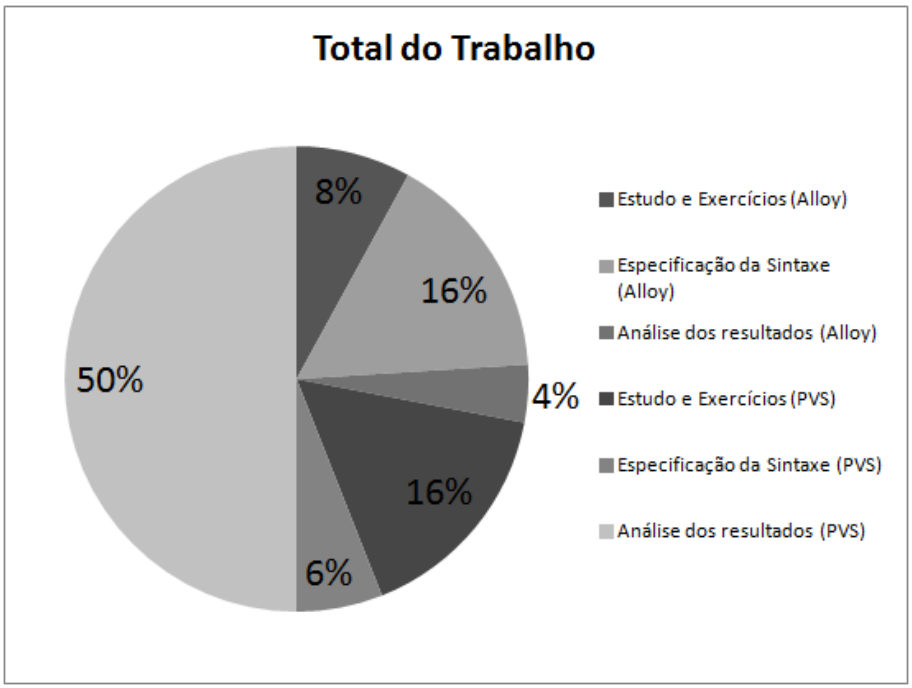

Figura 5.7: Gasto de tempo total estimado do trabalho

Em Alloy, primeiramente foi realizado um estudo com auxílio do livro Jackson (2011). Após 120 horas de estudo, foi iniciada a especificação de tipos e de regras de boa-formação. Na primeira parte da especificação em Alloy foram especificadas as assinaturas e as regras de boa-formação e durou cerca de 240 horas. Na segunda parte da especificação em Alloy foram realizadas as análises de resultados e durou cerca de 60 horas.

Em PVS, iniciamos com o estudo dos tutoriais e com exercícios. Foram necessários mais de 240 horas de estudo para familiarização com o PVS, o provador e os conceitos básicos. Além disso, foram gastos cerca de 90 horas para traduzir todos os tipos e as regras de boa-formação do Alloy para o PVS. No total foram necessárias cerca de 1.080 horas para obtenção de conhecimento e especificação da LPPN em PVS, que é cerca de 72 % do tempo total do trabalho.

De modo geral, foram gastas 330 horas com especificação e cerca de 750 horas com provas de teoremas. Sendo que a especificação não foi concluída antes do início das provas. Entretanto, enquanto as provas eram feitas, foram encontradas falhas na especificação. Em alguns casos, de falhas encontradas, foi necessário reescrever algumas definições.

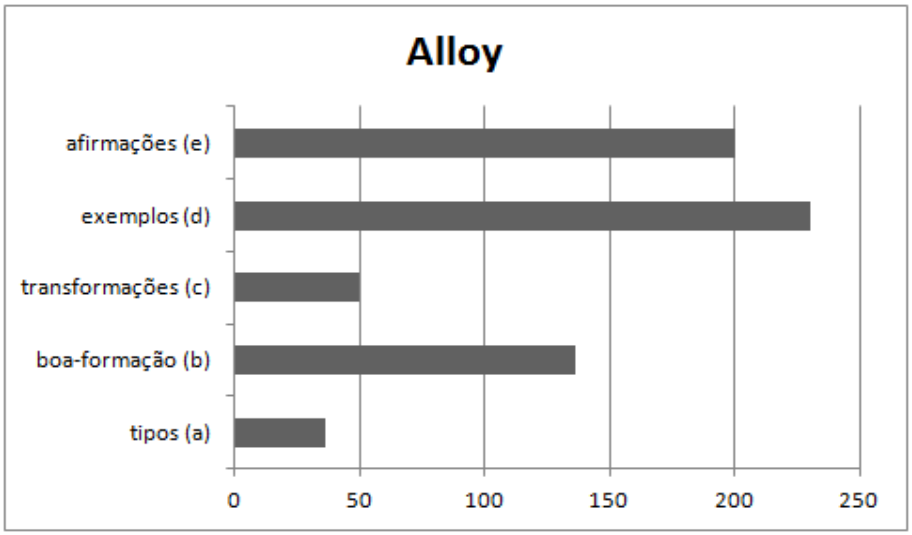

Figura 5.8: Quantidade de linhas especificadas em Alloy

Tabela 5.2: Quantidade de linhas especificadas em Alloy

	(a)	(b)	(c)	(d)	(e)	(f)
Quantidade de linhas	36	136	50	230	200	4

Uma visão mais detalhada do esforço da especificação da LPPN e das provas de teorema é visto na quantidade de linhas das especificações e verificações. A especificação em Alloy consiste em cinco arquivos contendo: (a) definições de tipo, (b) regras de boa-formação, (c) transformações, (d) um exemplo de LPPN, (e) especificação de afirmações, e (f) verificação (*model checking*). Na Figura 5.8, é possível visualizar o número total estimado de linhas escritas. E, na Tabela 5.2, é possível ver o número exato de linhas especificadas em Alloy.

Tabela 5.3: Quantidade de linhas especificadas em PVS

	(a)	(b)	(c)	(d)	(e)	(f)
Quantidade de linhas	52	52	66	78	360	10.170

A especificação e a formalização em PVS consiste em cinco arquivos contendo: (a) definições de tipo, (b) regras de boa-formação, (c) transformações, (d) um exemplo de

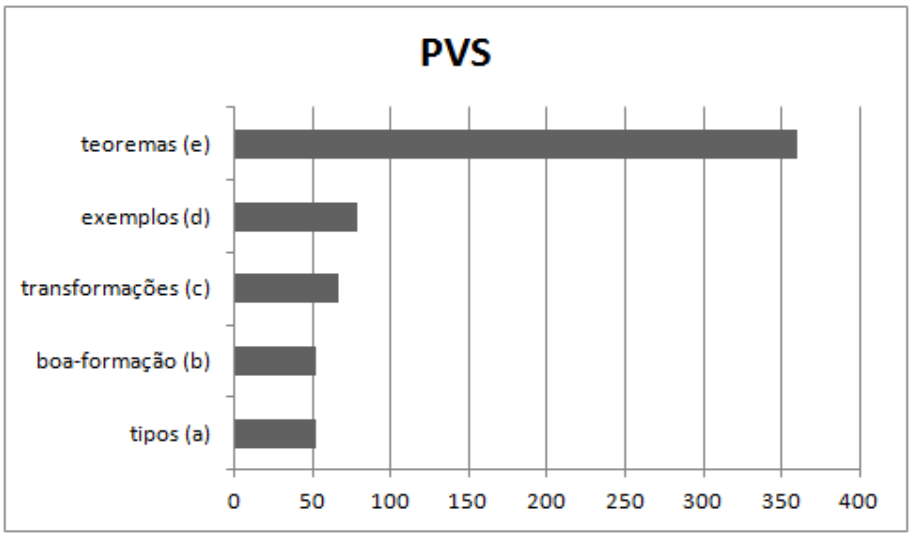

Figura 5.9: Quantidade de linhas especificadas em PVS

LPPN, (e) especificação de teoremas, e (f) verificação (prova). Na Figura 5.9, é possível visualizar o número total estimado de linhas escritas. Sendo que a quantidade total de linhas de verificação em Alloy e de prova de teoremas no PVS não foram colocadas nos gráficos pois ficam muito destoantes da quantidade de linhas da especificação. E, na Tabela 5.3, é possível ver o número exato de linhas especificadas em PVS.

5.3 Trabalhos Relacionados

Machado et al. (2011) caracteriza a variabilidade no processo de negócio e, em seguida, apresenta uma abordagem para gerenciar tal variabilidade. Como nosso trabalho, a abordagem empregada é composicional com base em aspectos. Ao contrário de nossa abordagem, eles não trataram de garantir a qualidade de uma maneira formal.

A abordagem ingênua de verificar a boa-formação de todos os programas individuais de uma linha de produtos não é viável por causa da explosão combinatória de variações do programa. Para resolver este problema, Apel et al. (2010) concebeu um sistema de tipos para uma versão simplificada da linguagem orientada. O sistema de tipos é satisfeito, ou seja, ele garante que todos os programas são bem tipados. Similar ao nosso trabalho, sua solução é geral no que diz respeito à propriedade sintática de instâncias da LP; além disso, a linguagem alvo é uma linguagem simplificada. Por outro lado, eles estão preocupados com os artefatos do programa, enquanto o nosso trabalho trata de um artefato de alto nível (processo de negócio) e sua prova foi realizada manualmente, enquanto o nosso foi realizado semi-automaticamente com um assistente de provas (PVS).

Um trabalho similar [Teixeira et al. (2011)] propõe uma abordagem automatizada para verificação de que a composição para LPSs são válidas com modelos de configuração de conhecimento explicitos. Porém, é uma abordagem voltada para LPSs de uma forma geral, neste trabalho temos uma abordagem para uma LPS específica, que é a LPPN.

Schaefer et al. (2011) forneceram uma base para a verificação composicional de tipo de linhas de produtos (LPs) de programas Java delta-orientados. Ao combinar os resultados

da análise dos módulos delta com a declaração da LP, eles mostraram que é possível estabelecer que todos os produtos da LP são bem tipados de acordo com o sistema de tipo Java. Sua abordagem é transformacional e trabalha com LP com verificação de tipo de objeto orientado em Java. Diferentemente, nossa abordagem é composicional e foca em um nível diferente de abstração (processo de negócio).

Rosa et al. (2011) propõem um método e um conjunto de ferramentas para o desenvolvimento de processos baseados no modelo de configuração de processos. Primeiro eles definiram o conceito de C-iEPC (uma extensão das cadeias de processos configuráveis orientadas a eventos, C-EPC) e sintaticamente correto C-iEPC. Em seguida, eles fornecem a definição e configuração da C-iEPC e mostram um algoritmo para individualizar C-iEPCs. Finalmente, eles provaram que o algoritmo é capaz de gerar o modelo de receber uma configuração sintaticamente correta. Ao contrário da nossa abordagem composicional, esta proposta emprega uma abordagem anotativa com o modo de ligação em tempo de execução. Esta especificação confunde com detalhes desnecessários, prejudicando a compreensão. Embora formal, sua abordagem emprega apenas a prova manual, enquanto a nossa emprega o provador de PVS.

