FORMALIZAÇÃO DA LINGUAGEM GUARANÁ DSL

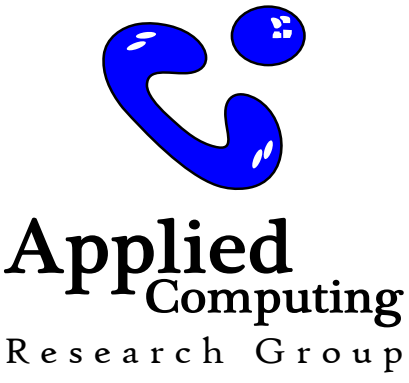
UMA ABORDAGEM MATEMÁTICA PARA ESPECIFICAÇÃO FORMAL DA SINTAXE ABSTRATA UTILIZANDO NOTAÇÃO Z

MAURI JOSÉ KLEIN

UNIJUÍ

DISSERTAÇÃO DE MESTRADO

ORIENTADOR: DR. SANDRO SAWICKI
COORIENTADOR: DRA. FABRÍCIA C. ROOS FRANTZ



MARÇO, 2015

First published in March 2015 by
Applied Computing Research Group - GCA
Department of Exact Sciences and Engineering
Rua Lulu Ilgenfritz, 480 - São Geraldo
Ijuí, 98700-000, Brazil.

Categorias (ACM 1998): D.1.2 [Automatic Programming]; D.1.3 [Concurrent Programming]; D.2.6 [Programming Environments]: Integrated environment, Programmer workbench; D.2.10 [Design]: Representation;

Financiamento: A pesquisa desenvolvida neste trabalho teve suporte da bolsa de mestrado concedida pela CAPES.

UNIJUÍ

A Comissão Examinadora, abaixo assinada, _____ a dissertação entitulada " Formalização da Linguagem Guaraná DSL: Uma Abordagem Matemática para Especificação Formal da Sintaxe Abstrata Utilizando Notação Z", elaborada por Mauri José Klein, como requisito parcial para a obtenção do título de Mestre em Modelagem Matemática.

_____

Dr. Sandro Sawicki
UNIJUÍ

_____ _____

Dra. Fabrícia C. Roos Frantz
UNIJUÍ

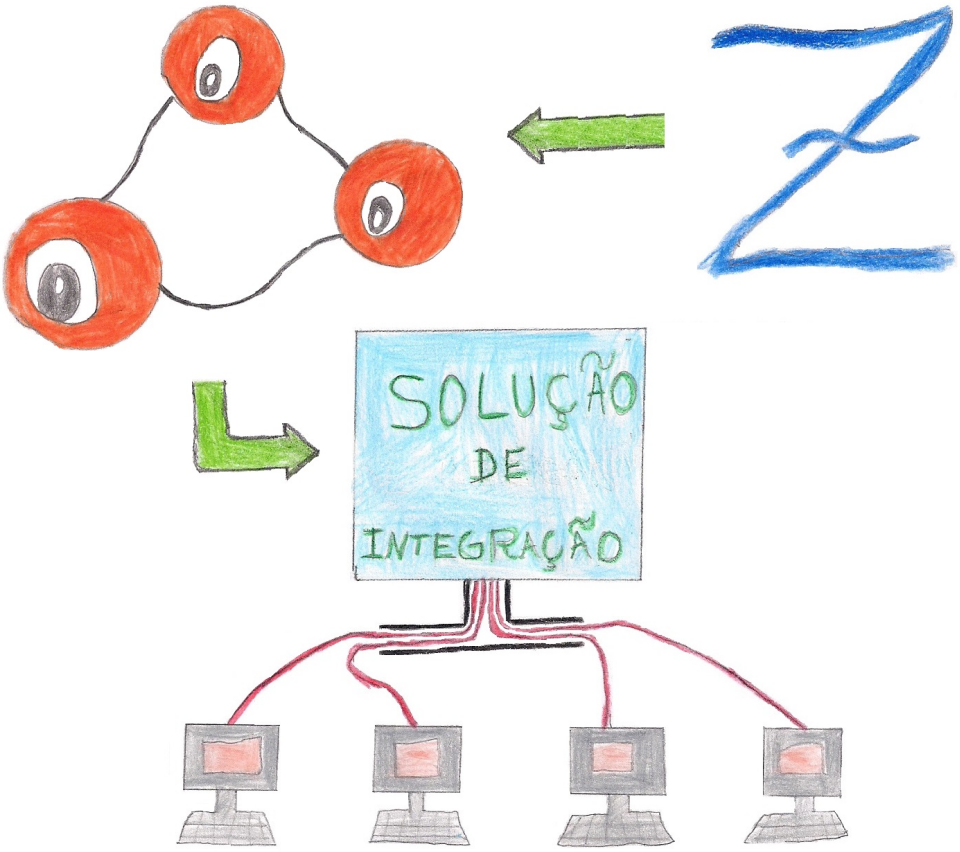
Dr. Daniel Welfer
UNIPAMPA

_____ _____

Dr. Rafael Zancan Frantz
UNIJUÍ

Dr. Manuel Binelo
UNIJUÍ

Ijuí, _____ de _____ de _____.



Formalização do Guaraná DJ por Bruno, 7 anos de idade.

Dedico este trabalho à minha esposa Marlise Inês
aos meus filhos Bruno Daniel e Keila Caroline.

Conteúdo

Índice de figuras

Índice de tabelas

Agradecimentos

A gratidão é um fruto de grande cultura;
não se encontra em gente vulgar.

Samuel Johnson, Escritor e pensador inglês (1709-1784)

Além de dedicar todo este trabalho a minha família, não posso deixar de agradecer aos meus filhos **Bruno Daniel** e **Keila Caroline** e à minha esposa **Marlise**. Sou muito grato pela compreensão, mas principalmente pelo apoio que me deram durante a período em que estive envolvido nesta pesquisa.

Agradeço também ao meu professor orientador **Dr. Sandro Sawicki**. Sempre o tive como referência desde a Graduação. E agora, nesta nova etapa, me ajudou muito para que o desafio fosse vencido com êxito.

A minha co-orientadora professora **Dra. Fabrícia Carneiro Roos Frantz**. Sempre muito dedicada e preocupada em proporcionar condições para que o trabalho progredisse.

Ao professor **Dr. Rafael Zancan Frantz**. Amplo conhecedor do assunto, auxiliou muito para viabilizar esta pesquisa.

Aos professores coordenadores do Mestrado em Modelagem Matemática **Dr. Paulo Sausen** e **Dra. Airam Sausen.**

Aos demais professores do Mestrado.

Aos meus colegas.

Obrigado!

Resumo

Se, a princípio, a ideia não é absurda,
então não há esperança para ela.

Albert Einstein, Físico teórico alemão (1879-1955)

A Tecnologia da Informação (TI) fornece suporte às organizações empresariais, proporcionando agilidade e qualidade aos seus processos de negócio. Este suporte é fornecido por aplicações que compõe o ecossistema de software que, de uma maneira geral, não apresenta as características necessárias à integração, dificultando a reutilização. Neste sentido, a Integração de Aplicações Empresariais (EAI) concentra-se na concepção e implementação de soluções de integração. A tecnologia Guaraná DSL é uma das ferramentas que fornecem este suporte, porém, ela difere das outras propostas por incluir um sistema de monitoramento que pode ser configurado usando uma linguagem baseada em regras para gerar soluções com tolerância a falhas. No entanto, Guaraná DSL ainda não está formalizado, e por esta razão, não é possível validar as regras escritas por engenheiros de software utilizando a linguagem baseada em regras. Além disso, não é possível gerar automaticamente as regras com base na semântica de uma solução de integração. Esta pesquisa propõe um estudo detalhado da tecnologia Guaraná DSL com enfase no mecanismo de tolerância a falhas, além do estudo e comparação dos métodos formais: Notação Z, B, Alloy, RSL, Redes de Petri, levando em consideração as características do Guaraná. Utilizando os metamodelos UML com restrições escritas em OCL da linguagem de domínio específico da tecnologia Guaraná encontrados na literatura, foi proposto um modelo matemático com a especificação formal da sintaxe abstrata e posterior validação. Para a especificação formal e validação utilizou-se a Notação Z por meio da ferramenta Z-EVES.

Abstract

If at first the idea is not absurd,
then there is no hope for it.

Albert Einstein, German theoretical physicist (1879-1955)

The Information Technology (IT) provides support to business organizations, providing agility and quality to its business processes. This support is provided by applications that composes the software ecosystem that, in general, does not have characteristics required for integration, making it difficult to reuse. In this sense, the Enterprise Application Integration - EAI focuses on the design and implementation of integration solutions. The Guaraná DSL technology is one of the tools that provide this support, but it differs from other proposals to include a monitoring system includes a monitoring system that can be configured using a rule-based language to endow solutions with fault-tolerance. However, Guaraná DSL is not yet formalized, and for this reason, it can not validate the rules written by software engineers using the rule-based language. In addition is not possible automatically generate rules based on the semantics of an integration solution. This research proposes a detailed study of Guaraná DSL technology with emphasis on fault tolerance mechanism in addition to the study and comparison of formal methods: Z notation, B, Alloy, RSL, Petri Nets, taking into account the characteristics of Guaraná. Using the UML metamodel with OCL constraints written in the domain specific language of Guaraná technology found in the literature, we propose a mathematical model with the formal specification of abstract syntax and subsequent validation. For the formal specification and validation was used the Z notation through the Z/EVES tool.

Parte I

Prefácio

Capítulo 1

Introdução

Todo começo é difícil.

Provérbio alemão

Esse trabalho apresenta a especificação formal da sintaxe abstrata da linguagem de domínio específico da tecnologia de integração de aplicações Guaraná. Neste contexto, o presente capítulo aborda, na Seção §1.1, o contexto desta pesquisa, seguido pela Seção §1.2 que discute a motivação deste trabalho. A Seção §1.3 descreve o objetivo geral e os objetivos específicos deste trabalho. A Seção §1.4 apresenta a metodologia seguida para alcançar os objetivos propostos. Na Seção §1.5 são descritas as contribuições resultantes deste trabalho. E por último, na Seção §1.6 é apresentada a estrutura desta dissertação.

1.1 Contexto da Pesquisa

A utilização de sistemas de informação através da Tecnologia da Informação (TI) vem ganhando cada vez mais destaque no mundo globalizado, proporcionando benefícios aos seus usuários. Aplicações para smartphones, computadores pessoais, comércio eletrônico e prestação dos mais diversos tipos de serviços, são alguns exemplos do emprego destes sistemas.

Geralmente os sistemas de informação são associados a utilização de programas de computador. No entanto, isto não está correto. Uma definição para sistemas de informação é dada por Laudon e Laudon [64] que os definem como sendo um conjunto de componentes trabalhando com a finalidade de coletar, recuperar, processar, armazenar e distribuir informações facilitando o planejamento, o controle, a coordenação, a análise e o processo de tomada de decisões em uma organização empresarial.

Outra definição similar atribui aos sistemas de informação papel importante na tomada de decisão em uma organização empresarial. Todo sistema que guarda dados e gera informação, mesmo sem a utilização de recurso computacional caracteriza-se como sistema de informação [94].

Audy e outros [8] destacam que existem diferentes abordagens que uma organização pode optar para disponibilizar seus sistemas de informação. Considera, no entanto, que as mesmas devem atender às necessidades de suporte ao controle e à integração das operações, à tomada de decisão e à obtenção de vantagens competitivas.

Esta abordagem também é dada por Rezende [94], que cita alguns benefícios que as organizações buscam alcançar através dos sistemas de informação: suporte a tomada de decisões, valor agregado aos produtos (bens e serviços), oportunidade de negócios e aumento da rentabilidade, mais segurança nas informações, menos erros, mais precisão, redução de custos e desperdícios.

Porém, alcançar estes benefícios não é simples. De acordo com Rezende [94], os sistemas de informação encontrados com frequência nas organizações apresentam características que oferecem complexidade de análise. O autor destaca que estes sistemas geralmente possuem um grande volume de dados e informações, muitos usuários ou clientes envolvidos, interligação de diversas técnicas e tecnologias, contexto abrangente, mutável e dinâmico, entre outras.

Devido às dificuldades apresentadas, a utilização de ferramentas e aplicações computacionais passou a ser mais intensa, com o objetivo de facilitar o armazenamento de dados e informações, a análise dos mesmos e a posterior tomada de decisões. Por meio de aplicações computacionais, os sistemas de informação oferecem melhor desempenho e agilidade em decisões, fornecendo a "informação certa, na hora certa" [86]. Rezende [94] considera ser muito difícil elaborar sistemas de informação sem utilizar a tecnologia da informação através de aplicações computacionais, considerando a complexidade e as necessidades das organizações.

Rezende [93] define uma aplicação computacional como sendo um conjunto de comandos, instruções ou ordens inseridas pelo usuário em um computador, com o objetivo de resolver problemas e desenvolver atividades ou tarefas específicas, relacionadas aos negócios da organização ao qual ele se destina.

Atualmente, com a capacidade de processamento e armazenamento de computadores e servidores, as organizações empresariais buscam agilizar, qualificar e dar suporte aos seus processos de negócio através da utilização de aplicações que compõe o seu ecossistema de software. Este ecossistema normalmente é heterogêneo. Aplicações distintas constituem este ecossistema, sejam elas de departamentos diferentes da própria organização, ou oriundas de uma fusão empresarial, por exemplo [80].

De maneira geral as aplicações não foram projetadas para operarem suas funções de forma integrada. Tonsig [109] considera que a ausência de integração gera retrabalho, e, com isso, a possibilidade de inconsistência de dados na organização aumenta consideravelmente, sem contar que erros de transcrição são bastante comuns nestes casos.

Frantz [34] considera que a integração de aplicações é necessária, porém não é uma tarefa trivial, e precisa ser realizada, na maioria dos casos, por meio de recursos, tais como as bases de dados, arquivos de dados, filas de mensagens e interfaces.

Este fator pode gerar alguns problemas que devem ser considerados:

- o código fonte das aplicações pode estar inacessível para ajustes e alterações;

- a alta complexidade dos códigos;

- um possível comprometimento da segurança das aplicações, devido a alterações das funcionalidades originais das aplicações;

- o tempo necessário para ajustar as funcionalidades.

Estes problemas motivaram a busca por uma nova forma de proceder a integração destas aplicações. Surge então o conceito de integração de aplicações empresariais.

Segundo Linthicum [70] a área de integração de aplicações empreariais busca oferecer compartilhamento irrestrito de dados e processos de negócios entre todas as aplicações conectadas e fontes de dados na empresa.

Segundo Hohpe e Woolf [49], a integração de aplicações fornece metodologias e ferramentas para desenvolver e implementar soluções de integração. O objetivo de uma solução de integração é justamente manter uma série de dados e funcionalidades das aplicações em sincronia ou desenvolver novas funcionalidades sobre as já existentes, sem que as aplicações tenham que ser alteradas e não sejam modificadas pela solução. Uma solução de integração é implantada no ecossistema de software como uma nova aplicação que fornece a seus usuários uma visão de alto nível das aplicações integradas com a qual eles podem interagir conforme representado na Figura §1.1.