Existem outras abordagens distintas da composicional utilizada neste trabalho. O trabalho de Aleixo et al. (2012), apresenta um estudo comparativo de abordagens, anotativa e composicional. Neste estudo eles concluem que a abordagem anotativa pode trazer muitas vantagens com relação à composicional, mas deixa a desejar na modularidade de elementos de processo associados a grupos específicos de variabilidades. O estudo ilustra que ambas abordagens têm suas próprias vantagens e limitações. Eles também concluem que os atuais mecanismos de composição do software EPF Composer pode ser integrado com abordagens de anotação especialmente para melhorar a modularidade de elementos associados ao processo variabilidades grossos e finos. Além disso, eles enfatizaram a grande necessidade de melhoria em detectar erros e na consistência das funcionalidades das atuais abordagens de anotação. A possível integração das abordagens composicional e anotativa podem efetivamente combinar os pontos fortes.

Thüm (2010) realizou um trabalho similar com o assistente de prova Coq para a linguagem Colored Featherweight Java (CFJ). Ele apresentou uma prova de boa-formação para a linguagem CFJ. Apresentou também um sistema de tipo simplificado para CFJ. Ele relatou os desafios de trabalhar com assistente de provas que exige um grande esforço para familiarização. A inicialização do trabalho pode ser difícil devido ao conhecimento que deve ser adquirido do provador para especificar definições e para criar os scripts de prova. Ele cita um exemplo difícil de prova que trabalha com indução. Ele comenta também sobre a vantagem de trabalhar com especificação formal e provas que é a verificação direta dos scripts de prova. Segundo o autor, esta vantagem dá a sensação de que foi realizado um bom trabalho pelos autores que não necessariamente são matemáticos. Um leitor de prova pode se concentrar nas definições e nos axiomas para descobrir se a especificação é útil no contexto que está sendo aplicada.

Em outro trabalho, Thüm et al. (2012) classificam-se análises que foram adaptadas a LPS. Os autores sustentam que uma variedade de análises é necessária para aumentar a confiabilidade do software e, em particular, dos produtos de uma LPS. O foco do trabalho é em análises que operam de forma estática e podem garantir a ausência de erros, como as seguintes: verificação de tipos (*Type Checking*), verificação de modelo (*Model Checking*), análise estática (*Static Analysis*) e provador de teoremas (*Theorem Proving*).

De acordo com esta classificação, nesta dissertação, foram feitas duas análises distintas: *Model Checking* e *Theorem Proving*. Em uma outra dimensão, os autores classificaram as abordagens de análise de linha de produtos existentes conforme a tentativa de redução do esforço de análise (a estratégia de análise). Distinguem-se três estratégias básicas: *product-based*, *Family-Based* e *Feature-Based*. O nosso trabalho pode ser classificado como *Family-Based*. A idéia geral da análise *Family-Based* é analisar artefatos de domínio e modelo de variabilidade a partir do qual conclui-se que algumas propriedades são válidas para todos os produtos. Uma análise é classificada como *Family-Based* se funciona apenas em artefatos de domínio e incorpora o conhecimento sobre combinações de características válidas.

5.4 Trabalhos Futuros

Como trabalho futuro, primeiramente finalizaremos a prova do teorema `WFM_EVAL`, para provar que a transformação `evaluateAfterAdvice` sempre gera produtos bem-formados. Além disto, propomos formalizar todas as transformações restantes da LPPN [Machado et al. (2011)]. Outro trabalho interessante a ser feito, é investigar as condições minimamente necessárias para preservar as propriedades de boa-formação e como estas interferem nas transformações. Pretende-se ainda especificar a semântica de outras construções de LPPN, como gateways, lanes e eventos, e quantificação no aspecto do subconjunto para aumentar a expressividade das LPPNs consideradas. Ainda como trabalho futuro, pretende-se explorar melhor a combinação de `model checking` e prova com assitente de provas, de modo a obter mais dados do custo-benefício e incrementar a automação da tradução de especificações em Alloy para PVS.

Referências

ABPMP. *Guide to the Business Process Management Common Body of Knowledge*. 2009.

Fellipe A. Aleixo, Marília Freire, Daniel Alencar, Edmilson Campos, e Uirá Kulesza. A comparative study of compositional and annotative modelling approaches for software process lines. In *SBES'12: XXVI Brazilian Symposium on Software Engineering*, 2012.

Sven Apel, Christian Kästner, Armin Größlinger, e Christian Lengauer. Type safety for feature-oriented product lines. *Automated Software Engineering*, 17:251–300, 2010. URL `http://dx.doi.org/10.1007/s10515-010-0066-8`. 10.1007/s10515-010-0066-8.

Victor R. Basili, Gianluigi Caldiera, e H. Dieter Rombach. The goal question metric approach. In *Encyclopedia of Software Engineering*. Wiley, 1994.

Rodrigo Bonifácio. *Modeling Software Product Line Variability in Use Case Scenarios - An Approach Based on Crosscutting Mechanisms*. PhD thesis, Universidade Federal de Pernambuco, fevereiro 2010.

Paul Clements e Linda Northrop. *Software Product Lines: Practices and Patterns*. John Wiley & Sons ltd., 2001.

Krysztof Czarnecki e Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley Professional, June 2000.

Jacob Daghlian. *Lógica e Álgebra de Boole*. 1995.

Marlon Dumas, Wil van der Aalst, e Arthur H. ter Hofstede. *Process-aware information systems: bridging people and software through process technology*. John Wiley and Sons, 2005.

Rohit Gheyi, Thiago Massoni, e Paulo Borba. A theory for feature models in alloy. In *First Alloy Workshop*, pages 71–80, 2006.

Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. London, revised edition edition, 2011.

John Jeston e Johan Nelis. *Business process management: practical guidelines to successful implementations*. Butterworth-Heinemann, 2006.

Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, e A. Spencer Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.

Charles Krueger. Easing the transition to software mass customization. In *Proceedings of the 4th International Workshop on Software Product-Family Engineering*, pages 282–293, Bilbao, Spain, October 2001.

Idarlan Machado, Rodrigo Bonifácio, Vander Alves, Lucinéia Turnes, e Giselle Machado. Managing variability in business processes: an aspect-oriented approach. In *Proceedings of the 2011 international workshop on Early aspects*, EA '11, pages 25–30, New York, NY, USA, 2011. ACM. doi: 10.1145/1960502.1960508. URL `http://doi.acm.org/10.1145/1960502.1960508`.

Congresso Nacional. *LEI N 8.112 (Acessado em novembro de 2012)*, 1990. URL `http://www.planalto.gov.br/ccivil_03/Leis/L8112compilado.htm`.

Object Mangement Group OMG. *Business Process Model and Notation (BPMN) (Acessado em novembro de 2012)*, 2009.

Sam Owre, John Rushby, N. Shankar, e David Stringer-Calvert. PVS: an experience report. In *Applied Formal Methods—FM-Trends 98*, volume 1641 of *Lecture Notes in Computer Science*, pages 338–345, Germany, 1998. Springer-Verlag.

Sam Owre, N. Shankar, John Rushby, e David Stringer-Calvert. *PVS Language Reference*, 2001.

Marcello La Rosa, Marlon Dumas, Arthur H.M. ter Hofstede, e Jan Mendling. Configurable multi-perspective business process models. *Information Systems*, 36(2):313–340, 2011. doi: 10.1016/j.is.2010.07.001.

Ina Schaefer, Lorenzo Bettini, e Ferruccio Damiani. Compositional type-checking for delta-oriented programming. In *Proceedings of the tenth international conference on Aspect-oriented software development*, AOSD '11, pages 43–56, New York, NY, USA, 2011. ACM. doi: 10.1145/1960275.1960283. URL `http://doi.acm.org/10.1145/1960275.1960283`.

SEGEP. *Simplification Guide (Acessado em outubro de 2012)*. URL `http://www.gespublica.gov.br/ferramentas/pasta.2010-04-26.1767784009`.

Leopoldo Teixeira, Paulo Borba, e Rohit Gheyi. Safe composition of configuration knowledge-based software product lines. In *Proceedings of the 2011 25th Brazilian Symposium on Software Engineering*, SBES '11, pages 263–272, Washington, DC, USA, 2011. IEEE Computer Society. doi: 10.1109/SBES.2011.15. URL `http://dx.doi.org/10.1109/SBES.2011.15`.

Thomas Thüm. A machine-checked proof for a product-line - aware type system. Master's thesis, University of Magdeburg, January 2010.

Thomas Thüm, Sven Apel, Christian Kästner, Martin Kuhlemann, Ina Schaefer, e Gunter Saake. Analysis strategies for software product lines. Technical Report FIN-004-2012, School of Computer Science, University of Magdeburg, April 2012.

Apêndice A

Especificação completa em Alloy

Este apêndice apresenta a especificação formal completa da LPPN em Alloy. Ele está dividido em seis seções. Na primeira seção, Seção B.1, são apresentados os tipos da LPPN que foram especificados como assinaturas em Alloy. Na Seção B.2, foram apresentadas as regras de boa-formação. Na Seção B.3, foi apresentado um exemplo de um modelo de LPPN (`BusinessProcessModel`) especificado. Posteriormente, na Seção B.4, foi apresentada a especificação das transformações. A seguir, na Seção A.5, possui a especificação das verificações feitas no Alloy de afirmações de teoremas do PVS. Por fim, na Seção A.6, foi apresentada a especificação das análises de afirmações feitas no Alloy.

A.1 Assinaturas

```
module tipos

sig Annotation {}
abstract sig Id {}
abstract sig idBP, idFO extends Id{}
sig Condition {}
sig Name {}
sig Value {}
sig Parameter {name: Name, value: Value}
abstract sig Pointcut {}
one sig Empty extends Pointcut{}
one sig PointcutConstrutor extends Pointcut {annotation: Annotation}
abstract sig FlowObjectType {}
one sig Activity extends FlowObjectType {}
abstract sig AdviceType {}
one sig Before, After, Around extends AdviceType {}
abstract sig FlowObject {}
one sig Start, End extends FlowObject {}
one sig Proceed extends FlowObject {}
sig FlowObjectComplexo extends FlowObject {
id: idFO,
    type: FlowObjectType,
```

```
annotations: set Annotation,
    parameters: set Parameter
}
sig Transition {
    startObject: FlowObject,
    endObject: FlowObject,
    condition: Condition
}
abstract sig ProcessType {}
one sig BasicProcess extends ProcessType{}
one sig Advice extends ProcessType {advType: AdviceType, pc: Pointcut}
sig BusinessProcess {
    pid: idBP,
    ptype: ProcessType,
    objects: set FlowObject,
    transitions: set Transition
}
sig BusinessProcessModel {processes: set BusinessProcess}
abstract sig Boolean {}
one sig True, False extends Boolean {}
```

A.2 Regras de boa-formação

```
module boaFormacao
open tipos

pred WF1[p: BusinessProcess]{
    Start+End in p.objects
}
run WF1 for 4

pred WF2[p: BusinessProcess]{
    all t: p.transitions | t.endObject !in Start and
                           t.endObject != none
}
run WF2 for 4

pred WF3[p: BusinessProcess]{
    all t: p.transitions | t.startObject !in End and
                           t.startObject != none
}
run WF3 for 4

pred WF4[p: BusinessProcess]{
    p.transitions.(startObject+endObject)  = p.objects
```

```
}
run WF4 for 4

pred WF5[p: BusinessProcess]{
    Start in p.*transitions.startObject
}
run WF5 for 4

pred WF6[p: BusinessProcess]{
    End in p.*transitions.endObject
}
run WF6 for 4

pred WF7[p: BusinessProcess]{
    all f: p.objects | f.type = Activity and f = FlowObjectComplexo and
    all t1, t2: p.transitions | t1.startObject = f and
                                t2.startObject = f => t1 = t2
}
run WF7 for 4

pred WF8[p: BusinessProcess]{
    p.ptype = Advice =>  not (some t: p.transitions |
                              t.startObject = Start and
                              t.endObject = End)
}
run WF8 for 4

pred WF9[p: BusinessProcess]{
    all f: p.objects | f = Start and
    all t1, t2: p.transitions | t1.startObject = f and
                                t2.startObject = f => t1 = t2
}
run WF9 for 4

pred wellFormed[p:BusinessProcess] {
    WF1[p] and WF2[p] and WF3[p] and WF4[p] and WF5[p] and
    WF6[p] and WF7[p] and WF8[p] and WF9[p]
}
run wellFormed for 4

pred wellFormedModel[p:BusinessProcessModel] {
    all bp: p.processes | wellFormed[bp]
}
run wellFormedModel

pred eq_pa (param1, param2: Parameter) {
```

```
    param1 != param2
    param1.value = param2.value
    param1.name = param2.name
}
run eq_pa for 8

pred eq_pas[pas1, pas2: set Parameter] {
    all x : pas1 | one y: pas2 | eq_pa[x, y]
    #pas1 = #pas2
}
run eq_pas for 8

pred eq_an[an1, an2: Annotation] {
    an1 = an2
}
run eq_an for 8

pred eq_ans[ans1, ans2: set Annotation] {
    all x : ans1 | one y: ans2 | eq_an[x, y]
    #ans1 = #ans2
}
run eq_ans for 8

pred eq_fo (obj1, obj2: FlowObject) {
    (
        (obj1 = obj2) and
        (obj1 = Start or obj1 = End)
    )   or
    (
        obj1 + obj2 !in  Start+End and
        obj1 != obj2  and
        obj1.id = obj2.id  and
        obj1.type = obj2.type  and
        eq_ans[obj1.annotations, obj2.annotations] and
        eq_pas[obj1.parameters, obj2.parameters]
    )
}
run eq_fo for 4

pred eq_tr (tr1, tr2: Transition) {
    tr1 != tr2
    tr1.condition = tr2.condition
    eq_fo[tr1.startObject, tr2.startObject]
    eq_fo[tr1.endObject, tr2.endObject]
}
run eq_tr for 2 BusinessProcessModel, 2 BusinessProcess,
```

```
                2 Annotation, 4 Id, 4 Condition, 2 Name,
                2 Parameter, 2 Value, 4 Transition,
                2 FlowObjectComplexo, 1 Start, 1 End

pred eq_fos(objs1,objs2: set FlowObject) {
    #objs1 = #objs2
    all x : objs1 | one y: objs2| eq_fo[x, y]
}
run eq_fos for 4

pred eq_trs[trs1, trs2: set Transition] {
    all x : trs1 | one y: trs2| eq_tr[x, y]
    #trs1 = #trs2
}
run eq_trs for 2 BusinessProcessModel, 2 BusinessProcess,
                2 Annotation, 4 Id, 4 Condition, 2 Name,
                2 Parameter, 2 Value, 4 Transition,
                2 FlowObjectComplexo, 1 Start, 1 End

pred eq_bp [bp1,bp2: BusinessProcess] {
    wellFormed[bp1]
    wellFormed[bp2]
    bp1 != bp2
    bp1.pid = bp2.pid
    bp1.ptype = bp2.ptype
    eq_fos[bp1.objects, bp2.objects]
    eq_trs[bp1.transitions, bp2.transitions]
}
run eq_bp for 4
```

A.3 Exemplo

```
module modelo2
open tipos as AS
open boaFormacao as PRED

one sig Aposentadoria in BusinessProcess {}
one sig Analisar in FlowObjectComplexo {}
one sig transition0, transition1 in Transition {}
one sig id0 in idBP {}
one sig id1 in idFO {}
one sig condition0, condition1 in Condition {}
one sig annotation0, annotation1 in Annotation {}
one sig parameter0 in Parameter {}
one sig name0 in Name{}
```

```
one sig value0 in Value{}
one sig Modelo in BusinessProcessModel {}

fact{
    Aposentadoria.ptype = Advice
    Advice.pc = PointcutConstrutor
    Advice.advType = After
    Aposentadoria.pid = id0
    Aposentadoria.objects = Start+End+Analisar
    Aposentadoria.transitions = transition0+transition1
    PointcutConstrutor.annotation = annotation1

    Analisar.type = Activity
    Analisar.id = id1
    Analisar.annotations = annotation0

    parameter0.name = name0
    parameter0.value = value0
    Analisar.parameters = parameter0

    transition0.condition = condition0
    transition1.condition = condition1

    transition0.startObject = Start
    transition0.endObject = Analisar
    transition1.startObject = Analisar
    transition1.endObject = End
}

fact{
    Modelo.processes = Aposentadoria
}
fact{
    transition0 !in transition1
    condition0 !in condition1
    annotation0 !in annotation1
}
fact{
    all disj bpm1, bpm2: BusinessProcessModel |
        no (bpm1.processes & bpm2.processes ) and
        no ((bpm1.processes.objects-(Start+End)) &
            (bpm2.processes.objects-(Start+End)))
}
pred show[]{}
run show for 8
```

A.4 Transformações

```
module transformacoes
open tipos
open modelo as MOD

fun SelectBusinessProcess[bpId: Id,
                          spl: BusinessProcessModel]:
    BusinessProcessModel{
    {
        bps: BusinessProcessModel | #bps.processes = 1
        and eq_bp[bps.processes,pid.bpId & spl.processes]
        and wellFormedModel[spl]
    }
}

fun matches [fo: FlowObject,
             pc: PointcutConstrutor]:
    set Boolean{
        {p: True | fo.annotations = pc.annotation} +
        {p: False | fo.annotations != pc.annotation}
    }

fun montaT[adv,bp: BusinessProcess]:
    set Transition{
        {b: bp.transitions |
            matches[ b.startObject, adv.ptype.pc] = False} ++
        {b: adv.transitions |
            b.startObject = FlowObjectComplexo and
            b.endObject = FlowObjectComplexo} +
        {c: Transition |
            c.startObject = {b: bp.transitions |
              matches[b.startObject, adv.ptype.pc] = True}.startObject
            and
            c.endObject   = {b: adv.transitions |
                                b.startObject in Start}.endObject} +
        {c: Transition |
            c.startObject = {b: adv.transitions |
                                b.endObject = End}.startObject and
            c.endObject   = {b: bp.transitions |
              matches[b.startObject, adv.ptype.pc] = True}.endObject}
}

fun evaluateAfterAdvice [adv, bp: BusinessProcess]:  BusinessProcess {
    {p:BusinessProcess |
        p.pid = bp.pid and
```

```
        p.ptype = bp.ptype and
        p.objects = {bp.objects} ++
                    {b: adv.objects | b != End and b != Start} and
        p.transitions = montaT[adv,bp]
    }
}
```

A.5 Verificações de afirmações de teoremas do PVS

```
module principal
open tipos as AS
open transformacoes as TRANS
open modelo as MOD

assert WFM_SBP{
    all i: idBP |
    all spl: BusinessProcessModel |
        wellFormedModel[spl]  and  i in spl.processes.pid
    => wellFormedModel[SelectBusinessProcess[i,spl]]
}
check WFM_SBP for 8

assert WFM_EVAL{
    all adv: BusinessProcess | wellFormed[adv] and
                               adv.ptype = Advice and
    all bp: BusinessProcess | wellFormed[bp] and
        some f: bp.objects | matches[f,adv.ptype.pc] = True and
    all disj obj1: adv.objects, obj2: bp.objects |
        obj1 = FlowObjectComplexo and
        obj2 = FlowObjectComplexo and
        obj1 !in obj2
    =>  wellFormed[evaluateAfterAdvice[adv,bp]]
}
check WFM_EVAL for 6

assert WFM_EVAL5{
    all adv: BusinessProcess | wellFormed[adv] and
                               adv.ptype = Advice and
    all bp: BusinessProcess | wellFormed[bp] and
        some f: bp.objects | matches[f,adv.ptype.pc] = True and
    all disj obj1: adv.objects, obj2: bp.objects |
        obj1 = FlowObjectComplexo and
        obj2 = FlowObjectComplexo and
        obj1 !in obj2
    =>  WF5[evaluateAfterAdvice[adv,bp]]
```

```
}
check WFM_EVAL5 for 6

fun BPT[adv,bp: BusinessProcess]: set Transition{
    {b: bp.transitions | adv.ptype = Advice and
        matches[b.startObject, adv.ptype.pc] = True}
}
fun FOBPA[adv,bp: BusinessProcess]: set FlowObject{
    {fo: bp.objects | fo in bp.*transitions.startObject and
    matches[bp.*transitions.startObject, adv.ptype.pc] = True}
}
fun FOBPE[adv,bp: BusinessProcess]: set FlowObject{
    {fo: bp.objects | fo in bp.*transitions.endObject and
    matches[bp.*transitions.startObject, adv.ptype.pc] = True}
}
assert WFM_EVAL5_05{
    all adv: BusinessProcess | wellFormed[adv] and
                               adv.ptype = Advice and
    all bp: BusinessProcess | wellFormed[bp] and
        some f: bp.objects | matches[f,adv.ptype.pc] = True and
    all disj obj1: adv.objects, obj2: bp.objects |
        obj1 = FlowObjectComplexo and
        obj2 = FlowObjectComplexo and
        obj1 !in obj2
    =>  evaluateAfterAdvice[adv,bp].objects =
        FOBPA[adv,bp] + FOBPE[adv,bp]
}
check WFM_EVAL5_05 for 8
```