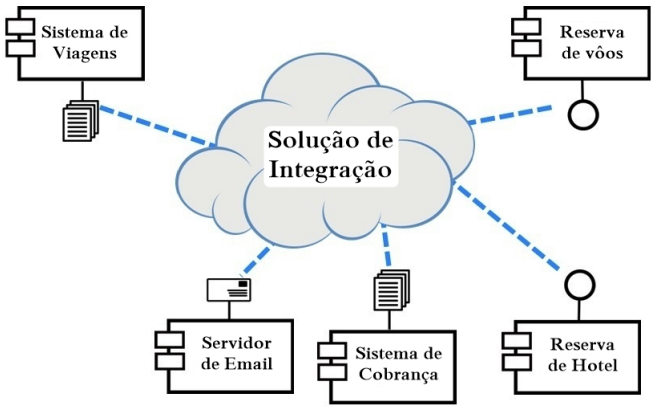


Figura 1.1: *Aplicação de uma Solução de Integração*

A grande demanda por soluções de integração, justificada pela presença massiva de ecossistemas de software heterogêneos nas organizações, constituiu um ambiente desafiador para concepção e desenvolvimento de tecnologias para este fim. Camel [52], Spring Integration [31], Mule [28] e Guaraná [36], são algumas das tecnologias com suporte para projeto e implementação de soluções de integração.

Projetadas para expressar as instruções em um determinado espaço ou domínio, as tecnologias para modelar soluções de integração possuem uma linguagem de domínio específico, através da criação de uma linguagem própria para a mesma. Através dela, é possível que engenheiros de software façam uma representação visual da solução, com alto nível de abstração, tornando mais fácil a sua compreensão, evitando uma abordagem de mais baixo nível e mais complexa proporcionada pelas linguagens de programação [36].

Outro fator relevante em uma tecnologia de integração de aplicações é que as soluções geradas a partir da mesma sejam tolerantes a falhas. É a partir de mecanismos de tolerância a falhas que aplicações ou soluções de integração são capazes de continuar provendo seus serviços mesmo na ocorrência de falhas. Considerando que tanto uma aplicação como uma solução de integração pode falhar é importante proporcionar mecanismos para o monitoramento de erros. Camel, Spring Integration, e Mule fornecem um mecanismo de detecção de erros baseada principalmente na utilização de try-catch [45]. A tecnologia Guaraná se diferencia por possuir um mecanismo baseado em um monitor externo para detecção de erros, que pode ser configurado utilizando uma linguagem baseada em regras definidas por engenheiros de software ou geradas automaticamente [34].

O mecanismo de tolerância a falhas do Guaraná está dividido em quatro etapas: *event reporting, error monitoring, error diagnosing* and *error recovering. Event reporting*: reporta eventos de uma solução. *Error monitoring*: etapa onde os eventos são armazenados e analisados para encontrar possíveis erros. Quando um erro é detectado, uma notificação é criada e enviada para o *error diagnosing*, cujo objetivo é identificar a causa do erro, as mensagens e as partes envolvidas. *Error recovering* é a fase de recuperação do erro [34].

A análise e detecção de possíveis falhas em uma solução de integração é de fundamental importância para este mecanismo. Nesta etapa define-se o que é considerado falha para a solução, e a sua origem deve ser diagnosticada, buscando uma solução. Estabelecer critérios, nesta etapa, que ofereçam corretude, completude, consistência, concisão e clareza representa uma boa base para o mecanismo. Estes atributos são características da formalização [83].

No entanto, a tecnologia Guaraná ainda carece de formalização. É o formalismo que torna possível validar as regras escritas por engenheiros de software utilizando a linguagem baseada em regras. Com a validação destas regras pode-se garantir que todas as possibilidades de falha em uma determinada solução de integração sejam cobertas. Outra possibilidade vislumbrada com a formalização é gerar automaticamente o conjunto de regras com base na semântica da solução de integração.

A formalização consiste em três fazes. A primeira fase é a*especificação formal* que define e descreve com maior rigor e clareza as propriedades de um sistema [60]. A segunda fase é constituída pela *verificação* que tem como finalidade validar a especificação foram. De maneira geral a verificação é realizada com a utilização de ferramentas que podem provar automaticamente teoremas simples, verificar a sintaxe, a semântica e o domínio, além de auxiliar na conferência das provas mais complicadas [83]. A terceira fase é a *implementação* que consiste na transformação da especificação em código fonte, expressa em alguma linguagem de programação [83].

A tecnologia Guaraná é composta de uma linguagem de domínio específico para projetar soluções de integração e um sistema de execução que permite a implementação e execução de soluções de integração, além de possuir um mecanismo de tolerância a falhas baseado em regras, que difere das demais tecnologias [36]. Além disso, a tecnologia Guaraná foi criada por pesquisadores que atuam no Grupo de Pesquisa em Computação Aplicada (GCA), mesmo grupo no qual a presente dissertação foi desenvolvida. Em função desta proximidade, optou-se por utilizar a tecnologia Guaraná como base nessa dissertação com o objetivo de contribuir para sua evolução no contexto da tolerância a falhas. Considerando que Guaraná ainda não foi formalizado, propõe-se neste trabalho fazer a especificação formal da sintaxe abstrata da linguagem de domínio específico do Guaraná, utilizando para isto a Notação Z com posterior validação.

1.2 Motivação

Os sistemas de informação estão presentes em grande parte das organizações através de aplicações que dão suporte aos processos de negócio. Estas aplicações compõe um ecossistema de software, geralmente heterogêneo, onde as tecnologias para integração se mostram potencialmente efetivas para este ambiente. A necessidade de fazer com que aplicações distintas forneçam suas funcionalidades de forma integrada, fomentou o surgimento de várias tecnologias de integração de aplicações. Dentre estas tecnologias o Guaraná tem características importantes para concepção de soluções de integração tolerantes a falhas. O mecanismo responsável pelo monitoramento, detecção e recuperação do erro é dividido em quatro etapas e está baseado na definição de regras, de forma automática ou escritas por um engenheiro de software. No entanto, para que estas regras possam ser construídas de forma automática ou validadas se inseridas manualmente, e que todas as possíveis falhas de uma solução sejam cobertas em um determinado

domínio, é necessário que a linguagem de domínio específico esteja formalizada. A especificação formal da sintaxe abstrata do Guaraná é o primeiro passo para o processo de formalização. Essa dissertação de mestrado está motivada em definir com rigor matemático todas as características da tecnologia Guaraná, descritas pelo modelo conceitual, por meio da especificação formal, e posterior validação do modelo.

1.3 Objetivos

A formalização de uma linguagem de domínio específico representa um passo importante para a evolução da linguagem. Neste sentido, esta dissertação tem os seguintes objetivos:

1.3.1 Geral

Propôr um modelo matemático com a especificação formal da sintaxe abstrata da linguagem de domínio específico da tecnologia Guaraná, a fim de contribuir para a formalização da mesma, tornando possível a validação das regras escritas por engenheiros de software ou gerá-las de forma automática.

1.3.2 Específicos

- Contextualizar os ambientes empresariais em relação aos sistemas de informação e as aplicações que compõe o seu ecossistema de software, a fim de apresentar os ambientes onde a integração é necessária.
- Destacar a importância das tecnologias de integração de aplicações empresariais em um ambiente heterogêneo nas organizações empresariais, a fim de demonstras as facilidades que as mesmas proporcionam aos usuários.
- Apresentar as especificidades da linguagem de domínio específico da tecnologia Guaraná buscando um entendimento claro e conciso a fim de modelar de forma integra e correta.
- Apresentar os conceitos de tolerância a falhas e do mecanismo de detecção de erro da tecnologia Guaraná, a fim de aplicá-los na especificação formal da sintaxe abstrata da linguagem de domínio específico da tecnologia Guaraná.

- Estudar os métodos formais a partir da literatura com a finalidade de apresentar a importância da formalização e da validação para sistemas de informação e principalmente para soluções de integração.

- Identificar os métodos formais que apresentam características desejáveis ao contexto do problema através de um framework de comparação, buscando contemplar as características apresentadas pela linguagem de domínio específico da tecnologia Guaraná.

- Modelar, através do método formal escolhido, a sintaxe abstrata da linguagem de domínio específico da tecnologia Guaraná, a fim de apresentar um modelo matemático preciso.

- Apresentar técnicas de validação para que sejam caracterizadas em relação ao contexto e utilizadas na validação do modelo.

- Validar o modelo formal proposto para a sintaxe abstrata da linguagem de domínio específico da tecnologia Guaraná através de técnicas específicas, como verificação de modelos e a prova de teoremas.

1.4 Metodologia

Esta pesquisa baseia-se no *framework* de referência fornecido pelo *UnifiedProcess* (UP) como a metodologia de trabalho. Cabe ressaltar que esta escolha é motivada pela experiência que o grupo de pesquisadores possui em utilizá-lo em pesquisas e transferências de tecnologia, e porque o seu ciclo de vida iterativo e incremental é apropriado ao desenvolvimento de projetos que visam alcançar um grande dinamismo, incorporando, a todo o momento, o *feedback* de outros grupos de trabalho e/ou projetos relacionados, mas mantendo sob controle os riscos que podem ocorrer no dia-a-dia. Considerando-se o *framework* UP, a pesquisa foi dividida nas seguintes fases:

INÍCIO: Nessa fase foram identificados, os principais temas relacionados com a formalização do Guaraná DSL. Após esta definição, foram considerados e destacados os principais riscos que poderiam afetar o desenvolvimento deste trabalho de pesquisa. Em seguida foram definidas as etapas e prioridades que foram seguidas no decorrer desta pesquisa. Inicialmente, este trabalho inicial foi realizado através de reuniões entre os pesquisadores envolvidos no projeto de formalização do Guaraná DSL.

ELABORAÇÃO: Esta fase consiste em identificar e descrever os principais blocos de trabalho e suas atividades. A partir desta definição deu-se inicio ao desenvolvimento das atividades, as quais foram avaliadas periodicamente, visando fazer um refinamento das ações, incrementando novos pontos não considerados no início do trabalho.

CONSTRUÇÃO: Esta fase dividiu as atividades dos blocos de trabalho em pequenas etapas que foram abordáveis de forma iterativa, buscando ciclos rápidos de análise detalhada da bibliografia, projeto da solução, prototipação e validação, a fim de que fosse possível obter, rapidamente, *feedbacks* úteis. Esta fase partiu de uma reunião inicial, reuniões periódicas para apresentação de resultados e seminários de pesquisa, nos quais foram apresentados os progressos alcançados, sendo eles, resultados parciais ou conclusões sobre a bibliografia estudada. Durante esta fase, participou-se de eventos científicos e tecnológicos relacionadas com o tema de pesquisa, com o objetivo de apresentar os resultados parciais, obter *feedback*, além de ampliar os contatos da equipe da equipe envolvida neste trabalho.

TRANSIÇÃO: O principal objetivo desta fase é a transferência de resultados obtidos. Inicialmente, a especificação formal da sintaxe abstrata, proposta nesta dissertação, será incorporada ao Guaraná DSL, tecnologia desenvolvida pelos pesquisadores envolvidos no projeto. Além disso, os resultados da pesquisa farão parte do processo completo de formalização da tecnologia Guaraná e, futuramente, serão aplicados em projetos reais em colaboração com empresas. Além disto, os resultados serão publicados em conferências e divulgados para outros grupos de pesquisa consolidando e ampliando a rede de colaborações acadêmicas. Dentro de cada fase, UP propõe o desenvolvimento de diversas disciplinas que formam adaptadas ao caso específico dos projetos de pesquisa, a saber:

GESTÃO: Inicialmente, foi efetuado o planejamento das etapas para o desenvolvimento das atividades de pesquisa, as quais consistem em: identificação do problema, contextualização do problema, revisão da bibliografia, trabalhos relacionados, modelagem e validação. Em seguida foi realizada a estimativa dos custos e recursos a serem utilizados. Após, o esforço focou-se no acompanhamento das atividades, a sua evolução, o gerenciamento dos riscos, bem como a identificação de novos problemas relacionados à formalização da linguagem do Guaraná DSL.

MODELAGEM: Esta disciplina focou-se em definir temas de estudo relacionados à formalização da tecnologia de integração de aplicações empresariais Guaraná DSL. Neste sentido, foi realizada a seleção e estudo da bibliografia. Com base neste estudo, iniciou-se a organização e projeto de novas soluções que integrem as existentes, potencializando seus pontos fortes e minimizando o impacto dos pontos fracos, melhorando-os sempre que possível.

PROTOTIPAÇÃO: Com base na fundamentação teórica, apresentou-se um modelo com a especificação formal da sintaxe abstrata do Guaraná DSL e sua validação. Este modelo, posteriormente, será incrementado com as demais etapas do processo de formalização.

DISSEMINAÇÃO: Esta disciplina considerou aspectos de publicidade para a especificação formal do Guaraná DSL, identificando os resultados que podem ser publicados em periódicos e em fóruns científicos ou tecnológicos nacionais e internacionais.

EXPLORAÇÃO: Deseja-se que os resultados obtidos através desta pesquisa de mestrado, contribuam para o processo de formalização do Guaraná DSL. Com isto, espera-se que a pesquisa seja capaz de gerar impacto positivo na sociedade e seja útil para fomentar o desenvolvimento, especialmente na região de inserção do Programa de Pós-Graduação da UNIJUI.

AMBIENTE: Os aspectos considerados por esta disciplina estão relacionados com a gestão dos recursos humanos e materiais do grupo de pesquisa. A gestão de recursos humanos ocupou-se com atividades de organização, acompanhamento, capacitação e motivação dos pesquisadores para que pudessem efetivamente contribuir com a pesquisa. Já a gestão de materiais do grupo ocupou-se em proporcionar uma infraestrutura de pesquisa adequada ao grupo e a presente pesquisa.

1.5 Resumo das Contribuições

O trabalho desenvolvido é parte integrante do projeto de pesquisa sobre tolerância a falhas no contexto de integração de aplicações empresariais, e tem como objetivo evoluir a tecnologia Guaraná. Esta dissertação busca dar o primeiro passo para o processo de formalização, sendo muito importante que a especificação formal da sintaxe abstrata esteja bem formulada.