A.6 Especificação de análises realizadas no Alloy

```
module principal
open tipos as AS
open transformacoes as TRANS
open modelo as MOD

pred pred_WFM_SBP(bpId:Id, spl: BusinessProcessModel){
    wellFormedModel[spl] and   bpId  in spl.processes.pid
    =>  wellFormedModel[SelectBusinessProcess[bpId,spl]]
}
run pred_WFM_SBP for 6

pred SelectWorks[bpId:Id, spl: BusinessProcessModel]{
    wellFormedModel[spl]  and bpId in spl.processes.pid and
    SelectBusinessProcess[bpId,spl] != none and
```

```
    spl = Modelo
}
run SelectWorks for 6 but 2 BusinessProcessModel

pred evaluateAfter(A, AC:BusinessProcess){
    evaluateAfterAdvice[A,AC] != none
    and A.ptype in Advice
    and A in Allowance -- apenas para teste
    and AC in AjudaCusto -- apenas para teste
}
run evaluateAfter for 8 but 1 BusinessProcessModel, 3 BusinessProcess
assert WFM_SBP{
    all i: idBP |
    all spl: BusinessProcessModel |
        wellFormedModel[spl]  and
        i in spl.processes.pid
    =>  wellFormedModel[SelectBusinessProcess[i,spl]]
}
check WFM_SBP for 8

assert WFM_EAA{
    all A, AC:BusinessProcess |
    all BPM : BusinessProcessModel |
        AC.ptype in BasicProcess
        and A.ptype in Advice
        and A in BPM.processes
        and AC in BPM.processes
        and wellFormedModel[BPM]
        and evaluateAfterAdvice[A,AC] != none
    =>  wellFormed[evaluateAfterAdvice[A,AC]]
}
check WFM_EAA for 8

assert WFM_EVAL{
    all adv, bp:BusinessProcess |
        bp.ptype in BasicProcess
        and adv.ptype in Advice
        and wellFormed[bp]
        and wellFormed[adv]
        and adv.objects not in bp.objects
        and evaluateAfterAdvice[adv,bp] != none
    =>  wellFormed[evaluateAfterAdvice[adv,bp]]
}
check WFM_EVAL for 8

assert WFM_EVAL2{
```

```
    all adv, bp:BusinessProcess |
    all BPM : BusinessProcessModel | adv.ptype in Advice
        and adv in BPM.processes
        and bp in BPM.processes
        and wellFormedModel[BPM]
        and adv.objects not in bp.objects
        and evaluateAfterAdvice[adv,bp] != none
    =>  wellFormed[evaluateAfterAdvice[adv,bp]]
}
check WFM_EVAL2 for 8 but 1 BusinessProcessModel, 3 BusinessProcess
```

Apêndice B

Especificação completa em PVS

Este apêndice apresenta a especificação formal completa da LPPN em PVS. Ele está dividido em sete seções. Na Seção B.1, são apresentados os tipos da LPPN que foram especificados em PVS. Na Seção B.2, são apresentadas as regras de boa-formação. Na Seção B.3, é apresentado um exemplo de um modelo de LPPN (`BusinessProcessModel`) especificado. Posteriormente, na Seção B.4, é apresentada a especificação das transformações. Na Seção B.5, são apresentadas as funções que definem quando há equivalência entre alguns tipos. A seguir, Seção A.5, a especificação das verificações feitas no Alloy de afirmações de teoremas do PVS. Por fim, nas Seções B.6 e B.7, são apresentados teoremas específicos e teoremas gerais, respectivamente.

B.1 Tipos

```
tipos: THEORY
 BEGIN

  Annotation: TYPE

  Id: TYPE

  idBP, idFO: TYPE FROM Id

  Condition: TYPE

  Name: TYPE

  Value: TYPE

  Parameter: DATATYPE
   BEGIN
    Parameter(Name: Name, Value: Value)
                 : Parameter?
   END Parameter
```

```
Pointcut: DATATYPE
 BEGIN
  Empty0: Empty0?
  CPC(annotation: Annotation): CPC?
 END Pointcut

FlowObjectType: DATATYPE
 BEGIN
  Activity: Activity?
 END FlowObjectType

AdviceType: DATATYPE
 BEGIN
  Before: Before?
  After: After?
  Around: Around?
 END AdviceType

FlowObject: DATATYPE
 BEGIN
  Start0: Start0?
  End0: End0?
  Proceed: Proceed?
  CFO(id: idFO, type0: FlowObjectType,
        annotations: set[Annotation],
        parameters: set[Parameter])
      : CFO?
 END FlowObject

Transition: DATATYPE
 BEGIN
  MkTransition(startObject: FlowObject,
                    endObject: FlowObject,
                    condition: Condition)
                  : MkTransition?
 END Transition

ProcessType: DATATYPE
 BEGIN
  BasicProcess: BasicProcess?
  Advice(advType: AdviceType, pc: Pointcut)
        : Advice?
 END ProcessType

BusinessProcess: DATATYPE
 BEGIN
```

```
    BusinessProcess(pid: idBP,
                    ptype: ProcessType,
                    objects: set[FlowObject],
                    transitions: set[Transition])
                  : BusinessProcess?
  END BusinessProcess

 BusinessProcessModel: DATATYPE
  BEGIN
   BPM(processes: set[BusinessProcess]): BusinessProcessModel?
  END BusinessProcessModel

END tipos
```

B.2 Regras de boa-formação

```
boa_formacao: THEORY
 BEGIN

  IMPORTING tipos, nocao_equivalencia

  extends(bp: BusinessProcess, origem: FlowObject)(fim: FlowObject): bool =
      (origem ∈ (objects(bp))) ∧
       (fim ∈ (objects(bp))) ∧
        ((origem = fim) ∨
            (∃ (t: Transition):
                 (t ∈ transitions(bp)) ∧
                  startObject(t) = origem ∧ endObject(t) = fim))

  extendsClosure(bp: BusinessProcess, origem: FlowObject)(fim: FlowOb-
ject): INDUCTIVE
          boolean =
          ((origem ∈ (objects(bp))) ∧ (fim ∈ (objects(bp)))) ∧
           ((extends(bp, origem)(fim)) ∨
               (∃ (meio: FlowObject):
                    (meio ∈ objects(bp)) ∧
                     extendsClosure(bp, origem)(meio) ∧
                      extends(bp, meio)(fim)))

  AXIOMA_Start: AXIOM
    ∀ (fo1, fo2: FlowObject):
      Start0?(fo1) ∧ Start0?(fo2) ⇒ fo1 = fo2

  AXIOMA_End: AXIOM
    ∀ (fo1, fo2: FlowObject): End0?(fo1) ∧ End0?(fo2) ⇒ fo1 = fo2
```

```
WF1(p: BusinessProcess): bool =
    (Start0 ∈ objects(p)) ∧ (End0 ∈ objects(p))

WF2(p: BusinessProcess): bool =
    (∀ (t: Transition):
        (t ∈ transitions(p)) ⇒ endObject(t) ≠ Start0)

WF3(p: BusinessProcess): bool =
    (∀ (t: Transition):
        (t ∈ transitions(p)) ⇒ startObject(t) ≠ End0)

WF4(p: BusinessProcess): bool =
    (∀ (t: Transition):
        (t ∈ transitions(p)) ⇒
          (startObject(t) ∈ objects(p)) ∧
            (endObject(t) ∈ objects(p)))

WF5(p: BusinessProcess): bool =
    (∀ (fo: FlowObject):
        (fo ∈ objects(p)) ⇒ extendsClosure(p, Start0)(fo))

WF6(p: BusinessProcess): bool =
    (∀ (fo: FlowObject):
        (fo ∈ objects(p)) ⇒ extendsClosure(p, fo)(End0))

WF7(p: BusinessProcess): bool =
    (∀ (fo: FlowObject):
        (fo ∈ objects(p)) ∧ CFO?(fo) ∧ type0(fo) = Activity ⇒
          (∀ (tt1, tt2: Transition):
              (tt1 ∈ transitions(p)) ∧
                (tt2 ∈ transitions(p)) ∧
                  startObject(tt1) = fo ∧ startObject(tt2) = fo
                  ⇒ EQ_TR(tt1, tt2)))

WF8(p: BusinessProcess): bool =
    Advice?(ptype(p)) ⇒ ¬ extends(p, Start0)(End0)

WF9(p: BusinessProcess): bool =
    (∀ (fo: FlowObject):
        (fo ∈ objects(p)) ∧ Start0?(fo) ⇒
          (∀ (tt1, tt2: Transition):
              (tt1 ∈ transitions(p)) ∧
                (tt2 ∈ transitions(p)) ∧
                  startObject(tt1) = fo ∧ startObject(tt2) = fo
                  ⇒ EQ_TR(tt1, tt2)))
```

```
  wellFormed(p: BusinessProcess): bool =
        WF9(p) ∧
         WF1(p) ∧
          WF2(p) ∧
           WF3(p) ∧
            WF4(p) ∧ WF5(p) ∧ WF6(p) ∧ WF7(p) ∧ WF8(p)

  wellFormedModel(p: BusinessProcessModel): bool =
        (∀ (bp: {p1: BusinessProcess | (p1 ∈ processes(p))}):
            wellFormed(bp))

END boa_formacao
```

B.3 Exemplo

```
modelo: THEORY
 BEGIN

  IMPORTING tipos

  obrigacaoA: AXIOM ∃ (x: Annotation): TRUE

  obrigacaoIBP: AXIOM ∃ (x: idBP): TRUE

  obrigacaoIFO: AXIOM ∃ (x: idFO): TRUE

  obrigacaoC: AXIOM ∃ (x: Condition): TRUE

  obrigacaoN: AXIOM ∃ (x: Name): TRUE;

  obrigacaoV: AXIOM ∃ (x: Value): TRUE;

  a0, a1: Annotation

  id0: idBP

  id1, id2: idFO

  c0, c1, c2: Condition

  v0, v1: Value

  n0, n1: Name
```

p_0: Parameter = Parameter(n_0, v_0)

p_1: Parameter = Parameter(n_1, v_1)

ans0: set[Annotation] = {a: Annotation | $a = a_0$}

ans1: set[Annotation] = {a: Annotation | $a = a_1$}

ps0: set[Parameter] = {p: Parameter | $p = p_0$}

ps1: set[Parameter] = {p: Parameter | $p = p_1$}

A: FlowObject = CFO(id1, Activity, ans0, ps0)

B: FlowObject = CFO(id2, Activity, ans1, ps1)

t_0: Transition = MkTransition(Start0, A, c_0)

t_1: Transition = MkTransition(A, B, c_1)

t_2: Transition = MkTransition(B, End0, c_2)

objs0: set[FlowObject] =
{fo: FlowObject |
fo = Start0 $\vee$ fo = A $\vee$ fo = B $\vee$ fo = End0}

ts0: set[Transition] =
{t: Transition | $t = t_0 \vee t = t_1 \vee t = t_2$}

bp0: BusinessProcess =
BusinessProcess(id0, BasicProcess, objs0,
{t: Transition |
$t = t_0 \vee t = t_1 \vee t = t_2$})

a_2: Annotation

id3: idBP

id4: idFO

c_3, c_4: Condition

v_2: Value

n_2: Name

p_2: Parameter = Parameter(n_2, v_2)

ans2: set[Annotation] = {a: Annotation | $a = a_2$}

ps2: set[Parameter] = {p: Parameter | $p = p_2$}

pc0: Pointcut = CPC(a_0)

C: FlowObject = CFO(id4, Activity, ans2, ps2)

t_3: Transition = MkTransition(Start0, C, c_3)

t_4: Transition = MkTransition(C, End0, c_4)

objs1: set[FlowObject] =
{fo: FlowObject | fo = Start0 $\vee$ fo = C $\vee$ fo = End0}

ts1: set[Transition] = {t: Transition | $t = t_3 \vee t = t_4$}

bp1: BusinessProcess =
BusinessProcess(id3, Advice(After, pc0), objs1, ts1)

bps0: set[BusinessProcess] =
{bp: BusinessProcess | bp = bp1 $\vee$ bp = bp0}

Modelo0: BusinessProcessModel = BPM(bps0)

id5: idBP

c_5, c_6: Condition

t_5: Transition = MkTransition(A, C, c_1)

t_6: Transition = MkTransition(C, B, c_4)

objs2: set[FlowObject] =
{fo: FlowObject |
fo = Start0 $\vee$ fo = A $\vee$ fo = B $\vee$ fo = C $\vee$ fo = End0}

ts2: set[Transition] =
{t: Transition | $t = t_0 \vee t = t_2 \vee t = t_5 \vee t = t_6$}

bp2: BusinessProcess = BusinessProcess(id0, BasicProcess, objs2, ts2)

bps1: set[BusinessProcess] = {bp: BusinessProcess | bp = bp2}

Modelo1 : BusinessProcessModel = BPM(bps1)

unicidadeID : AXIOM
$\neg$ id0 = id1 $\wedge$
$\neg$ id0 = id2 $\wedge$
$\neg$ id0 = id3 $\wedge$
$\neg$ id0 = id4 $\wedge$
$\neg$ id0 = id5 $\wedge$
$\neg$ id1 = id2 $\wedge$
$\neg$ id1 = id3 $\wedge$
$\neg$ id1 = id4 $\wedge$
$\neg$ id1 = id5 $\wedge$
$\neg$ id2 = id2 $\wedge$
$\neg$ id2 = id4 $\wedge$
$\neg$ id2 = id5 $\wedge$
$\neg$ id3 = id4 $\wedge$ $\neg$ id3 = id5 $\wedge$ $\neg$ id4 = id5

unicidadeA : AXIOM $\neg\ a_0 = a_1 \wedge \neg\ a_0 = a_2 \wedge \neg\ a_1 = a_2$

unicidadeN : AXIOM $\neg\ n_0 = n_1 \wedge \neg\ n_0 = n_2 \wedge \neg\ n_1 = n_2$

unicidadeV : AXIOM $\neg\ v_0 = v_1 \wedge \neg\ v_0 = v_2 \wedge \neg\ v_1 = v_2$

unicidadeC : AXIOM
$\neg\ c_0 = c_1 \wedge$
$\neg\ c_0 = c_2 \wedge$
$\neg\ c_0 = c_3 \wedge$
$\neg\ c_0 = c_4 \wedge$
$\neg\ c_0 = c_5 \wedge$
$\neg\ c_0 = c_6 \wedge$
$\neg\ c_1 = c_2 \wedge$
$\neg\ c_1 = c_3 \wedge$
$\neg\ c_1 = c_4 \wedge$
$\neg\ c_1 = c_5 \wedge$
$\neg\ c_1 = c_6 \wedge$
$\neg\ c_2 = c_3 \wedge$
$\neg\ c_2 = c_4 \wedge$
$\neg\ c_2 = c_5 \wedge$
$\neg\ c_2 = c_6 \wedge$
$\neg\ c_3 = c_4 \wedge$
$\neg\ c_3 = c_5 \wedge$
$\neg\ c_3 = c_6 \wedge$
$\neg\ c_4 = c_5 \wedge \neg\ c_4 = c_6 \wedge \neg\ c_5 = c_6$

END modelo

B.4 Transformações

```
transformacoes: THEORY
 BEGIN

  IMPORTING tipos, boa_formacao, nocao_equivalencia

  SelectBusinessProcess(spl: {sp: BusinessProcessModel | wellFormed-
Model(sp)},
                                bpId:
                                  {id: Id |
                                              ∃ (p: BusinessProcess):
                                                (p ∈ processes(spl)) ∧ pid(p) = id}):
        BusinessProcessModel =
      spl
        WITH [processes
                    := {bp: BusinessProcess |
                              (bp ∈ processes(spl)) ∧
                               pid(bp) = bpId}]

  matches(fo: FlowObject, pc: {c: Pointcut | CPC?(c)}): bool =
      IF (CFO?(fo))
        THEN ∃ (a: Annotation): (a ∈ annotations(fo)) ∧ a = annotation(pc)
      ELSE FALSE
      ENDIF



 BPT?(adv,bp: BusinessProcess)(x4: Transition):  bool =
   x4 ∈ {tx4:  Transition | transitions(bp)(tx4) ∧
   CFO?(startObject(tx4)) ∧ Advice?(ptype(adv)) ∧
   CPC?(pc(ptype(adv))) ∧ matches(startObject(tx4),pc(ptype(adv)))})

 ST?(adv: BusinessProcess)(x5: Transition): bool =
   x5 ∈ {tx5: Transition | transitions(adv)(tx5) ∧
   startObject(tx5) = Start0})

 ET?(adv: BusinessProcess)(x6: Transition): bool =
   x6 ∈ {tx6: Transition | transitions(adv)(tx6) ∧
   endObject(tx6) = End0}


  FO_adv?(adv: BusinessProcess)(fo: FlowObject): bool =
      objects(adv)(fo) ∧ (¬ fo = End0) ∧ (¬ fo = Start0)

  startTransition(adv: BusinessProcess): set[Transition] =
      {t: Transition |
```

```
            transitions(adv)(t) ∧ startObject(t) = Start0}

  disjuntosFOS(fos1, fos2: set[FlowObject]): bool =
      ∀ (fo1, fo2: FlowObject):
        CFO?(fo1) ∧ CFO?(fo2) ∧ (fo1 ∈ fos1) ∧ (fo2 ∈ fos2) ⇒
          ¬ EQ_FO(fo1, fo2)

  disjuntosTRS(trs1, trs2: set[Transition]): bool =
      ∀ (tr1, tr2: Transition):
        (tr1 ∈ trs1) ∧ (tr2 ∈ trs2) ⇒ ¬ EQ_TR(tr1, tr2)

  montaT1(adv: {a: BusinessProcess | wellFormed(a)},
            bp:
              {b: BusinessProcess |
                        wellFormed(b) ∧ disjuntosFOS(objects(b), ob-
jects(adv))}):
          set[Transition] =
      {t1: Transition |
           transitions(bp)(t1) ∧
            Advice?(ptype(adv)) ∧
             CPC?(pc(ptype(adv))) ∧
              (¬ matches(startObject(t1), pc(ptype(adv))))}

  montaT2(adv: {a: BusinessProcess | wellFormed(a)},
            bp:
              {b: BusinessProcess |
                        wellFormed(b) ∧ disjuntosFOS(objects(b), ob-
jects(adv))}):
          set[Transition] =
      {t2: Transition |
           transitions(adv)(t2) ∧
            CFO?(startObject(t2)) ∧ CFO?(endObject(t2))}

  montaT3(adv: {a: BusinessProcess | wellFormed(a)},
            bp:
              {b: BusinessProcess |
                        wellFormed(b) ∧ disjuntosFOS(objects(b), ob-
jects(adv))}):
          set[Transition] =
      {t3: Transition |
           ∃ (x1: (BPT?(adv, bp)), y1: (ST?(adv))):
             startObject(t3) = startObject(x1) ∧
              endObject(t3) = endObject(y1) ∧
               condition(t3) = condition(x1)}

  montaT4(adv: {a: BusinessProcess | wellFormed(a)},
```

```
            bp:
              {b: BusinessProcess |
                        wellFormed(b) ∧ disjuntosFOS(objects(b), ob-
jects(adv))}):
          set[Transition] =
        {t4: Transition |
            ∃ (x2: (ET?(adv)), y2: (BPT?(adv, bp))):
              startObject(t4) = startObject(x2) ∧
                endObject(t4) = endObject(y2) ∧
                  condition(t4) = condition(x2)}

  montaTT(adv: {a: BusinessProcess | wellFormed(a)},
            bp:
              {b: BusinessProcess |
                        wellFormed(b) ∧ disjuntosFOS(objects(b), ob-
jects(adv))}):
          set[Transition] =
        ((montaT1(adv, bp) ∪ montaT2(adv, bp)) ∪ (montaT3(adv, bp) ∪ montaT4(adv, bp)))

  montaT(adv: {a: BusinessProcess | wellFormed(a)},
           bp:
             {b: BusinessProcess |
                       wellFormed(b) ∧ disjuntosFOS(objects(b), ob-
jects(adv))}):
          set[Transition] =
        ((transitionsNotMatches ∪ innerTransitionsAdv) ∪ (transitionsBaseAdv ∪ transitionsAdvBase)
          WHERE transitionsNotMatches =
                  {t1: Transition |
                       transitions(bp)(t1) ∧
                         Advice?(ptype(adv)) ∧
                           CPC?(pc(ptype(adv))) ∧
                             (¬ matches(startObject(t1), pc(ptype(adv))))},
                innerTransitionsAdv =
                  {t2: Transition |
                       transitions(adv)(t2) ∧
                         CFO?(startObject(t2)) ∧ CFO?(endObject(t2))},
                transitionsBaseAdv =
                  {t3: Transition |
                       ∃ (x1: (BPT?(adv, bp)), y1: (ST?(adv))):
                         startObject(t3) = startObject(x1) ∧
                           endObject(t3) = endObject(y1) ∧ condi-
tion(t3) = condition(x1)},
                transitionsAdvBase =
                  {t4: Transition |
                       ∃ (x2: (ET?(adv)), y2: (BPT?(adv, bp))):
                         startObject(t4) = startObject(x2) ∧
```

```
                                endObject(t4) = endObject(y2) ∧
                                  condition(t4) = condition(x2)}

  evaluateAfterAdvice(adv: {a: BusinessProcess | wellFormed(a)},
                                bp:
                                  {b: BusinessProcess |
                                                wellFormed(b) ∧
                                                 disjuntosFOS(objects(b), ob-
jects(adv))}):
          BusinessProcess =
        bp
          WITH [pid := pid(bp),
                ptype := ptype(bp),
                objects := (objects(bp) ∪ FO_adv?(adv)),
                transitions := montaT(adv, bp)]

 END transformacoes
```

B.5 Noção de equivalência

```
nocao_equivalencia: THEORY
 BEGIN

  IMPORTING tipos

  EQ_FO(ff1, ff2: FlowObject): bool =
       (Start0?(ff1) ∧ Start0?(ff2)) ∨
        (End0?(ff1) ∧ End0?(ff2)) ∨
         (Proceed?(ff1) ∧ Proceed?(ff2)) ∨
          (CFO?(ff1) ∧
               CFO?(ff2) ∧
                id(ff1) = id(ff2) ∧
                 type0(ff1) = type0(ff2) ∧
                  parameters(ff1) = parameters(ff2) ∧
                   annotations(ff1) = annotations(ff2))

  EQ_TR(tt1, tt2: Transition): bool =
       startObject(tt1) = startObject(tt2) ∧
        endObject(tt1) = endObject(tt2) ∧
         condition(tt1) = condition(tt2)

  EQ_BP(bbp1, bbp2: BusinessProcess): bool =
       ptype(bbp1) = ptype(bbp2) ∧
        objects(bbp1) = objects(bbp2) ∧
         transitions(bbp1) = transitions(bbp2) ∧
```

pid(bbp1) = pid(bbp2)

END nocao_equivalencia

B.6 Especificação de teoremas específicos

teoremas_especificos: THEORY
BEGIN

IMPORTING tipos, transformacoes, modelo, nocao_equivalencia, boa_formacao

Teorema1: THEOREM wellFormed(bp0)

Teorema2: THEOREM wellFormed(bp1)

Teorema3: THEOREM wellFormed(bp2)

Teorema41: THEOREM
montaT1(bp1, bp0) = list2set[Transition]((:t_0, t_2:))

Teorema42: THEOREM empty?(montaT2(bp1, bp0))

Teorema43: THEOREM
montaT3(bp1, bp0) = list2set[Transition]((:t_5:))