Neste sentido, a principal contribuição deste trabalho está relacionada à qualidade proporcionada às próximas etapas do processo. No entanto, o trabalho teve outras contribuições importantes:

- Durante o período da revisão bibliográfica, construiu-se uma especificação formal da sintaxe abstrata, definindo apenas algumas características e restrições dos blocos construtores do Guaraná DSL. Esta especificação compôs um artigo enviado, aprovado e apresentado na 16º International Conference on Enterprise Information Systems (ICEIS), realizada em Lisboa, Portugal, nos dias 27 a 30 de abril de 2014, sendo que as publicações deste evento são classificadas com qualis B1 [59].
- O mesmo trabalho também foi apresentado no II Seminário de Formação Científica e Tecnológica, promovido pelo Grupo de Pesquisa em Computação Aplicada (SFCT) no dia 19 de Maio de 2014, Santa Rosa, RS, Brasil. Este evento tem como objetivo criar um espaço local de debate que possa contribuir positivamente sobre o trabalho que vem sendo desenvolvido pelos membros do grupo.

1.6 Estrutura da Dissertação

Essa dissertação está organizada da seguinte maneira:

Parte I: Prefácio. Compreende esta introdução.

Parte II: Revisão da Literatura. Proporciona ao leitor uma revisão da literatura técnica e científica relacionadas à pesquisa desenvolvida nessa dissertação. No Capítulo §2, apresenta-se o referencial teórico. No Capítulo §3, são introduzidos os trabalhos relacionados identificados ao longo dessa pesquisa.

Parte III: Trabalho Desenvolvido. É apresentado o trabalho desenvolvido com base nos objetivos propostos para esta dissertação. No Capítulo §4, é apresentado um estudo sobre os métodos e linguagens formais. No Capítulo §5 é fornecida uma descrição da Notação Z, sua estrutura e suas principais propriedades. No Capítulo §6 é apresentada a contribuição central da pesquisa desenvolvida. E no Capítulo §7 apresenta-se o trabalho de validação da proposta.

Parte IV: Considerações Finais. As conclusões a partir da pesquisa desenvolvida nessa dissertação são apresentadas no Capítulo §8, já os trabalhos futuros são apresentados no Capítulo §9.

Parte II

Revisão da Literatura

Capítulo 2

Referencial Teórico

É a teoria que decide
o que podemos observar.

Albert Einstein, físico teórico alemão (1879-1955)

A formalização de uma linguagem de domínio específico inicia pelo amplo conhecimento do contexto em que ela está inserida e com qual finalidade e aplicabilidade ela foi concebida. Neste sentido, este capítulo apresenta na Seção §2.1 os conceitos associados a integração de aplicações empresariais. Na Seção §2.2 são apresentadas as propriedades e a estrutura das linguagens de domínio específico. A Seção §2.3 apresenta os principais conceitos sobre tolerância à falhas e as suas fases. Na Seção §2.4 é dada a fundamentação teórica sobre a tecnologia Guaraná, sua estrutura e seu mecanismo de tolerância a falhas. E por fim, na Seção §2.5 o capítulo é resumido.

2.1 Integração de Aplicações Empresariais

Com o rápido avanço tecnológico e com uma demanda cada vez maior por serviços ágeis e de qualidade, grande parte das organizações empresariais fazem uso de sistemas de informação para gerenciar seus processos de negócio. Estes sistemas, no entanto, podem apresentar características específicas. Neste contexto surge o conceito de Integração de Aplicações Empresariais (EAI).

A integração de aplicações empresarias tem por objetivo fazer todos os sistemas de informação de uma determinada empresa trabalharem de forma unificada, como sendo uma única aplicação. Este fator é determinante pelo fato de que todas as informações estarão sempre disponíveis em todos os módulos do sistema [70]. De acordo com Laudon e Laudon [64] isto é importante, pois a utilização dos sistemas de informação em um contexto empresarial, é uma forma de manter dados e informações relevantes disponíveis para análise e posterior tomada de decisões, tendo em vista que as decisões podem representar o sucesso ou o fracasso de uma empresa.

Hohpe e Woolf [49] atribuem a uma solução de integração a tarefa de fazer com que as aplicações legadas sejam reutilizadas e todas as suas funcionalidades estejam disponíveis. Além disso, uma solução de integração deve possibilitar que novas funcionalidades sejam criadas sem que as aplicações sejam alteradas. Linthicum [70] define integração de aplicações como sendo o compartilhamento irrestrito de dados e processos de negócios entre todas as aplicações conectadas e fontes de dados na empresa.

Outra definição é dada por Cummins [25] que justifica a utilização da integração de aplicações para obter vantagem competitiva a organização, através da utilização de ferramentas, métodos e planejamento proporcionados com a integração de todas as aplicações em um sistema empresarial único. Este novo sistema é capaz de compartilhar informações e oferecer suporte aos processos de fluxos de negócios.

De maneira resumida, pode-se dizer que uma solução de integração de aplicações propicia a reutilização de duas ou mais aplicações, conforme ilustra a Figura §2.1. Isto vem ao encontro da demanda de mercado, ou seja, em um contexto econômico e industrial atual, deseja-se que os sistemas empresariais sejam projetados para atender as mudanças de requisitos funcionais, a evolução tecnológica e de negócio [20].

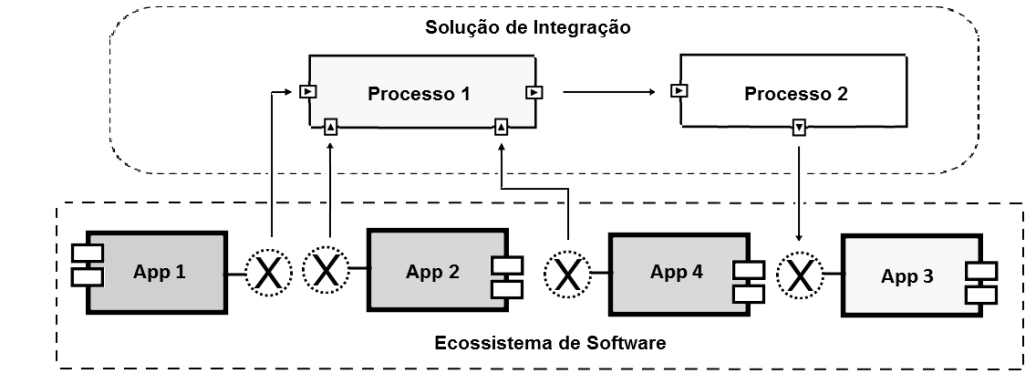


Figura 2.1: *Ambiente de Integração de Aplicações*

Deste modo, existe a necessidade de conciliar a criação de novos sistemas utilizando as tecnologias existentes sem deixar de aproveitar os sistemas legados e fazer com que os sistemas antigos se mantenham operantes nas novas plataformas ou trabalhem em conjunto com elas, considerando que, segundo Laudon e Laudon [64], a não reutilização das aplicações, a inconsistência e a duplicidade de dados, além da dificuldade de relacionar informações de diferentes aplicações, poderiam interferir negativamente em estratégias de negócios baseadas nestas informações.

2.1.1 Ecossistema de Software

Percebe-se que as aplicações computacionais fazem parte do cotidiano das organizações empresariais. Com o objetivo de gerenciar os seus processos de negócio, as mesmas desenvolvem um papel fundamental.

No entanto, as aplicações de uma organização podem não compor um único sistema de software, mas sim um conjunto de aplicações distintas que atendam, cada qual um setor ou departamento específico. De acordo com Messerschmitt e Szyperski [80] este ambiente é denominado de ecossistema de software. Aplicações distintas constituem este ecossistema, cf. Figura §2.2, sejam elas desenvolvidas na própria empresa, aplicações legadas ou aplicações absorvidas por uma fusão empresarial.

De acordo com Yu e outros [114] um ecossistema de software se refere a um conjunto de aplicações que apresentam um determinado grau de relação. Já van den Berk e outros [111] fazem uma abordagem mais ampla do conceito. Segundo ele ecossistemas de software são formados por uma

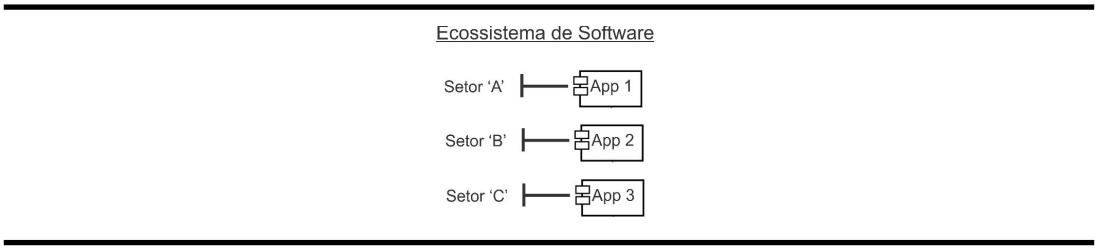


Figura 2.2: *Ecossistema de Software*

plataforma de software, um conjunto de desenvolvedores internos e externos, e uma comunidade especialista no domínio em serviços para o conjunto de usuários, que compõe soluções relevantes para as suas necessidades. Com isso, tem-se a participação de fatores externos que contribuem para a heterogeneidade deste ecossistema.

Dada a dinâmica aplicada a composição do ecossistema de software empresarial, constata-se que, em grande parte dos casos, as aplicações em questão constituem um ecossistema heterogêneo, e não foram desenvolvidas levando em conta a integração e por isto não oferecem características que facilitariam a mesma. É neste contexto que a integração de aplicações desempenha seu papel: integrar este conjunto de aplicações que compõe o ecossistema e fazer com que operem como um sistema único.

Dois estilos de integração são considerados por Hohpe e Woolf [49]. A integração de aplicações para o *compartilhamento de dados* através da transferência de arquivos ou com a utilização de um banco de dados. E a integração de aplicações para o *compartilhamento de funcionalidades* de uma determinada aplicação que podem ser acessadas pelas demais aplicações, através de uma chamada de procedimento remota ou através de um sistema baseado em mensagens.

2.1.2 Estilos de Integração

A integração de dados é caracterizada pela troca de informações entre aplicações. Esta troca pode ocorrer através da transferência de arquivos conforme apresentado na Figura §2.3 (a), ou através da utilização de um gerenciador de banco de dados conforme apresentado na Figura §2.3 (b).

Algumas características são importantes e precisam ser consideradas para o compartilhamento de informações através da utilização de arquivos. Como

aplicações distintas acessam o arquivo, seja para escrever no mesmo ou para fazer a leitura dos dados nele contidos, é necessário que se conheça todos os atributos deste arquivo, como nome, extensão e localização, além do formato do arquivo [49].

Uma grande vantagem considerada por Hohpe e Woolf [49], é que neste estilo de integração, não é necessário que se tenha acesso ao código fonte das aplicações envolvidas, sendo apenas necessário que se conheça o conteúdo e o formato dos arquivos. Estes arquivos são utilizados pela aplicação interessada sem sofrer ou causar interferência para as demais aplicações. Com isso, tem-se aplicações independentes entre si utilizando o mesmo arquivo de dados.

Porém, um fator negativo é o custo computacional para persistir novos dados ou para ler dados destes arquivos. Por isto, este processo é executado periodicamente pela solução de integração. Assim, quanto maior este período, maior a chance da solução de integração estar trabalhando com dados obsoletos [49].

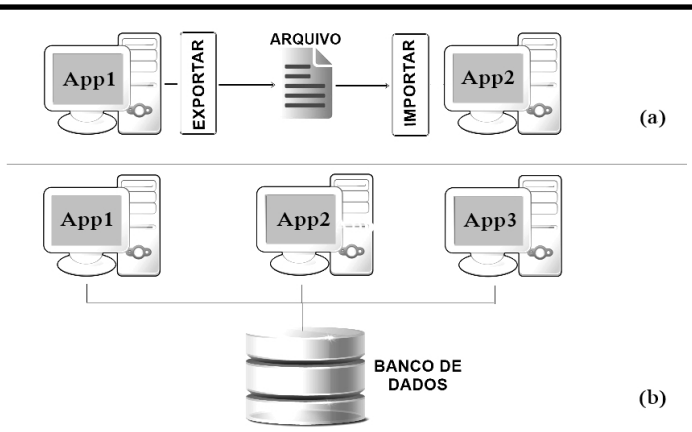


Figura 2.3: *Compartilhamento de dados (de Hohpe e Woolf [49])*

O compartilhamento de dados utilizando banco de dados, cf. Figura §2.4 (b) é caracterizado por permitir o acesso à todas as aplicações participantes de uma solução de integração para executar os processo de leitura, escrita e alteração. Este tipo de compartilhamento exige que algumas regras sejam estabelecidas e obedecidas pelas aplicações. Por exemplo, caso uma aplicação esteja executando um processo de alteração em alguma tabela do banco de dados, esta tabela não pode ser acessada simultaneamente por outro processo [49].

Algumas vantagens são consideradas em relação ao compartilhamento de dados através de arquivos. A utilização de banco de dados proporciona a comunicação assíncrona entre as aplicações. Além disso, os dados são consistentes, pois os processos são executados sempre no mesmo banco de dados, mantendo as suas informações atualizadas e disponíveis para o acesso a todas as aplicações [49].

Hohpe e Woolf [49] destacam dois pontos negativos neste tipo de compartilhamento de dados. Eles consideram de grande dificuldade a criação e manutenção de um esquema de dados único, considerando as constantes alterações e as novas funcionalidades que são demandadas pelo crescimento das empresas. Outro fator considerado, é a perda de desempenho, provocada pelo acesso ao banco de dados por vários processos simultaneamente.

Já a integração de aplicações para o *compartilhamento de funcionalidades* é caracterizado pela disponibilização de operações de uma aplicação para que possam ser acessadas por outras aplicações. Este estilo de integração pode ser realizado através de procedimento remoto ou baseada em um sistema de mensagens [49].

Uma chamada de procedimento remoto representada na Figura §2.4 (a), possibilita que aplicações acessem e executem operações disponibilizadas por uma outra aplicação através de uma API. Assim, uma aplicação pode enviar uma solicitação para outra aplicação e receberá a resposta para esta solicitação, de forma muito semelhante a uma chamada de procedimento local [49].

No entanto, segundo Hohpe e Woolf [49], alguns problemas podem ocorrer e precisam ser considerados. O desempenho é atrofiado, pelo tempo da chamada e da resposta, além, da possibilidade de se ter inúmeras solicitações para a mesma funcionalidade, sujeitando a máquina onde esta localizada a aplicação disponibilizadora a um processamento maior e consequentemente perdendo desempenho. Múltiplas solicitações a mesma funcionalidade, também podem provocar uma sobrecarga, resultando em falhas na aplicação.

Por fim, o compartilhamento de funcionalidades baseado em um sistema de mensagens é caracterizado pela transferência de mensagens entre aplicações conforme representado na Figura §2.4 (b). Isto proporciona que, além do compartilhamento de funcionalidades, também seja possível a transferência de dados [49].