Teorema44: THEOREM
montaT4(bp1, bp0) = list2set[Transition]((:t_6:))

Teorema45: THEOREM
montaTT(bp1, bp0) = transitions(evaluateAfterAdvice(bp1, bp0))

Teorema46: THEOREM
montaTT(bp1, bp0) = list2set[Transition]((:t_0, t_2, t_5, t_6:))

Teorema4: THEOREM EQ_BP(evaluateAfterAdvice(bp1, bp0), bp2)

Teorema5: THEOREM wellFormed(evaluateAfterAdvice(bp1, bp0))

Teorema6: THEOREM $\forall$ (f_1, f_2: FlowObject): $f_1 = f_2 \Rightarrow$ EQ_FO(f_1, f_2)

Teorema7: THEOREM
$\forall$ (f_1, f_2: FlowObject): ($\neg$ (EQ_FO(f_1, f_2))) $\Rightarrow f_1 \neq f_2$

Teorema8: THEOREM
$\forall$ (t_1, t_2: Transition): ($\neg$ (EQ_TR(t_1, t_2))) $\Rightarrow t_1 \neq t_2$

```
  simpleEA(bp11, bp22: BusinessProcess, fo1, fo2: FlowOb-
ject, tt1, tt2, tt3: Transition): bool =
       CFO?(fo2) ∧
        CFO?(fo1) ∧
         BasicProcess?(ptype(bp22)) ∧
          Advice?(ptype(bp11)) ∧
           (objects(bp11)(fo1)) ∧
            startObject(tt1) = fo1 ∧
             endObject(tt1) = End0 ∧
              startObject(tt2) = fo1 ∧
               endObject(tt2) = fo2 ∧
                startObject(tt3) = fo2 ∧
                 endObject(tt3) = End0 ∧
                  (transitions(bp11)(tt1)) ∧
                   (transitions(bp22)(tt2)) ∧
                    (transitions(bp22)(tt3)) ∧
                     (∀ (tt: {t: Transition | transitions(bp22)(t)}):
                          ¬ EQ_TR(tt1, tt))
                      ∧
                      (∀ (tt: {t: Transition | transitions(bp11)(t)}):
                           ¬ EQ_TR(tt2, tt))
                       ∧
                       (∀ (tt: {t: Transition | transitions(bp11)(t)}):
                            ¬ EQ_TR(tt3, tt))
                        ∧
                        (objects(bp11) ∪ list2set[FlowObject]((:fo2:))) =
                         objects(bp22)
                         ∧
                         (transitions(bp11) ∪ list2set[Transition]((:tt2, tt3:))) =
                          (transitions(bp22) ∪ list2set[Transition]((:tt1:)))
                          ∧
                          (∀ (fo: {f: FlowObject | objects(bp11)(f)}):
                               ¬ EQ_FO(fo2, fo))

  Lema1: LEMMA
    ∀ (bp11, bp22: BusinessProcess, fo1, fo2: FlowOb-
ject, tt1, tt2, tt3: Transition):
      (simpleEA(bp11, bp22, fo1, fo2, tt1, tt2, tt3) ∧ well-
Formed(bp11)) ⇒
       ((¬ EQ_TR(tt2, tt3)) ∧
          (∀ (t: {tr: Transition | transitions(bp11)(tr)}):
               (¬ EQ_TR(t, tt2)) ∧ (¬ EQ_TR(t, tt3))))

  Lema2: LEMMA
```

```
    ∀ (bp11, bp22: BusinessProcess, fo1, fo2: FlowOb-
ject, tt1, tt2, tt3: Transition):
      simpleEA(bp11, bp22, fo1, fo2, tt1, tt2, tt3) ∧ wellFormed(bp11) ⇒
       (∀ (t1, t2: Transition):
            transitions(bp22)(t1) ∧
             transitions(bp22)(t2) ∧ startObject(t1) = fo1 ∧ startOb-
ject(t2) = fo1
              ⇒ EQ_TR(t1, t2))

  Lema3: LEMMA
    ∀ (bp11, bp22: BusinessProcess, fo1, fo2: FlowOb-
ject, tt1, tt2, tt3: Transition):
      simpleEA(bp11, bp22, fo1, fo2, tt1, tt2, tt3) ∧ wellFormed(bp11) ⇒
       (∀ (t1, t2: Transition):
            transitions(bp22)(t1) ∧
             transitions(bp22)(t2) ∧ startObject(t1) = fo2 ∧ startOb-
ject(t2) = fo2
              ⇒ EQ_TR(t1, t2))

  Lema41: LEMMA
    ∀ (bp11, bp22: BusinessProcess, fo1, fo2, x: FlowOb-
ject, tt1, tt2, tt3: Transition):
      simpleEA(bp11, bp22, fo1, fo2, tt1, tt2, tt3) ∧
       wellFormed(bp11) ∧ objects(bp11)(x)
       ⇒ objects(bp22)(x)

  Lema4: LEMMA
    ∀ (bp11, bp22: BusinessProcess, fo1, fo2, x: FlowOb-
ject, tt1, tt2, tt3: Transition):
      ∀ (fo: FlowObject):
        simpleEA(bp11, bp22, fo1, fo2, tt1, tt2, tt3) ∧
         wellFormed(bp11) ∧ objects(bp11)(x)
         ⇒ objects(bp22)(x)

  Lema5: LEMMA
    ∀ (bp11, bp22: BusinessProcess, fo1, fo2: FlowOb-
ject, tt1, tt2, tt3: Transition):
      (∀ (fo: {f: FlowObject | objects(bp11)(f) ∧ objects(bp22)(f)}):
          simpleEA(bp11, bp22, fo1, fo2, tt1, tt2, tt3) ∧
           wellFormed(bp11) ∧ extendsClosure(bp11, Start0)(fo1)
           ⇒ extendsClosure(bp22, Start0)(fo1))

  Lema6: LEMMA
    ∀ (bp11, bp22: BusinessProcess, fo1, fo2: FlowOb-
ject, tt1, tt2, tt3: Transition):
      (∀ (fo: {f: FlowObject | objects(bp11)(f) ∧ objects(bp22)(f)}):
```

```
            simpleEA(bp11, bp22, fo1, fo2, tt1, tt2, tt3) ∧
              wellFormed(bp11) ∧ extendsClosure(bp11, Start0)(fo1)
              ⇒ extendsClosure(bp22, Start0)(fo1))
        ∧ simpleEA(bp11, bp22, fo1, fo2, tt1, tt2, tt3) ∧ wellFormed(bp11)
        ⇒ extendsClosure(bp22, Start0)(fo2)

  Lema61: LEMMA
    ∀ (bp11, bp22: BusinessProcess, fo1, fo2: FlowOb-
ject, tt1, tt2, tt3: Transition):
      simpleEA(bp11, bp22, fo1, fo2, tt1, tt2, tt3) ∧ wellFormed(bp11) ⇒
       extendsClosure(bp22, Start0)(fo2)

  Lema7: LEMMA
    ∀ (bp11, bp22: BusinessProcess, fo1, fo2: FlowOb-
ject, tt1, tt2, tt3: Transition):
      (∀ (fo: {f: FlowObject | objects(bp11)(f) ∧ objects(bp22)(f)}):
            simpleEA(bp11, bp22, fo1, fo2, tt1, tt2, tt3) ∧
              wellFormed(bp11) ∧ extendsClosure(bp11, Start0)(fo1)
              ⇒ extendsClosure(bp22, Start0)(fo1))
        ∧ simpleEA(bp11, bp22, fo1, fo2, tt1, tt2, tt3) ∧ wellFormed(bp11)
        ⇒ extendsClosure(bp22, Start0)(End0)

  Lema8: LEMMA
    ∀ (bp11, bp22: BusinessProcess, fo1, fo2: FlowOb-
ject, tt1, tt2, tt3: Transition):
      simpleEA(bp11, bp22, fo1, fo2, tt1, tt2, tt3) ⇒
       (∀ (f: FlowObject):
            objects(bp11)(f) ⇒ objects(bp22)(f))

  Lema9: LEMMA
    ∀ (bp11, bp22: BusinessProcess, fo1, fo2: FlowOb-
ject, tt1, tt2, tt3: Transition):
      simpleEA(bp11, bp22, fo1, fo2, tt1, tt2, tt3) ⇒
       (∀ (t: Transition):
            transitions(bp11)(t) ⇒
             transitions(bp22)(t) ∨ t = tt1)

  Lema10: LEMMA
    ∀ (bp11, bp22: BusinessProcess, fo1, fo2: FlowOb-
ject, tt1, tt2, tt3: Transition):
      simpleEA(bp11, bp22, fo1, fo2, tt1, tt2, tt3) ⇒
       objects(bp22)(fo2)

  Lema11: LEMMA
    ∀ (bp11, bp22: BusinessProcess, fo1, fo2: FlowOb-
ject, tt1, tt2, tt3: Transition):
```

```
        simpleEA(bp11, bp22, fo1, fo2, tt1, tt2, tt3) ⇒
          objects(bp22)(fo1)

  Lema12: LEMMA
    ∀ (bp11, bp22: BbusinessProcess, fo1, fo2: FlowOb-
ject, tt1, tt2, tt3: Transition):
        simpleEA(bp11, bp22, fo1, fo2, tt1, tt2, tt3) ⇒
          objects(bp11)(fo1)

  Lema13: LEMMA
    ∀ (bp11, bp22: BusinessProcess, fo1, fo2: FlowOb-
ject, tt1, tt2, tt3: Transition):
        (∀ (fo: {f: FlowObject | objects(bp11)(f) ∧ objects(bp22)(f)}):
             simpleEA(bp11, bp22, fo1, fo2, tt1, tt2, tt3) ∧ well-
Formed(bp11) ⇒
              extendsClosure(bp22, Start0)(fo))

  Lema14: LEMMA
    ∀ (bp11, bp22: BusinessProcess, fo1, fo2: FlowOb-
ject, tt1, tt2, tt3: Transition):
        simpleEA(bp11, bp22, fo1, fo2, tt1, tt2, tt3) ∧ wellFormed(bp11) ⇒
          extendsClosure(bp22, fo1)(End0)

  Lema15: LEMMA
    ∀ (bp11, bp22: BusinessProcess, fo1, fo2: FlowOb-
ject, tt1, tt2, tt3: Transition):
        simpleEA(bp11, bp22, fo1, fo2, tt1, tt2, tt3) ∧ wellFormed(bp11) ⇒
          extendsClosure(bp22, fo2)(End0)

  Lema16: LEMMA
    ∀ (bp11, bp22: BusinessProcess, fo1, fo2: FlowOb-
ject, tt1, tt2, tt3: Transition):
        ∀ (fo: {f: FlowObject | objects(bp11)(f) ∧ objects(bp22)(f)}):
           simpleEA(bp11, bp22, fo1, fo2, tt1, tt2, tt3) ∧ well-
Formed(bp11) ⇒
            extendsClosure(bp22, fo)(End0)

  Lema17: LEMMA
    ∀ (bp11, bp22: BusinessProcess, fo1, fo2: FlowOb-
ject, tt1, tt2, tt3: Transition):
        simpleEA(bp11, bp22, fo1, fo2, tt1, tt2, tt3) ∧ wellFormed(bp11) ⇒
          WF1(bp22)

  Lema18: LEMMA
    ∀ (bp11, bp22: BusinessProcess, fo1, fo2: FlowOb-
ject, tt1, tt2, tt3: Transition):
```

simpleEA(bp11, bp22, fo1, fo2, tt1, tt2, tt3) ∧ wellFormed(bp11) ⇒
WF2(bp22)