A transferência de mensagens é realizada de forma assíncrona através de um canal, conhecido como barramento de mensagens. Com isso, é

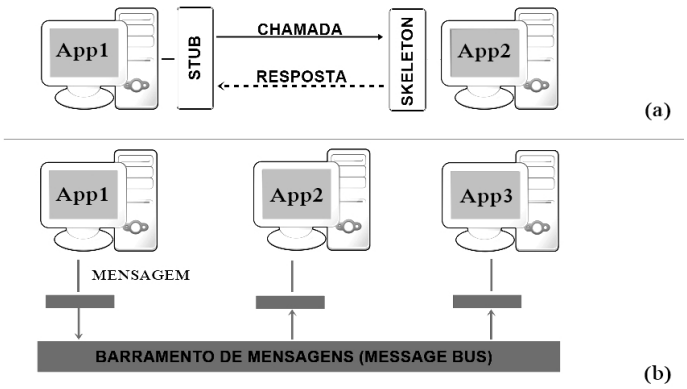


Figura 2.4: *Compartilhamento de funcionalidades (de Hohpe e Woolf [49])*

possível enviar uma mensagem mesmo que a aplicação de destino não esteja em funcionamento. A mensagem permanece no canal até que possa ser entregue. Neste momento ela é processada e a resposta é novamente enviada pelo canal, sendo entregue a aplicação que solicitou a funcionalidade, assim que ela esteja sendo executada [49].

2.1.3 Topologias

Uma topologia para a integração de aplicações pode ser vista basicamente como uma arquitetura que determina a forma com que a solução será implementada. São consideradas as topologias ponto-a-ponto e Enterprise Service Bus (ESB).

Uma integração baseada na topologia ponto-a-ponto se caracteriza basicamente por uma aplicação servir de cliente e servidor ao mesmo tempo. Nesta topologia as aplicações compartilham serviços e dados entre si sem a necessidade de um servidor central. É uma abordagem simples, porém ultrapassada. Resulta em códigos de integração personalizados e espalhados entre as aplicações de forma descentralizada dificultando o monitoramento dos processos [79].

Por outro lado uma topologia ESB consiste em uma arquitetura mais elaborada que permite integrar diferentes aplicações através de um barramento de comunicação. Nesta topologia as aplicações se comunicam entre si através deste barramento, de forma independente e sem interferência entre as aplicações [79]. Chappell [21] destaca que esta topologia é utilizada em grande parte das soluções de integração de aplicações empresariais.

De acordo com Chappell [21] a topologia ESB é composta por um conjunto de adaptadores, uma linguagem e um motor de orquestração. Os adaptadores permitem aos engenheiros de software abstrair os detalhes relativos às distintas tecnologias para a comunicação com as aplicações do ecossistema de software. A linguagem de orquestração permite criar modelos que descrevam a um alto nível de abstração as soluções de integração. O motor de orquestração, também conhecido como motor de integração, proporciona todo o suporte necessário à execução de soluções de integração.

Frantz e outros [36] evidenciam que uma ESB fornece a tecnologia necessária para implementar soluções de integração. Segundo o autor, grande parte das ESBs aderiram a especificação da arquitetura pluggable de serviços fornecido pela JBI (Java Business Integration), como por exemplo, Open ESB, Petals ESB, ServiceMix e Fuse ESB. Em JBI, adaptadores e conectores são tratados como componentes de ligação, e a linguagem utilizada é a orquestração Business Process Execution Language (BPEL).

Diversos tipos de componentes de ligação estão disponíveis atualmente, o que permite que as soluções de integração se conectem a praticamente qualquer aplicação existente. Esta característica é aproveitada pela tecnologia Guaraná que está estruturada sobre a topologia ESB através da utilização Open ESB [36].

2.1.4 Tecnologias

Utilizar todas as tecnologias disponíveis é fundamental para uma organização empresarial. É preciso integrar dados, informações e funcionalidades, através das topologias disponíveis, para que o serviço prestado seja o de qualidade. Este fator fortaleceu a demanda por soluções de integração, contribuindo para o surgimento de algumas tecnologias. Estas tecnologias oferecem suporte a concepção e implementação de soluções.

Camel é um framework Java que permite implementar soluções de integração empresarial. Através de uma DSL concisa mas sofisticada torna possível abstrair a lógica de integração de uma aplicação utilizando Java, XML, ou Scala [52].

Fisher e outros [31] descrevem Spring Integration que estende o Spring Framework para suportar os padrões de integração de aplicações empresariais descritos por Hohpe e Woolf [49]. Como o próprio Spring Framework tem a preocupação com a produtividade do desenvolvedor, torna mais fácil construir, testar e manter soluções de integração empresarial.

Outra tecnologia encontrada é Mule [28]. De acordo com Dossot e outros [28], Mule segue os padrões de integração propostos por Hohpe e Woolf [49], é leve e personalizável.

Frantz e outros [36] apresentam uma tecnologia chamada Guaraná, com a qual é possível gerar soluções de integração com alto nível de abstração. A tecnologia oferece aos seus usuários um mecanismo de tolerância a falhas baseado em regras. Este mecanismo difere o Guaraná das demais tecnologias, as quais utilizam um mecanismo de detecção de erros baseado principalmente na utilização de *try-catch*. Este mecanismo de tratamento de exceções utiliza três palavras chave: *try, catch e throw*. Assim, os comandos do programa que devem ser monitorados para as exceções, estão contidos no bloco de prova (*try*). Caso ocorra uma exceção dentro do bloco de prova, o controle do programa é transferido para o gerenciador de exceção apropriado (*catch*). O código do comando *catch* tenta reparar a exceção. Quando a exceção é reparada, a execução da aplicação continuará normalmente com os comandos que seguem o *catch*. No entanto, se o erro não for corrigido o bloco catch encerra a execução da aplicação [45].

Todas as tecnologias apresentadas oferecem suporte à soluções de integração. No entanto, destacou-se a tecnologia Guaraná, pelo fato da mesma ser criada por pesquisadores que atuam no mesmo grupo de pesquisa em que a presente dissertação foi desenvolvida, o que proporcionou facilidade de acesso ao código fonte e a suas funcionalidades, além de possuir como diferencial uma abordagem baseada em regras do mecanismo de tolerância a falhas.

2.2 Linguagem de Domínio Específico

Para escrever códigos de aplicações, são utilizadas linguagens de programação. Quando uma linguagem possui sua expressividade limitada a um determinado conjunto de problemas, ela é conceituada como linguagem de domínio específico (DSL).

Para Van Deursen e outros [112] cabe uma ampla discussão sobre o exato significado para o termo, mas atribuem um conceito simples: uma linguagem de domínio específico é uma linguagem destinada a um domínio do problema. Para os autores, estas linguagens geralmente são pequenas, oferecendo apenas um conjunto restrito de notações e abstrações, alcançando maior facilidade de uso se comparadas com as linguagens de propósito Geral (GPL).

2.2.1 DSLs x GPLs

Optar por uma determinada linguagem é parte importante no processo de desenvolvimento de aplicações ou soluções de integração. Inúmeras linguagens de propósito geral já estão disponíveis e bem estabelecidas, enquanto que o número de linguagens de domínio específico vem evoluindo gradativamente, fortalecido pelos seus grandes benefícios. Neste sentido, é importante conhecer os principais conceitos sobre estas duas abordagens.

De acordo com Raja e Lakshmanan [89] uma GPL se aplica a praticamente todos os problemas. No entanto, estas linguagens devem aceitar qualquer algoritmo que possa ser executado, não fazendo referência à métodos específicos de um domínio restrito.

Kosar e outros [61] destacam que as GPLs como Java e C, por exemplo, estão perfeitamente estabelecidas no ciclo de vida de desenvolvimento de software e suas características são amplamente difundidas entre os engenheiros de software. No entanto, exigem boas habilidades de programação por parte dos engenheiros.

Um linguagem de domínio específico, no entanto, tem uma abrangência mais restrita, mas com grande praticidade, facilidade e eficiência em promover soluções. De acordo com Fowler [32] as DSLs estão estritamente relacionadas com um domínio de aplicação podendo ser muito poderosas dentro dele, mas não cobrem situações alheias a este domínio. Para o autor, as DSLs podem simplificar código complexo, promover uma comunicação eficaz com os clientes, melhorar a produtividade e remover gargalos no desenvolvimento.

Getir e outros [44] consideram que uma Linguagem de Modelagem de Domínio Específico pode fornecer uma abstração suficiente e dar suporte para o desenvolvimento de Sistemas com maior eficiência. Kelly e Tolvanen [57] destacam que uma DSL abrange uma classe específica de problemas, mas oferece maior precisão e expressividade para modelar um determinado problema.

Em Raja e Lakshmanan [89] encontramos algumas vantagens das DSLs sobre as GPLs. A capacidade de qualquer especialista usar uma DSLs para desenvolver aplicações é destacada como principal vantagem, pois contrasta com a complexidade das GLPs, que exigem conhecimento técnico avançado. No entanto, outras características também são lembradas. Segundo os autores, DSLs são muito expressivas e a sua sintaxe é legível e de fácil compreensão. Além disso, são mais produtivos, pois terão menos tempo de programação em comparação com GPLs.

Kosar e outros [61] apresentam um comparativo das duas abordagens e mostrando as principais vantagens das DSLs: manutenção de software simplificada com documentação disponível, reutilização de software e melhor desempenho de processamento devido ao domínio restrito e o conhecimento centralizado. Estes aspectos são responsáveis por diminuir os custos de engenharia e reengenharia, além de aumentar a confiabilidade e facilidade de manutenção do software construídas com DSLs.

No mesmo artigo os autores mostram um rendimento 15% maior quando os programadores foram submetidos a uma DSL em comparação com a GPL. Foram avaliadas questões de aprendizagem, percepção e evolução.

2.2.2 Características Desejadas da DSL

Considerando os benefícios e a sua crescente aplicabilidade, mais estudos estão sendo feitos recentemente sobre as DSLs. Algumas características são fundamentais, segundo Raja e Lakshmanan [89], para que uma linguagem de domínio específico esteja bem estabelecida. Para eles uma DSL deve conter ao menos as seguintes características:

- ser simples de modo a reduzir o tempo de aprendizagem;

- deve estar construída sobre o vocabulário de domínio do usuário;

- a sintaxe fornecida pela DSL deve esconder o aspecto de programação inerente à aplicação do cliente;

- apresentar um custo de implementação menor comparado a uma GPL.

Na Figura §2.5 é apresentado um gráfico que compara o custo de implementação através de uma metodologia convencional com uma metodologia baseada em uma DSL. Apesar de um custo inicial maior da metodologia baseada em uma DSL, o custo final é expressivamente menor em comparação com uma metodologia convencional [51].

2.2.3 Modelo e Metamodelo

A definição de modelo geralmente é associada a um conjunto de elementos utilizados para descrever um sistema, de fácil entendimento e relacionado a algo real que se queira representar [77].

Assmann e outros [6] definem modelos como sendo representações fiéis, porém simplificadas e contextualizadas, de algo real, e considera que bons

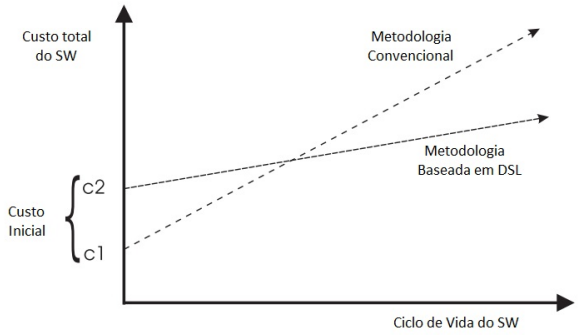


Figura 2.5: *Comparação de custos de implementação (de Hudak [51]).*

modelos podem servir como meios de comunicação. Os autores complementam que os modelos possuem um grau de abstração maior que o do sistema, o que possibilita que detalhes irrelevantes sejam ignorados.

De acordo com Mellor [77] e Assmann e outros [6], os modelos normalmente apresentam uma combinação de desenhos e texto e podem ser utilizados para representar muitos tipos diferentes de realidades, como por exemplo, domínios, linguagens ou sistemas.

Umas das principais, mais conhecidas e utilizadas linguagens para construção de modelos é a Unified Modeling Language (UML). A UML é uma linguagem que utiliza diagramas para especificar, visualizar e documentar modelos de software orientados por objetos.

Por outro lado, um meta-modelo é utilizado para descrever e especificar modelos, ou seja, ele descreve os elementos que podem ser utilizados na composição dos modelos. De acordo com Assmann e outros [6], um meta-modelo faz declarações sobre o que pode ser expresso nos modelos de uma certa linguagem de modelagem. Assim, um meta-modelo é um modelo prescritivo de uma linguagem de modelagem.

De acordo com Mellor [77] um meta-modelo define a estrutura, semântica e restrições para um grupo de modelos que compartilham sintaxe e semântica comum. O autor destaca que um metamodelo UML descreve como modelos UML pode ser estruturados, os elementos que podem conter (classes, atributos, associações) e define as propriedades para estes elementos.

A Figura §2.6 apresenta um exemplo simples de aplicação de modelo e metamodelo para uma máquina de estados finita. O modelo apresenta um estado inicial, várias elementos de transição, estados intermediários e um ponto

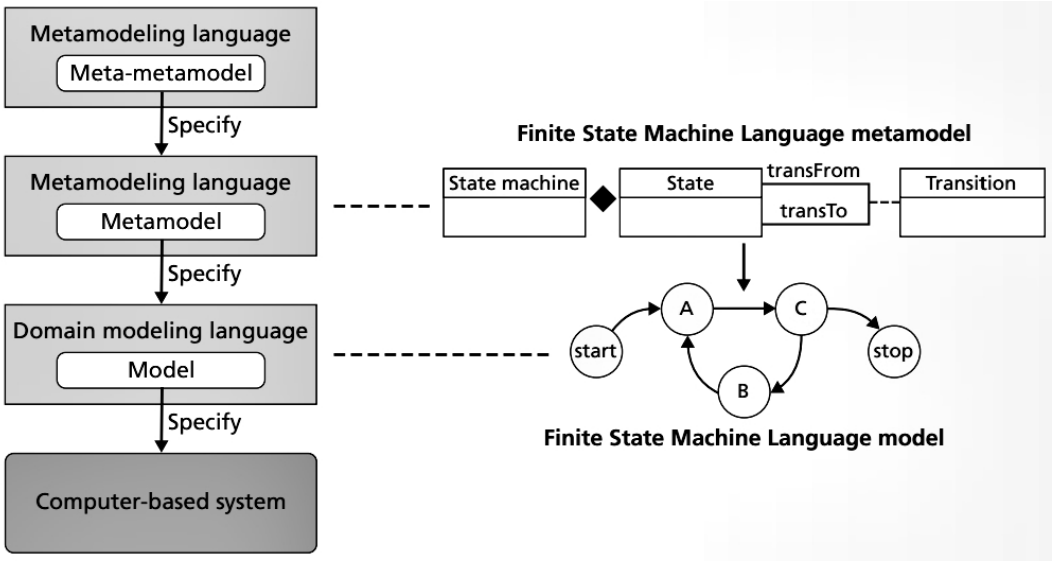


Figura 2.6: *Exemplo de modelo e meta-modelo (de Lédeczi e outros [67]).*

de parada. Já o metamodelo caracteriza e define cada um destes elementos utilizados.