Lema19: LEMMA
∀ (bp11, bp22: BusinessProcess, fo1, fo2: FlowObject, tt1, tt2, tt3: Transition):
simpleEA(bp11, bp22, fo1, fo2, tt1, tt2, tt3) ∧ wellFormed(bp11) ⇒
WF3(bp22)

Lema20: LEMMA
∀ (bp11, bp22: BusinessProcess, fo1, fo2: FlowObject, tt1, tt2, tt3: Transition):
simpleEA(bp11, bp22, fo1, fo2, tt1, tt2, tt3) ∧ wellFormed(bp11) ⇒
WF4(bp22)

Lema21: LEMMA
∀ (bp11, bp22: BusinessProcess, fo1, fo2: FlowObject, tt1, tt2, tt3: Transition):
simpleEA(bp11, bp22, fo1, fo2, tt1, tt2, tt3) ∧ wellFormed(bp11) ⇒
WF5(bp22)

Lema211: LEMMA
∀ (bp11, bp22: BusinessProcess, fo1, fo2: FlowObject, tt1, tt2, tt3: Transition):
simpleEA(bp11, bp22, fo1, fo2, tt1, tt2, tt3) ∧ wellFormed(bp11) ⇒
WF5(bp22)

Lema22: LEMMA
∀ (bp11, bp22: BusinessProcess, fo1, fo2: FlowObject, tt1, tt2, tt3: Transition):
simpleEA(bp11, bp22, fo1, fo2, tt1, tt2, tt3) ∧ wellFormed(bp11) ⇒
WF6(bp22)

Lema221: LEMMA
∀ (bp11, bp22: BusinessProcess, fo1, fo2: FlowObject, tt1, tt2, tt3: Transition):
simpleEA(bp11, bp22, fo1, fo2, tt1, tt2, tt3) ∧ wellFormed(bp11) ⇒
WF6(bp22)

Lema23: LEMMA
∀ (bp11, bp22: BusinessProcess, fo1, fo2: FlowObject, tt1, tt2, tt3: Transition):
simpleEA(bp11, bp22, fo1, fo2, tt1, tt2, tt3) ∧ wellFormed(bp11) ⇒
WF7(bp22)

Lema24: LEMMA

```
    ∀ (bp11, bp22: BusinessProcess, fo1, fo2: FlowOb-
ject, tt1, tt2, tt3: Transition):
      simpleEA(bp11, bp22, fo1, fo2, tt1, tt2, tt3) ∧ wellFormed(bp11) ⇒
       WF8(bp22)

  Lema25: LEMMA
    ∀ (bp11, bp22: BusinessProcess, fo1, fo2: FlowOb-
ject, tt1, tt2, tt3: Transition):
      simpleEA(bp11, bp22, fo1, fo2, tt1, tt2, tt3) ∧ wellFormed(bp11) ⇒
       WF9(bp22)

  Teorema10: THEOREM
    ∀ (bp11, bp22: BusinessProcess, fo1, fo2: FlowOb-
ject, tt1, tt2, tt3: Transition):
      simpleEA(bp11, bp22, fo1, fo2, tt1, tt2, tt3) ∧ wellFormed(bp11) ⇒
       wellFormed(bp22)

END teoremas_especificos
```

B.7 Especificação de teoremas gerais

```
teoremas_gerais: THEORY
BEGIN

  IMPORTING tipos, transformacoes, boa_formacao, nocao_equivalencia

  WFM_SBP: THEOREM
    ∀ (spl: {sp: BusinessProcessModel | wellFormedModel(sp)},
        bpId:
          {id: Id |
                    ∃ (p: BusinessProcess):
                      (p ∈ processes(spl)) ∧ pid(p) = id}):
      wellFormedModel(SelectBusinessProcess(spl, bpId))

  BPT_singleton: AXIOM
    ∀ (adv, bp: BusinessProcess, fo: FlowObject):
    objects(bp)(fo) ⊃ singleton?[Transition](BPT?(adv, bp))

  ST_singleton: AXIOM
    ∀ (adv: BusinessProcess): singleton?[Transition]((ST?(adv)))



FOAnotado?(adv: {a: BusinessProcess | wellFormed(a) ∧
                                      Advice?(ptype(a)) ∧
                                      CPC?(pc(ptype(a)))},
```

```
            bp: {b: BusinessProcess | wellFormed(b) ∧
                                      disjuntosFOS(objects(b),objects(adv))})
  (f: FlowObject): bool =
    f ∈ {fo: FlowObject | objects(bp)(fo) ∧ CFO?(fo) ∧
    matches(fo,pc(ptype(adv))) ∧
    startObject(BPT?(adv,bp)) = fo})


FOBPA?(adv,bp: BusinessProcess)(inicio: FlowObject): bool =
  inicio ∈ {f: FlowObject | objects(bp)(f) ∧
  extendsClosure(bp,f)(startObject(BPT?(adv,bp)))})


FOBPE?(adv,bp: BusinessProcess)(fim: FlowObject): bool =
  fim ∈ {fo: FlowObject | objects(bp)(fo) ∧
  extendsClosure(bp,endObject(BPT?(adv,bp)))(fo)})


FOBPV?(adv,bp: BusinessProcess)(fim: FlowObject): bool =
  fim ∈ {fo: FlowObject | objects(bp)(fo) ∧
  extendsClosure(bp,startObject(BPT?(adv,bp)))(fo) ∧
  extendsClosure(bp,fo)(endObject(BPT?(adv,bp)))})


TRBPA?(adv,bp: BusinessProcess)(tr: Transition): bool =
  tr ∈ {ttr: Transition | transitions(bp)(ttr) ∧
  (FOBPA?(adv,bp)(startObject(ttr))) ∧
  (FOBPA?(adv,bp)(endObject(ttr)))})


TRBPE?(adv,bp: BusinessProcess)(tr: Transition): bool =
  tr ∈ {ttr: Transition | transitions(bp)(ttr) ∧
  (FOBPE?(adv,bp)(startObject(ttr)))} ∧
  (FOBPE?(adv,bp)(endObject(ttr)))


FO_start?(adv: BusinessProcess)(fo: FlowObject): bool =
  fo ∈ {ffo: FlowObject | objects(adv)(ffo) ∧
  extends(adv,Start0)(ffo) }


 Lema_teste: LEMMA
   ∀ (adv:
          {a: BusinessProcess |
                    wellFormed(a) ∧ Advice?(ptype(a)) ∧ CPC?(pc(ptype(a)))},
        bp:
          {b: BusinessProcess |
```

```
                        wellFormed(b) ∧ disjuntosFOS(objects(b), ob-
jects(adv))},
            f: (FOAnotado?(adv, bp))):
        singleton?((BPT?(adv, bp))) ⇒
         singleton?((FOAnotado?(adv, bp)))

  Lema_teste1: LEMMA
    ∀ (adv:
            {a: BusinessProcess |
                        wellFormed(a) ∧ Advice?(ptype(a)) ∧ CPC?(pc(ptype(a)))},
          bp:
            {b: BusinessProcess |
                        wellFormed(b) ∧ disjuntosFOS(objects(b), ob-
jects(adv))},
            f: (FOAnotado?(adv, bp))):
        singleton?((BPT?(adv, bp))) ∧
         startObject((singleton_elt[Transition]((BPT?(adv, bp))))) ≠
          endObject((singleton_elt[Transition]((BPT?(adv, bp)))))
         ⇒
         extends(bp, startObject(singleton_elt[Transition](BPT?(adv, bp))))
                  (endObject(singleton_elt[Transition]
                                    (BPT?(adv, bp))))

  Lema_teste2: LEMMA
    ∀ (adv:
            {a: BusinessProcess |
                        wellFormed(a) ∧ Advice?(ptype(a)) ∧ CPC?(pc(ptype(a)))},
          bp:
            {b: BusinessProcess |
                        wellFormed(b) ∧ disjuntosFOS(objects(b), ob-
jects(adv))},
            f: (FOAnotado?(adv, bp))):
        startObject(singleton_elt[Transition](BPT?(adv, bp))) = f

  Lema_teste4: LEMMA
    ∀ (adv:
            {a: BusinessProcess |
                        wellFormed(a) ∧ Advice?(ptype(a)) ∧ CPC?(pc(ptype(a)))},
          bp:
            {b: BusinessProcess |
                        wellFormed(b) ∧ disjuntosFOS(objects(b), ob-
jects(adv))},
            f: (FOAnotado?(adv, bp))):
        ∀ (f1, f2: {fo: FlowObject | objects(bp)(fo)}):
          extendsClosure(bp, Start0)(f1) ∧ extendsClosure(bp, Start0)(f2) ⇒
           extendsClosure(bp, f1)(f2) ∨ extendsClosure(bp, f2)(f1)
```

```
  Lema_teste3: LEMMA
    ∀ (adv:
            {a: BusinessProcess |
                    wellFormed(a) ∧ Advice?(ptype(a)) ∧ CPC?(pc(ptype(a)))},
          bp:
            {b: BusinessProcess |
                    wellFormed(b) ∧ disjuntosFOS(objects(b), ob-
jects(adv))},
          f: (FOAnotado?(adv, bp))):
      ∀ (fo: (FOBPA?(adv, bp))): ¬ FOBPE?(adv, bp)(fo)

  Lema_EVAL1: LEMMA
    ∀ (adv:
            {a: BusinessProcess |
                    wellFormed(a) ∧ Advice?(ptype(a)) ∧ CPC?(pc(ptype(a)))},
          bp:
            {b: BusinessProcess |
                    wellFormed(b) ∧ disjuntosFOS(objects(b), ob-
jects(adv))},
          f: (FOAnotado?(adv, bp))):
      WF1(evaluateAfterAdvice(adv, bp))

  Lema_EVAL2: LEMMA
    ∀ (adv:
            {a: BusinessProcess |
                    wellFormed(a) ∧ Advice?(ptype(a)) ∧ CPC?(pc(ptype(a)))},
          bp:
            {b: BusinessProcess |
                    wellFormed(b) ∧ disjuntosFOS(objects(b), ob-
jects(adv))},
          f: (FOAnotado?(adv, bp))):
      WF2(evaluateAfterAdvice(adv, bp))

  Lema_EVAL3: LEMMA
    ∀ (adv:
            {a: BusinessProcess |
                    wellFormed(a) ∧ Advice?(ptype(a)) ∧ CPC?(pc(ptype(a)))},
          bp:
            {b: BusinessProcess |
                    wellFormed(b) ∧ disjuntosFOS(objects(b), ob-
jects(adv))},
          f: (FOAnotado?(adv, bp))):
      WF3(evaluateAfterAdvice(adv, bp))

  Lema_EVAL4: LEMMA
```

```
    ∀ (adv:
            {a: BusinessProcess |
                    wellFormed(a) ∧ Advice?(ptype(a)) ∧ CPC?(pc(ptype(a)))},
          bp:
            {b: BusinessProcess |
                    wellFormed(b) ∧ disjuntosFOS(objects(b), ob-
jects(adv))},
          f: (FOAnotado?(adv, bp))):
      WF4(evaluateAfterAdvice(adv, bp))

  Lema_EVAL5_01_01: LEMMA
    ∀ (adv:
            {a: BusinessProcess |
                    wellFormed(a) ∧ Advice?(ptype(a)) ∧ CPC?(pc(ptype(a)))},
          bp:
            {b: BusinessProcess |
                    wellFormed(b) ∧ disjuntosFOS(objects(b), ob-
jects(adv))},
          f: (FOAnotado?(adv, bp))):
      ∀ (f: FlowObject):
        objects(bp)(f) ⇒
         objects(evaluateAfterAdvice(adv, bp))(f)

  Lema_EVAL5_01_02: LEMMA
    ∀ (adv:
            {a: BusinessProcess |
                    wellFormed(a) ∧ Advice?(ptype(a)) ∧ CPC?(pc(ptype(a)))},
          bp:
            {b: BusinessProcess |
                    wellFormed(b) ∧ disjuntosFOS(objects(b), ob-
jects(adv))},
          f: (FOAnotado?(adv, bp))):
      ∀ (t: {tr: Transition | transitions(bp)(tr)}):
        ¬ transitions(adv)(t)

  Lema_EVAL5_01_03: LEMMA
    ∀ (adv:
            {a: BusinessProcess |
                    wellFormed(a) ∧ Advice?(ptype(a)) ∧ CPC?(pc(ptype(a)))},
          bp:
            {b: BusinessProcess |
                    wellFormed(b) ∧ disjuntosFOS(objects(b), ob-
jects(adv))},
          f: (FOAnotado?(adv, bp))):
      ∀ (t: (TRBPA?(adv, bp))):
        transitions(evaluateAfterAdvice(adv, bp))(t)
```

```
Lema_EVAL5_07: LEMMA
  ∀ (adv:
       {a: BusinessProcess |
            wellFormed(a) ∧ Advice?(ptype(a)) ∧ CPC?(pc(ptype(a)))},
     bp:
       {b: BusinessProcess |
            wellFormed(b) ∧ disjuntosFOS(objects(b), ob-
jects(adv))},
     f: (FOAnotado?(adv, bp))):
    ∀ (f1:
         {f: FlowObject |
              endObject((singleton_elt[Transition]((ST?(adv))))) = f}):
      extendsClosure(evaluateAfterAdvice(adv, bp), Start0)(f1)

Lema_EVAL5_08: LEMMA
  ∀ (adv:
       {a: BusinessProcess |
            wellFormed(a) ∧ Advice?(ptype(a)) ∧ CPC?(pc(ptype(a)))},
     bp:
       {b: BusinessProcess |
            wellFormed(b) ∧ disjuntosFOS(objects(b), ob-
jects(adv))},
     f: (FOAnotado?(adv, bp))):
    ∀ (f1:
         {f: FlowObject |
              endObject((singleton_elt[Transition]((BPT?(adv, bp))))) =
                f}):
      extendsClosure(evaluateAfterAdvice(adv, bp), Start0)(f1)

Lema_EVAL5_01: LEMMA
  ∀ (adv:
       {a: BusinessProcess |
            wellFormed(a) ∧ Advice?(ptype(a)) ∧ CPC?(pc(ptype(a)))},
     bp:
       {b: BusinessProcess |
            wellFormed(b) ∧ disjuntosFOS(objects(b), ob-
jects(adv))},
     f: (FOAnotado?(adv, bp))):
    ∀ (ff: (FOBPA?(adv, bp))):
      extendsClosure(bp, Start0)(ff) ⇒
       extendsClosure(evaluateAfterAdvice(adv, bp), Start0)(ff)

Lema_EVAL5_02: LEMMA
  ∀ (adv:
       {a: BusinessProcess |
```

```
                        wellFormed(a) ∧ Advice?(ptype(a)) ∧ CPC?(pc(ptype(a)))},
          bp:
            {b: BusinessProcess |
                        wellFormed(b) ∧ disjuntosFOS(objects(b), ob-
jects(adv))},
            f: (FOAnotado?(adv, bp))):
       ∀ (f1: (FO_adv?(adv))):
         extendsClosure(evaluateAfterAdvice(adv, bp), Start0)(f1)

  Lema_EVAL5_03: LEMMA
    ∀ (adv:
            {a: BusinessProcess |
                        wellFormed(a) ∧ Advice?(ptype(a)) ∧ CPC?(pc(ptype(a)))},
          bp:
            {b: BusinessProcess |
                        wellFormed(b) ∧ disjuntosFOS(objects(b), ob-
jects(adv))},
            f: (FOAnotado?(adv, bp))):
       ∀ (f1: (FOBPE?(adv, bp))):
         extendsClosure(evaluateAfterAdvice(adv, bp), Start0)(f1)

  Lema_EVAL5_04_03: LEMMA
    ∀ (adv:
            {a: BusinessProcess |
                        wellFormed(a) ∧ Advice?(ptype(a)) ∧ CPC?(pc(ptype(a)))},
          bp:
            {b: BusinessProcess |
                        wellFormed(b) ∧ disjuntosFOS(objects(b), ob-
jects(adv))},
            f: (FOAnotado?(adv, bp))):
       FOBPE?(adv, bp)(End0) ∧
        FOBPA?(adv, bp)(Start0) ∧
         FOBPA?(adv, bp)(f) ∧
          FOBPE?(adv, bp)
                (endObject(singleton_elt[Transition]
                                    (BPT?(adv, bp))))

  Lema_EVAL5_04_08: LEMMA
    ∀ (adv:
            {a: BusinessProcess |
                        wellFormed(a) ∧ Advice?(ptype(a)) ∧ CPC?(pc(ptype(a)))},
          bp:
            {b: BusinessProcess |
                        wellFormed(b) ∧ disjuntosFOS(objects(b), ob-
jects(adv))},
            f: (FOAnotado?(adv, bp))):
```

```
        empty?(FOBPV?(adv, bp))

    Lema_EVAL5_04: LEMMA
      ∀ (adv:
            {a: BusinessProcess |
                    wellFormed(a) ∧ Advice?(ptype(a)) ∧ CPC?(pc(ptype(a)))},
          bp:
            {b: BusinessProcess |
                    wellFormed(b) ∧ disjuntosFOS(objects(b), ob-
jects(adv))},
          f: (FOAnotado?(adv, bp))):
        objects(bp) = ((FOBPA?(adv, bp)) ∪ (FOBPE?(adv, bp)))

    Lema_EVAL5_041: LEMMA
      ∀ (adv:
            {a: BusinessProcess |
                    wellFormed(a) ∧ Advice?(ptype(a)) ∧ CPC?(pc(ptype(a)))},
          bp:
            {b: BusinessProcess |
                    wellFormed(b) ∧ disjuntosFOS(objects(b), ob-
jects(adv))},
          f: (FOAnotado?(adv, bp))):
        singleton?((BPT?(adv, bp))) ⇒
         objects(bp) =
          ((FOBPA?(adv, bp)) ∪ (FOBPE?(adv, bp)))

    Lema_EVAL5_042: LEMMA
      ∀ (adv:
            {a: BusinessProcess |
                    wellFormed(a) ∧ Advice?(ptype(a)) ∧ CPC?(pc(ptype(a)))},
          bp:
            {b: BusinessProcess |
                    wellFormed(b) ∧ disjuntosFOS(objects(b), ob-
jects(adv))},
          f: (FOAnotado?(adv, bp))):
        singleton?((BPT?(adv, bp))) ⇒
         (∀ (f: {fo: FlowObject | objects(bp)(fo)}):
            FOBPA?(adv, bp)(f) ∨ FOBPE?(adv, bp)(f))

    Lema_EVAL5_05: LEMMA
      ∀ (adv:
            {a: BusinessProcess |
                    wellFormed(a) ∧ Advice?(ptype(a)) ∧ CPC?(pc(ptype(a)))},
          bp:
            {b: BusinessProcess |
```

```
                        wellFormed(b) ∧ disjuntosFOS(objects(b), ob-
jects(adv))},
            f: (FOAnotado?(adv, bp))):
        objects(evaluateAfterAdvice(adv, bp)) =
          ((FOBPA?(adv, bp)) ∪ ((FO_adv?(adv)) ∪ (FOBPE?(adv, bp))))

  Lema_EVAL5: LEMMA
    ∀ (adv:
            {a: BusinessProcess |
                        wellFormed(a) ∧ Advice?(ptype(a)) ∧ CPC?(pc(ptype(a)))},
          bp:
            {b: BusinessProcess |
                        wellFormed(b) ∧ disjuntosFOS(objects(b), ob-
jects(adv))},
            f: (FOAnotado?(adv, bp))):
        WF5(evaluateAfterAdvice(adv, bp))

  Lema_EVAL6: LEMMA
    ∀ (adv:
            {a: BusinessProcess |
                        wellFormed(a) ∧ Advice?(ptype(a)) ∧ CPC?(pc(ptype(a)))},
          bp:
            {b: BusinessProcess |
                        wellFormed(b) ∧ disjuntosFOS(objects(b), ob-
jects(adv))},
            f: (FOAnotado?(adv, bp))):
        WF6(evaluateAfterAdvice(adv, bp))

  Lema_EVAL7: LEMMA
    ∀ (adv:
            {a: BusinessProcess |
                        wellFormed(a) ∧ Advice?(ptype(a)) ∧ CPC?(pc(ptype(a)))},
          bp:
            {b: BusinessProcess |
                        wellFormed(b) ∧ disjuntosFOS(objects(b), ob-
jects(adv))},
            f: (FOAnotado?(adv, bp))):
        WF7(evaluateAfterAdvice(adv, bp))

  Lema_EVAL8: LEMMA
    ∀ (adv:
            {a: BusinessProcess |
                        wellFormed(a) ∧ Advice?(ptype(a)) ∧ CPC?(pc(ptype(a)))},
          bp:
            {b: BusinessProcess |
```

wellFormed(b) $\wedge$ disjuntosFOS(objects(b), objects(adv))},
f: (FOAnotado?(adv, bp))):
WF8(evaluateAfterAdvice(adv, bp))

Lema_EVAL9: LEMMA
$\forall$ (adv:
{a: BusinessProcess |
wellFormed(a) $\wedge$ Advice?(ptype(a)) $\wedge$ CPC?(pc(ptype(a)))},
bp:
{b: BusinessProcess |
wellFormed(b) $\wedge$ disjuntosFOS(objects(b), objects(adv))},
f: (FOAnotado?(adv, bp))):
WF9(evaluateAfterAdvice(adv, bp))

WFM_EVAL: THEOREM
$\forall$ (adv:
{a: BusinessProcess |
wellFormed(a) $\wedge$ Advice?(ptype(a)) $\wedge$ CPC?(pc(ptype(a)))},
bp:
{b: BusinessProcess |
wellFormed(b) $\wedge$ disjuntosFOS(objects(b), objects(adv))},
f: (FOAnotado?(adv, bp))):
wellFormed(evaluateAfterAdvice(adv, bp))

END teoremas_gerais