2.2.4 Restrições no Modelo com OCL

Uma das abordagens mais utilizadas para estabelecer a verificação de restrições em modelos é através da Object Constraint Language (OCL) [18]. A OCL foi proposta como parte integrante de UML, com a finalidade de permitir escrever restrições sobre os objetos de um sistema a serem especificados por meio de invariantes, operações de pré e pós-condições [73].

Cranefield e Purvis [24] destacam que OCL pode ser usado para restringir os valores de atributos e possíveis instâncias dos relacionamentos. Segundo os autores a linguagem é poderosa e permite estabelecer restrições que não podem ser escritas por uma descrição lógica. Chiorean e outros [22] acrescentam-lhe as característica de linguagem formal e flexível.

A grande maioria dos autores, como Marcano e Levy [73], Roe e outros [97], Ziemann e Gogolla [116], no entanto, classificam a mesma como uma linguagem semi-formal. A principal justificativa para esta classificação está baseada no fato de que OCL é uma linguagem textual para especificar restrições adicionais sobre os objetos do sistema [116]. Este estilo de especificação

pode levar os usuários a más interpretações do modelo UML de um sistema e isto torna a UML/OCL uma base insegura para o desenvolvimento de software [73].

Marcano e Levy [73] associam a falta de semântica formal padronizados para UML/OCL ao número pequeno de ferramentas de apoio disponíveis para análise e verificação. Do mesmo modo, Richters e Gogolla [96] destacam que geralmente não há um apoio substancial para restrições escritas na linguagem de restrição Object (OCL).

Frantz e outros [36] utilizam um metamodelo UML com as restrições impostas descritas em OCL, para representar a sintaxe abstrata do Guaraná DSL. Getir e outros [44], no entanto, consideram importante que haja uma especificação formal para a linguagem. Segundo ele, através da representação formal consegue-se identificar uma definição equivocada, definir um significado mais preciso à um modelo, além de ter a possibilidade de gerar código com definição precisa e correta das propriedades para a implementação real dos modelos a partir da ferramenta baseada na linguagem.

2.2.5 Sintaxe e Semântica

Além das características que uma DSL deve prover, existe uma estrutura básica estabelecida que dá suporte à uma linguagem: a sintaxe (abstrata e concreta) e a semântica. Kelly e Tolvanen [57] definem de maneira geral os conceitos para cada elemento. A *sintaxe abstrata* é um metamodelo que descreve os conceitos e as regras de modelagem da DSL. A *sintaxe concreta* é relacionada com a notação gráfica que representa, através de símbolos, as classes da sintaxe abstrata. E, por fim, a *semântica* que fornece significado para cada elemento gráfico da DSL.

A definição destes elementos é fundamental para caracterizar uma linguagem de modelagem. A sintaxe geralmente é definida baseada em um metamodelo. No entanto, nem todos os detalhes estruturais e restrições de uma DSL podem ser expressos em um diagrama de classe através do metamodelo. Sendo assim, é necessário que sejam feitas as definições e implementações das restrições que constituem a semântica [116].

2.2.6 DSL interno e externo

Raja e Lakshmanan [89] destacam e conceituam dois tipos de DSLs: DSLs internos e DSLs externos.

Conforme os autores apresentam, uma DSL interna se caracteriza por utilizar e estender uma linguagem de programação existente, também chamada de linguagem nativa. Nesta abordagem, toda a infra-estrutura da linguagem base é herdada pela DSL. O DSL é construído e incorporado como uma biblioteca de funções escritas na linguagem nativa, criando novos tipos de dados, rotinas, procedimentos e macros [89]. A principal vantagem destacada pelos autores é o ganho de tempo em relação a programação convencional. Isto é alcançado através da criação de programas que manipulam outros programas, denominada de meta-programação [89]. A meta-programação torna a programação mais legível (fluente) para um determinado propósito [32]. Raja e Lakshmanan [89] citam alguns exemplos de DSLs internas: Ruby on Rails e RSpec utilizando Ruby, "Cascading"e SmartFrog utilizando Java.

Por outro lado, as DSLs externas são linguagens totalmente novas. Para Raja e Lakshmanan [89] a ideia que norteia esta abordagem é conceber uma linguagem para um domínio específico a partir do zero (ex.: SQL e Make). Para tanto, é necessário compor a gramática e o compilador para esta linguagem, para que seja possível analisar e processar a sintaxe e mapeá-la para a semântica.

Raja e Lakshmanan [89] apresentam um comparativo entre as duas abordagens. Uma vantagem da DSL interna é que o compilador da linguagem base é reutilizado. Já a validação da sintaxe é mais difícil em uma DSL interna, pois o código pode ser processado de forma dinâmica, ao passo que em uma DSL externa a definição da gramática serve também para validar a sintaxe. No entanto, a verificação de erros e a validação precisam ser feitos. Além disso, uma DSL interna sofre limitações pela estrutura e sintaxe da linguagem base, contrastando com a flexibilidade proporcionada por uma DSL externa.

2.3 Tolerância a Falhas

O desenvolvimento de softwares tem crescido muito nos últimos anos. Este crescimento exige mais agilidade por parte dos desenvolvedores. No entanto, esta grande demanda, muitas vezes faz com que o desenvolvimento de aplicações computacionais perca qualidade, tornando-as mais suscetíveis à falhas. E falhas são indesejáveis, porém são inevitáveis.

De acordo com Lee e Anderson [69], uma falha é caracterizada como sendo um desvio da especificação, ou seja, quando uma aplicação tem um comportamento diferente do esperado e esta ocorrência afeta diretamente o usuário. Neste contexto, vários fatores podem ser considerados como causas para a ocorrência de falhas. Todos os fatores, no entanto, são associados à defeitos ou erros. Lee e Anderson [69] conceitua estes termos.

Defeito: é um problema físico ou de algoritmo que ocorre em um nível mais baixo. O defeito é a causa de um erro, mas nem sempre provoca um erro. Isto ocorre, por exemplo, quando uma determinada linha de código que contém um defeito não executada.

Erro: é caracterizado por um estado inconsistente ou inesperado durante a execução de uma aplicação, ocasionado pela execução de um defeito. Um erro pode provocar uma falha.

Além disso, existem diversas classificações para falhas na literatura técnica [5, 63, 87]. As falhas aparecem geralmente classificadas em falhas físicas e falhas humanas, compreendendo as falhas dos componentes eletrônicos e as falhas de projeto e de interação, respectivamente. Conforme estes autores, pode-se destacar como principais causas de falhas: os problemas de especificação, problemas de implementação, componentes defeituosos, imperfeições de manufatura, fadiga dos componentes físicos além de distúrbios externos como radiação, interferência eletromagnética, variações ambientais (temperatura, pressão, umidade) e também problemas de operação. Por outro lado, sabendo a causa da falha pode-se definir também a natureza da mesma: falha de hardware, falha de software, falha de projeto e falha de operação.

Falhas de software e também de projeto são consideradas atualmente o mais grave problema em computação crítica, pois apresentam um grande potencial de comprometer a confiabilidade do sistema, levando-se em consideração que estes sistemas geralmente são construídos para suportar falhas físicas [63].

Apesar da incidência de falhas, as mesmas podem ser suprimidas pelo uso adequado de técnicas de tolerância a falhas. Fuks e Pimentel [39] conceituam a tolerância a falhas como sendo a forma de fazer com que o sistema esteja operacional e execute suas funções corretamente ou interrompa as suas funções de forma a não provocar dano a outros sistema ou pessoas que dele dependam.

Campbell e Randell [19] destacam que fornecer um sistema tolerante a falhas é fornecer um sistema que continua provendo corretamente os seus serviços mesmo na presença de falhas de hardware ou de software, e seus defeitos não chegam a ser visíveis para o usuário, pois o sistema detecta e mascara defeitos antes que eles alcancem os limites do sistema.

2.3.1 Fases para Tolerância a Falhas

Uma abordagem comum para a técnica de tolerância a falhas, esta baseada em quatro fases de aplicação. Lee e Anderson [69] denominaram estas

fases como: fases de detecção, confinamento, recuperação e tratamento. Eles consideram o mascaramento de falhas como sendo uma técnica a parte, não usada em complemento às demais.

Outra definição é dada por Frantz [34] que também definiu quatro etapas para o mecanismo de tolerância a falhas: *Event Reporting, Error Monitoring, Error diagnosing e Error Recovery*. Esta classificação é aplicada à Tecnologia Guaraná e é com esta classificação que estaremos trabalhando nesta pesquisa, mais especificamente com a segunda fase, a qual faz o monitoramento e a detecção do erro.

2.4 Tecnologia Guaraná

Uma solução de integração de aplicações empresariais precisa oferecer suporte a dois tipos de problemas encontrados em ecossistemas de software: manter os dados, informações e aplicações em sincronia, além de proporcionar que novas funcionalidades sejam criadas sobre as já existentes [49].

Neste contexto, a tecnologia Guaraná fornece suporte para construir soluções de integração de uma forma acessível aos engenheiros de software. As soluções de integração geradas com a tecnologia, utilizam padrões de integração por meio de um modelo gráfico permitindo que engenheiros de software tenham a visão de todo o conjunto de processos que compõe uma solução de integração [36] conforme demostrado na Figura §2.7.

Portanto, com uma visão voltada para a facilidade de uso, a tecnologia Guaraná se mostra uma opção interessante quando se fala em integração de aplicações. A construção gráfica de soluções e a tolerância a falhas baseada em regras a partir de um monitor externo, são diferenciais da ferramenta [36].

2.4.1 DSL

A tecnologia Guaraná possui uma linguagem de domínio especifico para projetar soluções de integração de aplicações empresariais em um alto nível de abstração, baseada nos padrões de integração documentados por Hohpe e Woolf [49] e apresentados por Frantz [34].

De acordo com Frantz e outros [36], a tecnologia Guaraná refere-se a um projeto cujo objetivo é proporcionar, aos engenheiros de software, ferramentas que eles possam usar para criar e implementar soluções de integração. A

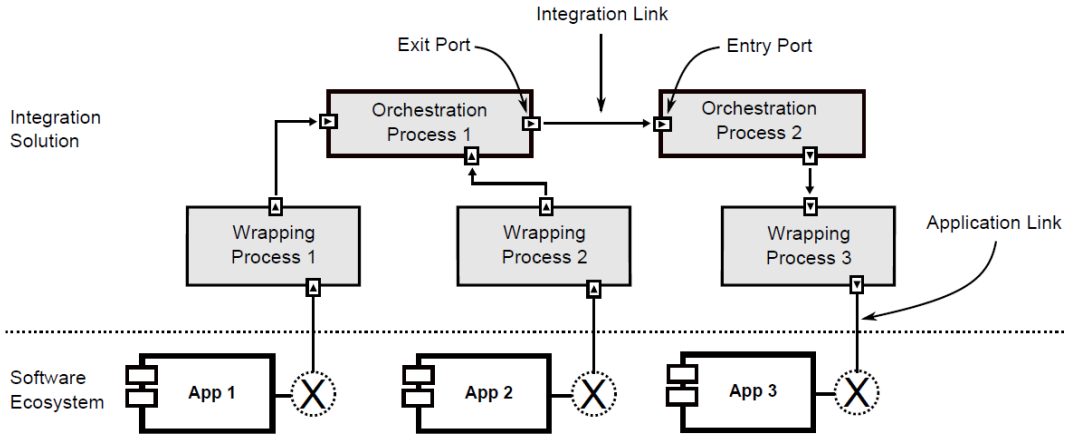


Figure 1. A typical EAI solution.

Figura 2.7: *Visão abstrata de uma Solução de Integração (de Frantz e outros [36])*

Figura §2.8 apresenta visão abstrata do processo de construção da tecnologia. Note que o *metamodelo* é dividido em duas partes: uma principal (*Core DSL*), que suporta um subconjunto de conceitos que são assumidos como sendo úteis para um conjunto grande de soluções de integração; e uma série de caixas de ferramentas de tarefas (*Task Toolbox*), que suportam subconjuntos de tarefas com características específicas para um determinado domínio de integração.

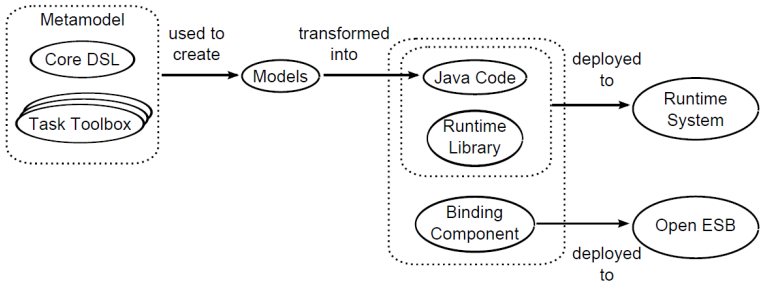


Figura 2.8: *Processo de construção do Guaraná (de Frantz e outros [36])*

A partir deste processo de construção, um engenheiro de software pode

usar os conceitos definidos no metamodelo para criar seus próprios modelos, que são soluções específicas para os problemas de integração específicos. Os modelos construídos com o Guaraná são gráficos e permitem encontrar soluções de integração em um alto nível de abstração.

Além disso, a tecnologia Guaraná também fornece um conjunto de transformações através do qual um modelo construído pode ser transformado em código Java (*Java Code*). Este código invoca uma biblioteca em tempo de execução (*Runtime Library*) que fornece as classes necessárias para implementar os conceitos suportados pelo metamodelo. Além do código Java e da biblioteca, também são necessários alguns componentes de ligação (*Binding Component*), os quais devem ser configurados e implantados para executar em um ESB (*Enterprise Service Bus*) independentemente. Os processos que compõe a solução devem usar estes componentes de ligação para interagir com as aplicações que estão sendo integradas ou com outros processos [36].

2.4.2 SDK

Além da linguagem de domínio especifico, a tecnologia Guaraná também possui um Kit de Desenvolvimento de Software (SDK) denominado de Guaraná SDK. De acordo com Frantz e Corchuelo [33], o Guaraná SDK é a implementação java do Guaraná DSL e fornece suporte para a implementação de soluções de integração.

O Guaraná SDK está organizado em duas camadas: o *framework* e o kit de ferramentas, cf. Figura §2.9. O *framework* fornece uma série de classes e interfaces que fornecem a base para implementar tarefas, adaptadores e fluxos de trabalho. Já o kit de ferramentas estende o *framework* e proporciona a execução de tarefas e adaptadores que visam atender a um propósito geral. Além disso, Frantz e Corchuelo [33] destacam que outros kits de ferramentas para contextos específicos podem ser desenvolvimentos.

Outra propriedade encontrada na kit de ferramentas do Guaraná SDK, é o sistema em tempo de execução (*Runtime System*). Esta ferramenta utiliza um modelo de execução com base em tarefas para executar as soluções. Esta abordagem difere o Guaraná das outras tecnologias que utilizam um modelo de execução baseado em processos [33].

Considerando este modelo de execução do Guaraná baseado em tarefas, não incide o problema de prioridade de execução como no modelo baseado em processos. Isto permite utilizar os recursos do sistema mais eficientemente. Este nível mais baixo de granularidade permite que as tarefas

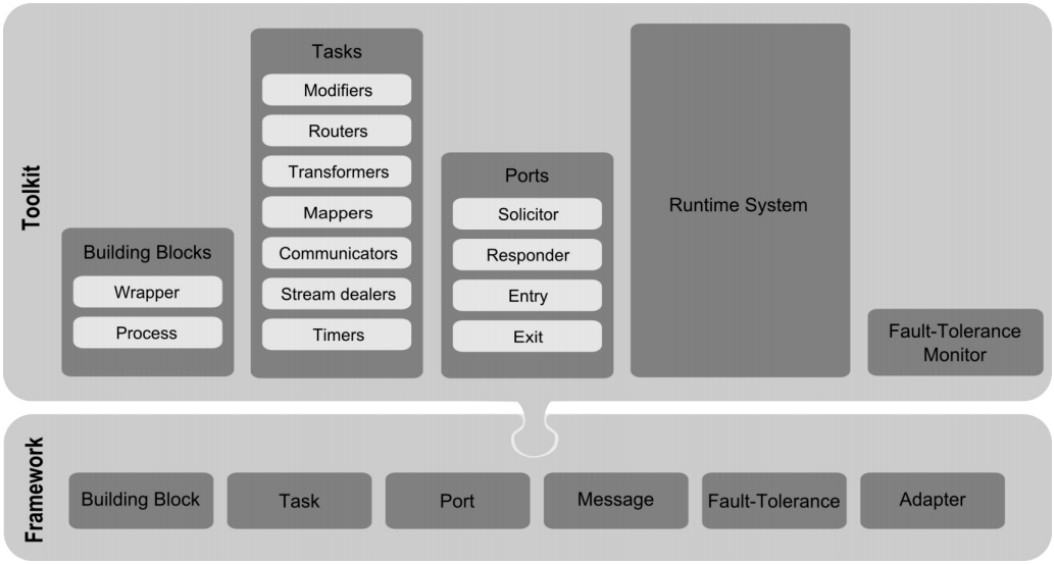


Figura 2.9: *Arquitetura do Guaraná SDK (de Frantz e Corchuelo [33])*

sejam executadas com maior prioridade, já que não é necessário que todas as mensagens, para iniciar um processo, estejam disponíveis [33].

De acordo com Frantz e Corchuelo [33] o Guaraná SDK fornece uma arquitetura bem projetada, com menor custo de manutenção quando comparado com as tecnologias Camel e Spring Integration. Além disso, a arquitetura se adapta com mais facilidade para um contexto específico.

2.4.3 Blocos de Construção do Guaraná

O modelo conceitual, cf. Figura §2.10 apresenta os conceitos necessários para modelar soluções de integração. As funcionalidades e a estrutura de uma solução de integração são completamente definidas utilizando blocos construtores: *portas, tarefas, decorador, slots e links*.

Segundo Frantz e outros [36] uma solução apresenta um conjunto de processos que integram um conjunto de aplicações. De forma resumida, um processo pode ser visto como um processador de mensagens. Uma mensagem é uma abstração de uma parte da informação que é trocada e transformada através uma solução de integração. A estrutura de mensagens depende completamente das soluções em que elas estão envolvidas. As tare-

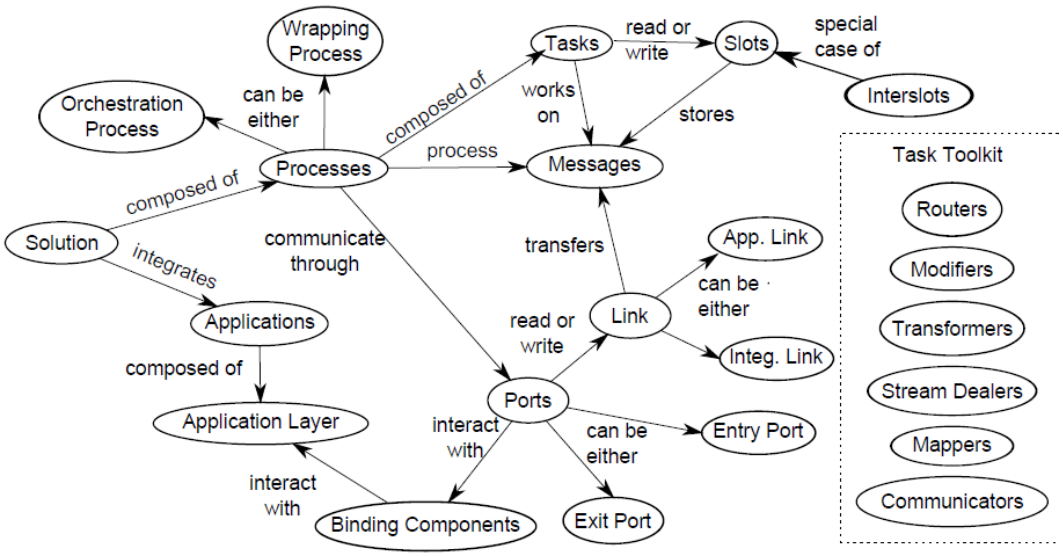


Figura 2.10: *Mapa Conceitual da Tecnologia Guaraná (de Frantz e outros [36])*

fas são extremamente dependentes de contexto, o que faz com que seja de pouco interesse pensar em uma coleção de propósito geral para tarefas.

Os processos utilizam portas para se comunicar uns com os outros ou com as aplicações envolvidas em uma solução de integração. De maneira geral, uma porta é responsável por abstrair os detalhes necessários para interagir com um componente de ligação (*comunicator*), o qual, por sua vez, abstrai os detalhes necessários para interagir com um aplicativo ou com um processo. A interação com uma aplicação pode ocorrer em uma ou mais camadas: a camada de acesso a dados, a camada da lógica de negócios e/ou a camada de interface do usuário.

As portas (*ports*) podem ser ou portas de entrada ou de saída, dependendo da sua concepção, isto é, se foram criadas para ler mensagens de um processo ou de uma aplicação, ou escrever mensagens para eles. As portas normalmente precisam transformar as mensagens para posterior transferência, o que implica que elas sejam compostas de tarefas.

Como as tarefas se comunicam entre si por meio de *slots*, implica que uma porta necessariamente tenha *slots*, os quais tornam possível o trabalho assíncrono das tarefas. Já a comunicação entre uma tarefa de uma porta com uma tarefa de um processo também ocorre por meio de um *slot*, porém com características um pouco diferentes e por isto denominado de *interslots* [36]. Um

exemplo simples de solução de integração com a utilização dos blocos construtores do Guaraná DSL pode ser observado na Figura §2.11.

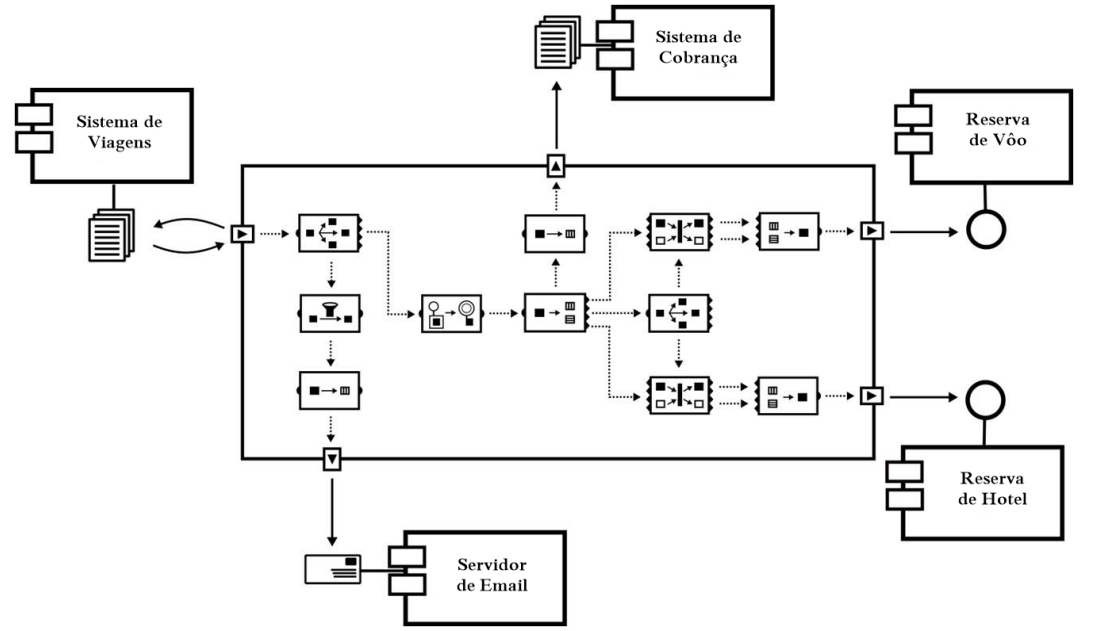


Figura 2.11: *Solução de Integração com o Guaraná DSL*

A tecnologia Guaraná propõe diferentes caixas de ferramentas de tarefas para diferentes contextos de integração, sendo que cada caixa de ferramentas resulta em uma versão diferente do DSL e um editor específico. No entanto, várias tarefas, incluindo várias instâncias da mesma tarefa, podem ser executadas em paralelo. A comunicação direta entre tarefas, deste modo, fica inviável. Em vez disso, elas se comunicam indiretamente por meio de *slots*.

Um outro fator importante da tecnologia Guaraná, é que os modelos obtidos são independentes de plataforma, de modo que os engenheiros não dependem de conhecimentos e habilidades em uma tecnologia de integração de baixo nível ao projetar suas soluções. Assim, os projetos modelados podem ser reutilizados para gerar automaticamente soluções de integração executáveis para diferentes plataformas.

2.4.4 Sintaxe Concreta

As classes fornecidas pela sintaxe abstrata da tecnologia Guaraná, são representadas através da simbologia que compõe a sintaxe concreta [36].

Através destes símbolos é possível expressar todas as classes propostas: Application, Process, EntryPort, ExitPort, IntegrationLink, ApplicationLink, Slot e Task, conforme a Figura §2.12.

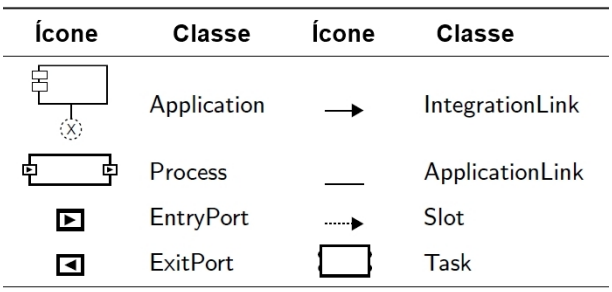


Figura 2.12: *Simbologia da Sintaxe Abstrata do Guaraná (de Frantz e outros [36])*

No entanto a classe Task é representada por um símbolo genérico. Como as tarefas são fornecidas em caixas de ferramentas e portanto elas não fazem parte do núcleo da linguagem.

O símbolo utilizado para representar Task possui entradas e saídas que podem ser observadas como saliências na lateral do ícone. Estas entradas e saídas são utilizadas para conectar tarefas entre si, através de Slots.

Os ícones utilizados para Process, EntryPort e ExitPort também é abreviado. Estas três classes são compostas por outras classes. No entanto, elas podem ser representadas sem a especificação da sua estrutura interna.

2.4.5 Linguagem Baseada em Regras

A Linguagem de Domínio Específico da tecnologia Guaraná está baseada na definição de regras. Esta característica diferencia o Guaraná das demais tecnologias e evidencia sua linguagem para uma utilização mais intensa por engenheiros de diferentes áreas de conhecimento, considerando a facilidade de uso e a possibilidade de inserção de regras específicas de acordo com a necessidade. Este fator, segundo Frantz e outros [35] credita grande importância para a definição das regras.

A sintaxe utilizada para escrever textualmente as regras é composta por duas partes separadas por uma seta, conforme a Figura §2.13. A parte representada do lado esquerdo determina a entrada da regra, enquanto que a parte

ao lado direito determina a saída da regra. Assim, de acordo com as entradas ocorridas do lado esquerdo da representação, espera-se uma determinada saída do lado direito.

$$P_1\,[n_1..m_1]\ \&\ P_2\,[n_2..m_2]\ \ldots\ P_k\,[n_k..m_k] \longrightarrow P_{k+1}\,[n_{k+1}..m_{k+1}]\ \&\ P_{k+2}\,[n_{k+2}..m_{k+2}]\ \ldots\ P_{k+q}\,[n_{k+q}..m_{k+q}]$$

Figura 2.13: *Sintaxe textual para representar regras (de Frantz e outros [35]).*

A representação das entradas e saídas é expressa na forma: P [min..max], onde P refere-se ao nome da porta, e min e max são números naturais que representam o número mínimo e máximo de mensagens que são permitidos na porta P observando sempre que min $\leq$ max. A Figura §2.14 mostra a forma de representar estas cardinalidades.

a) P[+] = P[1 .. Integer.MAX_VALUE]

b) P[*] = P[0 .. Integer.MAX_VALUE]

c) P[?] = P[0..1]

d) P[n] = P[n..n]

Figura 2.14: *Definição de cardinalidades para as regras (de Frantz e outros [35]).*

Na Figura §2.15 pode ser observada a definição de uma regra para um exemplo simples de solução de integração representado na Figura §2.11. Neste caso a regra *R1* está associada a *Process 1* e envolve a porta de entrada *P1* e as portas de saída *P2, P3, P4* e *P5*. A definição atribuída nesta regra define que para cada mensagem que entra na Porta *P1* é necessário que haja uma mensagem correlacionada nas portas de saída *P2, P3, e P4,* enquanto que na porta de saída *P5* pode haver zero ou uma mensagem.

Esta proposta é interessante, pois não está vinculada a uma solução específica ou a um modelo de execução, nem impõe quaisquer limitações práticas, sendo que os algoritmos em que se baseia são computacionalmente tratáveis apresentando bom desempenho quando submetidas a processamento intenso [35].

```
Process 1

R1 = P1[1] ⟶ P2[1] & P3[1] & P4[1] & P5[?]
```

Figura 2.15: *Exemplo de especificação de regras.*

2.4.6 Mecanismo de Tolerância à Falhas

Como uma solução de integração é composta por aplicações distintas, geralmente com características específicas, ela é, portanto, vulnerável a uma variedade de erros que podem fazer com que a mesma se comporte de forma anormal. Estes erros ocorrem devido a falhas, que podem ser: permanentes, devido a defeito de software, ou transiente, devido a um recurso que está temporariamente indisponível [35]. Caso os erros não sejam tratados adequadamente, os mesmos serão percebidos como falhas por usuários finais [19].

Frantz e outros [35] apresentaram um mecanismo de tolerância a falhas para a tecnologia Guaraná DSL baseado em regras, as quais são concebidas e inseridas por um engenheiro de software de acordo com a necessidade e a especificidade de cada solução. Este mecanismo também está estruturado em quatro etapas: *Event Reporting, Error Monitoring, Error diagnosing e Error Recovery*.

O *Event Reporting* reporta eventos sobre o funcionamento de uma solução de integração, relatando se uma porta foi capaz de lidar com uma mensagem ou não. Na etapa de *Error Monitoring*, os eventos são armazenados e analisados para encontrar correlações que posteriormente deverão ser verificadas. Quando um erro é detectado, uma notificação é criada e enviada para a etapa *Error diagnosing*, cujo objetivo é identificar a causa do erro, as mensagens e as partes envolvidas. Por fim, a etapa de *Error Recovery* que busca executar ações de recuperação para ajudar o sistema compensar a existência de falhas e a ocorrência de erros [35].

É na etapa de *Error Monitoring* que as regras inseridas para determinada solução de integração, são analisadas. Esta etapa ilustrada na Figura §2.16 se baseia em um banco de dados (externo ao monitor) chamado *Meta-information* que fornece as informações sobre as soluções que estão sendo monitoradas. O *Monitor* é constituído por dois subsistemas e duas bases de

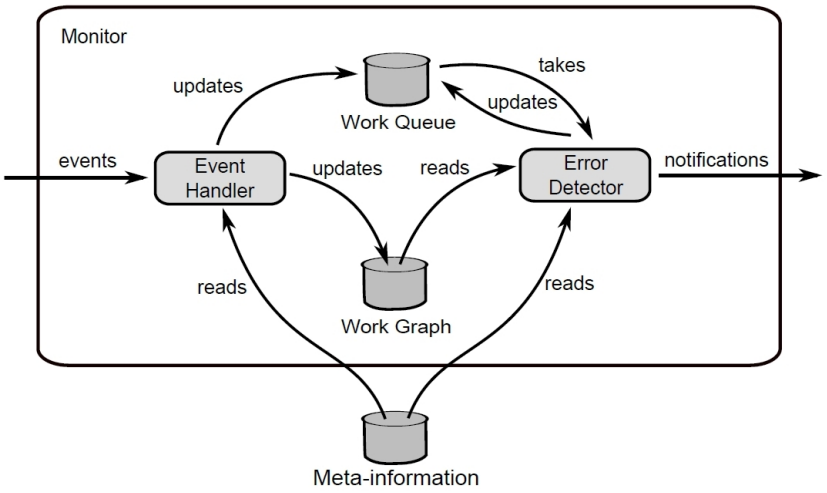


Figura 2.16: *Visão abstrata do monitor de erros (de Frantz e outros [35])*

dados. O *Event Handler* usa eventos de entrada para construir uma estrutura gráfica que é armazenada no banco de dados *Work Graph*; este gráfico mantém o controle das trocas de mensagens e relacionamentos entre os processos. O *Error Detector* é responsável para analisar este gráfico e de encontrar e verificar correlações. O *Work Queue* é utilizado como uma ferramenta intermediária que permite ao *Event Handler* e ao *Error Detector* trabalharem de forma totalmente assíncrona. Todas as vezes que ocorre um evento, o *Error Detector* analisa o banco de dados *Work Graph* em um ponto específico, a fim de encontrar a correlação em que uma mensagem específica esta envolvida é emite uma notificação caso necessário. A validação das correlações é realizada através de regras definidas pelo usuário [35].

Esta proposta, no entanto, ainda carece de formalização. É o formalismo que possibilita a validação das regras definidas pelos engenheiros ou fornece os recursos necessários para que as regras possam ser geradas automaticamente.

2.5 Resumo do Capítulo

Nesse capítulo foram elencados os principais conceitos sobre as temáticas que envolvem a pesquisa. De acordo com a abordagem, a utilização de soluções de integração é importante para que as organizações consigam fornecer

suporte aos processos de negócio de maneira eficiente e precisa. Deste modo, as aplicações que compõe o ecossistema de software são integradas através de uma tecnologia de integração que deve apresentar características de usabilidade. Além disso, foram apresentados os conceitos de tolerância a falhas, que são imprescindíveis em soluções de integração, considerando a complexidade do contexto em que se encontram. Assim, a tecnologia Guaraná apresenta-se como uma ferramenta interessante para prover soluções de integração, por apresentar uma DSL com um alto nível de abstração em seus modelos e um mecanismo de tolerância a falhas baseado em regras.

Capítulo 3

Trabalhos Relacionados

Os investimentos em conhecimento
geram os melhores dividendos.

Benjamin Franklin, Cientista norte-americano (1706-1790)

A partir do estudo sobre a teoria das Linguagens Formais de Noam Chomsky e John Backus, o formalismo passou a ser utilizado para descrever a sintaxe de linguagens de programação. Por volta de 1965 os estudos sobre a Teoria dos Autômatos e a Teoria de Linguagens Formais se consolidaram através do surgimento das especificações formais para o desenvolvimento de Sistemas Computacionais. A ideia que se fomentava era fornecer corretude aos programas a partir dos métodos formais [101]. Porém, nos últimos anos o interesse pela formalização vem aumentando gradativamente, principalmente quanto à formalização de linguagens [16, 46, 102]. Harel e Rumpe [46] destacam a importância da definição de regras claras para sintaxe e uma rígida descrição do seu significado, quando se trata de linguagens de domínio específico. Neste sentido, existem diferentes abordagens e inúmeras obras na literatura que contêm especificações formais para este conjunto de linguagens. Na Seção §3.1 estão descritos os principais trabalhos relacionados que utilizam a Notação Z para formalização de linguagens. A Seção §3.2 apresenta os trabalhos relacionados aos demais métodos formais apresentados nesta dissertação: Alloy, Método B, RSL e Redes de Petri. Por fim, a Seção §3.3 apresenta o resumo de capítulo.

3.1 Notação Z

Diversos trabalhos utilizam a Notação Z como base formal para definir com precisão e clareza as propriedades de linguagens de domínio específico.

Mostafa e outros [82] propõem a formalização da sintaxe de um subconjunto de diagramas UML (diagrama de caso de uso, diagrama de classe e diagrama de máquina de estados) utilizando a notação Z. De acordo com Mostafa e outros [82] a UML é uma linguagem semi-formal, que carece de definições mais precisas para atender as normas e padrões, o que é alcançado com a utilização da Notação Z. Ele ainda destaca que a formalização reduz os riscos e aumenta a segurança e a confiabilidade no desenvolvimento de softwares.

Shroff e France [102] também apresentaram uma formalização por meio da Notação Z para os principais diagramas UML utilizados para construir estruturas de classe, com o objetivo de expressar precisamente o seu significado. Segundo eles, é importante que se tenha uma base semântica formalmente definida, para definir a estrutura e o comportamento de um modelo, o qual pode ser rigorosamente analisado.

Outra abordagem similar é realizada por Kim e David [58]. Utilizando a precisão proporcionada pela notação Z, eles fornecem uma base formal para a estrutura sintática e semântica dos métodos construtores de classe UML, além das regras para o desenvolvimento de um diagrama de classe bem definidas. Com base nesta descrição formal, os construtores de classe UML são depois traduzidos para Object-Z e são aplicadas técnicas de prova para validar os diagramas de classe.

Já Richters e Gogolla [95], apresentam uma semântica formal para a OCL. Esta linguagem faz parte da UML e é utilizada como complemento na modelagem de sistemas de informação, proporcionando a definição de restrições de integridade, ao passo que a UML é utilizada para descrever a estrutura geral. Porém, a OCL é considerada uma linguagem semi-formal, e deste modo as restrições definidas com a mesma, não apresentam uma precisão necessária. Richters e Gogolla [95], atribuíram significado preciso aos conceitos OCL e também a alguns aspectos centrais dos modelos de classe UML. Além disso, consideram que a semântica formal facilita a verificação, validação e simulação de modelos, além de ajudar a melhoras a qualidade dos modelos e projetos de software.

Roe e outros [97] propõem um mapeamento para traduzir sistemas modelados em UML com restrições especificadas em OCL para Object-Z. Zhang e outros [115] apresentam um framework para modelagem orientada a objeto com base em Z, buscando, desta forma, alcançar modelos formais precisos e mais fáceis de serem construídos.

Jiang e Wang [56] propõem a formalização da Linguagem de Modelagem de Domínio Específico XMML, baseado na lógica de primeira ordem. Eles apresentam a especificação formal da semântica estrutural da linguagem, da qual são formalizados os relacionamentos e as restrições com posterior verificação de consistência. Jackson e Sztipanovits [55] apresentaram trabalho similar utilizado a lógica de Horn, incluindo demonstração de como o formalismo apresentado pode complementar as ferramentas existentes. Além disso, proporciona algoritmos para a análise de linguagens de modelagem de domínio específico e para a transformação de modelos.

3.2 Outras Linguagens Formais

As linguagens formais: Alloy, Metodo B, RSL e Redes de Petri, destacadas nesta dissertação, também são utilizadas em trabalhos de formalização de linguagens. A seguir são descritos alguns destes trabalhos.

Em seu trabalho, Anastasakis e outros [4], fazem uso de técnicas baseadas em modelo para a transformação automática de diagramas de classe UML com restrições definidas em OCL para uma especificação formal escrita em Alloy. Além disso, os autores demonstram aspectos que dificultam esta transformação pelas diferenças entre as duas linguagens. Por fim, ainda é mostrado um exemplo de aplicação em um sistema de comércio eletrônico.

Getir e outros [44] utilizam a linguagem de especificação formal Alloy para apresentar a semântica formal para uma Linguagem de Modelagem de Domínio Específico para web semântica de um sistema multi-agente.

Outra aplicação da linguagem de especificação Alloy é encontrada em Svendsen e outros [107], que apresentam a formalização de uma linguagem especifica utilizada para controle de trem. O trabalho descreve o comportamento e as restrições necessárias para controlar de forma automática uma estação de trem. Para demonstrar a eficiência do estudo foram realizadas simulações hipotéticas com intensa circulação de trens a fim de ilustrar a abordagem.

Em Sun e outros [106] encontra-se uma abordagem formal para a especificação e verificação de modelos de recursos. Neste trabalho, os autores

definem a semântica formal para a linguagem de modelagem de recursos usando lógica de primeira ordem. Além disso, fornece uma validação da semântica usando a prova de teoremas através da ferramenta Z/EVES. Por fim, ainda foi demonstrado que a consistência de um modelo de funcionalidades e as suas configurações podem ser verificados automaticamente, por meio da codificação da semântica para a Alloy Analyzer. Schobbens e outros [100] desenvolveram um trabalho semelhante, oferecendo uma semântica formal para os diagramas de recursos livres (FFD) do método FODA (feature oriented domain analysis).

Meyer e Souquières [81] proporcionam uma abordagem sistemática para transformar especificações semi-formais expressas com notações OMT (Object Modeling Technique) em especificações formais utilizando o método B. As transformações foram apresentadas como modelos genéricos, e automaticamente provadas dentro do provador de B.

Em outro trabalho, Ledang e outros [66] também destaca a formalização dos comportamentos de diagramas UML para Notação B. Nesta contribuição, os autores propõem a derivação automática de diagramas UML para especificações em B através da utilização de frameworks para tradução. Ledang e Souquières [65] mostram técnicas para integração de UML e especificação B, através da transformação sistemática de expressões OCL para B.

Mauco e outros [74] propõem um conjunto de regras para converter especificações LEL (Language Extended Lexicon) em RSL. De acordo com os autores, uma linguagem natural, como por exemplo LEL, é útil durante os primeiros estágios de desenvolvimento de software. Já os métodos formais ajudam a aumentar a qualidade e a confiabilidade do software. Neste sentido o trabalho propõe uma estratégia que consiste em um conjunto de regras de derivação baseada em RSL que fornecem uma maneira sistemática e consistente de transformar as informações contidas no LEL em tipos abstratos e concretos RSL. Em outro trabalho os autores buscaram a formalização de algumas das heurísticas para derivar tipos RSL do LEL. Este trabalho pretende servir como base para uma estratégia semi-automática que pode ser implementada por uma ferramenta [75].

Outro exemplo de utilização de RSL para especificação é encontrado em Debnath e outros [26]. A partir de um modelo UML o trabalho propõe a transformação de restrições expressas em OCL em expressões RSL, através da utilização de um conjunto de regras. De acordo com os autores, a verificação de propriedade do modelo descrito pelo diagrama de classe UML e pelas invariantes OCL, pode agora ser realizada por técnicas com suporte do método RAISE.

Outra abordagem pode ser encontrada em López-Grao e outros [71]. Eles utilizam redes de Petri para a formalização de diagramas UML. A proposta deste trabalho é oferecer uma semântica formal para os diagramas de atividade.

3.3 Resumo do Capítulo

Nesse capítulo foram apresentados alguns trabalhos encontrados na literatura que utilizam a formalização de linguagens. Dentre as abordagens, nota-se uma incidência maior na utilização de determinadas linguagens de especificação formal, como por exemplo, a Notação Z e Alloy. Além disso, vários estudos abordam a especificação formal da linguagem orientada a modelo UML associada com restrições definidas em OCL. Contudo, o formalismo é considerado uma obrigação por muitos autores, independente de linguagem, pois permite que técnicas poderosas de análise para modelos de software sejam feitas.

Parte III

Pesquisa Desenvolvida

Capítulo 4

Linguagens e Métodos Formais

Não existem métodos fáceis
para resolver problemas difíceis.

René Descartes, Matemático francês (1596-1650)

A formalização de uma Linguagem de Domínio Específico parte da definição de um método de especificação formal, com características que possibilitam representar as especificidades da linguagem a ser formalizada. Neste sentido, este capítulo tem por objetivo apresentar as principais linguagens e métodos formais. Na Seção §4.1 é realizada uma contextualização a partir da concepção das linguagens formais e a importância da sua utilização na área de Engenharia de Software. A Seção §4.2 apresenta uma classificação das linguagens formais relacionando características em comum. Na Seção §4.3 são apresentados alguns dos principais métodos formais com características desejáveis ao contexto do problema. A Seção §4.4 apresenta uma análise para comparação dos métodos formais com mais similaridade com as propriedades a serem representadas na especificação formal. Por fim, a Seção §4.5 apresenta o resumo do capítulo.

4.1 Contextualização

Aproximadamente dez anos após o surgimento do conceito de computação através da máquina de Turing, Noam Chomsky e John Backus desenvolveram uma pesquisa para descrever teoricamente as linguagens naturais, por meio da Teoria das Linguagens Formais [78]. Este estudo, no entanto, logo pôde ser aplicado em linguagens artificiais originárias na ciência da computação. A partir deste estudo as linguagens formais passaram e ser utilizadas para descrever a sintaxe de linguagens de programação, através do formalismo de descrição de Sintaxe [101].

Por volta de 1965 os estudos sobre a Teoria dos Autômatos e a Teoria de Linguagens Formais se consolidaram através do surgimento das especificações formais para o desenvolvimento de Sistemas Computacionais. A ideia que se fomentava era fornecer corretude de programas a partir dos métodos formais [101]. Além disso, as linguagens formais tem aplicações em análise léxica e sintática de linguagens de programação, desenhos de hardware e relacionamentos com linguagens naturais [78].

Uma definição mais simplificada para linguagens formais é dada por Moura [83], que as define como sendo o estudo de modelos matemáticos que tornam possível que se construa uma especificação formal por meio de uma linguagem específica, que pode ser reconhecida, e com ela pode-se definir estruturas, propriedades, características e relacionamentos. Estes estudos sobre as linguagens formais e autômatos formam a base teórica da computação, com influência direta em linguagens de programação, computadores e também no reconhecimento de linguagens naturais [90]. Segundo Ramos e outros [90] as linguagens formais podem ser vistas como conjuntos e por consequência existe uma relação fundamentada na teoria dos conjuntos da matemática discreta.

Neste sentido, as linguagens e métodos formais fornecem técnicas matemáticas para auxiliar no desenvolvimento, manutenção e documentação destes sistemas. Pressman [88] atribui aos métodos formais características importantes para a integridade do sistema como a detecção precoce de falhas e a especificação mais completa e consistente dos requisitos, não permitindo ambiguidades.

Segundo Almeida e outros [3], o principal objetivo da utilização de métodos formais é a possibilidade de garantir o comportamento de um determinado sistema ou solução de integração. Por meio da especificação formal, é

possível descrever o comportamento desejado, além de definir o que deve ser implementado, ou garantir que o comportamento do sistema está de acordo com o que se espera. Neste sentido a especificação formal pode ser realizada em dois níveis de abstração: alto e baixo. O nível mais alto (abstrato), alcança um modelo com a descrição do comportamento esperado do sistema. O nível mais baixo fornece uma abordagem mais operacional onde a descrição ocorre de forma implícita, contida pelo modelo.

Contudo, por meio da especificação formal pode-se construir um sistema ou garantir que um sistema já construído está fornecendo suas funcionalidades de acordo com o esperado. Isto pode ser obtido através da apresentação da prova formal de suas propriedades [83].

Neste sentido, para o desenvolvimento da uma especificação formal são utilizados métodos formais, os quais podem ser divididos em: *métodos formais leves*, que não exigem conhecimentos profundos e se utilizam de ferramentas que automatizam os processos; e *métodos formais pesados*, que geralmente, são mais complexos, porém com mais recursos e aplicados na especificação formal de sistemas mais robustos onde sua utilização e custos se justificam [3].

Segundo Holloway [50] são os métodos formais que fornecem a base para uma definição precisa com coerência, integridade, especificação, implementação e correção. De maneira geral os engenheiros de software utilizam os métodos formais para o desenvolvimento de software desde a fase de análise de requisitos, passando pela especificação e projeto até a fase de desenvolvimento. Outros, porém, utilizam as técnicas formais para construir modelos de sistemas já desenvolvidos, buscando uma abstração mais matemática, a fim de explicar o comportamento de um sistema [12].

Conforme Bjørner e Henson [12] a especificação formal é baseada na matemática e tem aprovação de grande parte dos engenheiros de software. Isso ocorre devido a sua funcionalidade, ao passo que os modelos de engenharia de software em geral estão mais preocupados com desempenho e confiabilidade.

Geralmente utilizadas em sistemas críticos como controle de tráfego aéreo, sistemas médicos, sinalização ferroviária, entre outros [11], as linguagens de especificação formal também podem ser utilizadas para garantir o comportamento correto de um sistema ou de uma solução de integração, sendo que o principal interesse é torná-los tolerantes a falhas.

Caracterizados principalmente pela construção de sistemas mais confiáveis e com menos erros, as linguagens de especificação formal se diferenciam

umas das outras pelas suas propriedades. No entanto, algumas destas propriedades são comuns a todas as linguagens. São elas: construção de sistemas mais confiável, com menos erros; melhorar a manutenção de software; oferecer suporte formal para o desenvolvimento desde a especificação até a implementação; abstração, verificação e refinamento sistêmico; realizar justificativas (regras de prova) e prover uma documentação mais detalhada; manipulação de concorrência; oferecer suporte a modularização e conceitos de orientação a objetos, tais como polimorfismo, herança e encapsulamento; criação de módulos reutilizáveis [50].

Almeida e outros [3], no entanto, defendem a utilização de métodos formais no desenvolvimento de software. Com isso, os engenheiros de software são forçados a pensar em todas as especificidades do sistema e descrevê-las detalhadamente evitando ambiguidades em seu entendimento. Outra propriedade citada por Almeida e outros [3] é a possibilidade de executar uma especificação e observar diretamente o seu comportamento sem que se necessite implementar o sistema. De acordo com o autor, utilizando-se de um protótipo é possível realizar a validação do sistema. Este protótipo é considerado uma entidade formal, passível de ser manipulada matematicamente, verificando-se a sua prova através de teoremas ou verificação de modelos.

4.2 Categorização

De acordo com Almeida e outros [3] duas abordagens são realizadas quando se trata de métodos formais. A primeira parte do princípio da definição clara das modificações realizadas por cada operação sobre o estado atual do sistema modelado. Nesta abordagem são modeladas as operações, mecanismos disponíveis (serviços), ou ações que podem ou não ser executadas. São as linguagens orientadas a modelo. A segunda abordagem está voltada para a especificação dos dados manipulados, como eles evoluem, ou a forma como eles estão relacionados. Esta classe de especificações inclui especificações algébricas.

Bjørner e Henson [12] também definiram dois grandes grupos de linguagens:

Orientadas a Modelo: expressam a especificação em termos de construção matemática, isto é, a partir de modelos matemáticos, como conjuntos, cartesianos, listas e funções. Estes métodos são adequados para especificação de estruturas complexas por meio de funções simples. Como exemplos de linguagens formais orientadas a modelo destacam-se: VDM, Notação Z, Método B e Redes de Petri.

Orientadas a Propriedades: caracteriza-se por expressar em termos de propriedades lógicas as especificidades do sistema. Geralmente são utilizadas em formas de axiomas para definir o conjunto mínimo de propriedades que determinado sistema deve satisfazer. São exemplos de linguagens orientadas a propriedades: OBJ3, CafeOBJ e CASL.

Já Nami e Hassani [85] definiram cinco categorias de linguagens formais conforme as suas características:

Orientadas a Modelo: caracterizam-se pela construção de um modelo matemático que representa o sistema. Este modelo descreve o estado e as operações possíveis sobre estes estados. As operações são funções que manipulam o valor atual do estado através de parâmetros que definem o seu novo valor. Este tipo de linguagem normalmente é utilizada para descrever em detalhes, objetos matemáticos específicos, como estruturas de dados ou funções. São exemplos de linguagens orientados a modelo o VDM e a notação Z.

Algébrica: utilizadas para especificar sistemas de informação através da álgebra abstrata. Elas descrevem as características principais dos sistemas de informação sem prejuízo de propriedades. Um exemplo de linguagem de especificação algébrica é o ActOne. Ele possibilita provar o processo de desenvolvimento com detalhes de implementação.

Orientadas a Processo: utilizadas para descrever sistemas concretos baseados em um modelo implícito específico para a concorrência. As expressões desta linguagem descrevem processos simples e são construídas a partir de suas expressões elementares. Suas operações produzem processos mais complexos. São exemplos de linguagens Orientadas à Processos: CCS e CSP.

Imperativa: caracterizam-se principalmente pela simplicidade e pelo aspecto restrito em que suas funções são escritas. Com isso, obtêm-se funções que facilmente se transformam em expressões compreendidas computacionalmente. Como as especificações de funções que definem uma propriedade são bastante restritas nessa linguagem, a sua prova se torna mais eficiente e simples.

Lógica: estas linguagens não foram utilizadas inicialmente para especificação de sistemas de informação. Porém, dividida em dois grupos de uso, ela pode ser empregada na especificação na forma de axiomas ou por meio da utilização de notações matemáticas mais rigorosas tendo, com isso, boa base formal. Um exemplo de linguagem lógica é a Notação Z que, baseada na teoria dos conjuntos, utiliza axiomas e notações matemáticas bem definidas.

Além destas cinco categorias, existe ainda um grupo de linguagens de especificação formal hibrido, ou seja, possui as características de duas ou mais categorias. A linguagem RSL é um exemplo disto. Ela possui suporte às categorias: orientada a modelo, algébrica, orientada a processos, imperativa e lógica. Outra linguagem hibrida é a linguagem LOTOS, que possui as características das categorias: Orientada a Modelo e Métodos Algébricos. Esta linguagem é uma notação destinada, principalmente, à especificação de protocolos de comunicação.

4.3 Métodos

Atualmente a engenharia de software vem tendo uma nova abordagem com a utilização dos métodos formais. Conforme mencionado anteriormente, os métodos formais podem ser aplicados em diferentes etapas do processo de construção do software. Mas, de maneira geral, são empregados para tornar os sistemas mais concisos associando características de tolerância a falhas.

Com isso, o estudo dos métodos formais tem avançado muito nos últimos anos. No entanto, ainda são raros os trabalhos que apresentam uma comparação efetiva entre os métodos, com o objetivo de diferenciá-los, indicando a melhor aplicação de cada. Neste sentido, foi realizada uma comparação entre as principais e mais utilizadas linguagens formais: Notação Z, B, Alloy, RSL e Redes de Petry.

4.3.1 Notação Z

A Notação Z é uma das linguagens formais mais utilizadas no contexto da engenharia de software. Desenvolvida na Universidade de Oxford no final da década de 1970 pelos membros do grupo de pesquisa de programação, Z é uma linguagem baseada em lógica de primeira ordem e na teoria dos conjuntos. Apresenta uma estruturação com blocos construtores definindo: tipos básicos, definição axiomática e Schemas, o que possibilita a modularização de uma especificação formal [83].

Características como composição, agregação e herança também são encontradas na Notação Z. Moura [83] destaca que a composição é caracterizada na Notação Z, quando ocorre uma sequência de transformações de estado de esquemas em sucessão através da mudança de valores de certas variáveis de estado do sistema. A ação consecutiva de dois ou mais esquemas pode ser capturada por um novo esquema. Sendo assim, este novo esquema poderia ser decomposto na ação dos esquemas componentes, quando aplicados na sequência correta.

Nami e Hassani [85] destacam características encontradas na Notação Z, como polimorfismo, herança e encapsulamento, importantes para reutilização de componentes e por combinar várias abstrações.

Outra característica desejável em uma notação formal é a sua capacidade de compreensão, pois uma especificação deve ser compreendida por especialistas de diferentes áreas de atuação. Isto se torna ainda mais importante quando esta especificação define as propriedades de uma DSL, sendo que os modelos construídos a partir dela, nem sempre são feitos por especialistas na área de computação. Neste sentido, Spivey [105], destaca que a mais complexa das definições realizadas na Notação Z não é nada mais que teoria matemática organizada de uma maneira estruturada, e consequentemente utiliza conceitos matemáticos comuns e de fácil compreensão.

Em relação a ferramentas disponíveis para especificação, formatação, verificação de tipo e provas na Notação Z, destacam-se algumas: Ferramenta Z/EVES é uma ferramenta interativa para análise, verificação de tipo, cálculo de pré-condições, provas de refinamento e provas de teoremas [76]. A ideia básica da ferramenta Z/EVES é a facilidade de usar e provar com a ajuda de um editor gráfico. Outra ferramenta é a HOL-Z usada, principalmente, para a prova de teoremas. Com ela é possível importar uma especificação escrita em latex para a verificação de tipo e prova de teorema [29]. Cádiz [110] é um conjunto de ferramentas baseado em UNIX para verificação e composição de especificações na Notação Z. Z Type Checker (ZTC) e a ferramenta Fuzz também suportam a Notação Z, além da verificação de tipo das especificações. RoZ gera automaticamente os esqueletos de esquemas da Notação Z correspondentes a um diagrama de classe UML [76].

4.3.2 Método B

O Método B, criado por J. Abrial para especificação e desenvolvimento de software com apoio de ferramentas, utiliza a noção de máquinas abstratas [2]. Algumas das principais características são apresentadas por Snook e Butler

[103]. O Método B propõe o desenvolvimento de sistemas com base em refinamentos sucessivos, desde a especificação em um nível mais abstrato de um sistema até a sua implementação.

Outra característica importante citada pelos mesmos autores, é que uma especificação pelo Método B pode utilizar vários módulos ao invocar as operações abstratas das suas máquinas. Um componente no Método B também permite que um estado abstrato possa ser dividido em várias partes, de forma que essas partes possam ser encapsuladas. No entanto, o Método B não possui suporte à algumas características da orientação a objeto [103].

Em relação às ferramentas que oferecem suporte ao Método B, Attiogbe [7] destaca a utilização da ferramenta Atelier-B que possui um provador de teoremas que possibilita o uso operacional do Método B para construção de software livre de defeitos (software formal). A ferramenta é capaz de gerar condições de verificação para máquinas abstratas B que permitem provar a conformidade comportamental. Já ferramenta ProB apresenta características semelhantes ao Atelier-B, mas se destaca por oferecer funcionalidades para exibir a visualização gráfica dos autômatos referentes ao modelo especificado.

Existe ainda a ferramenta B-Toolkit que oferece suporte para a escrita e especificação no Método B. Além disso, a ferramenta possui provador de teoremas e animador para máquinas abstratas, possibilitando o refinamento e implementação correta do software especificado [62].

4.3.3 Alloy

Alloy é um método formal proposto por Daniel Jackson do Massachusetts Institute of Technology (MIT) aplicado à especificação e análise de software com a intenção de proporcionar um método que facilite a aplicação de métodos formais na indústria de software. O motivo da pouca utilização de métodos formais citada por Jackson é a sintaxe matemática, que geralmente intimida os projetistas de software [54].

Alloy é uma linguagem de especificação baseada na Notação Z, da qual herdou as principais características, no entanto, com uma notação mais leve [54]. Segundo Jackson [53] uma notação leve é caracterizada por uma sintaxe mais próxima das linguagens naturais ou de programação normalmente conhecidas por programadores e projetistas. No entanto, a notação utiliza menos simbologia matemática.

Getir e outros [44] destacam que Alloy tem uma boa capacidade de descrição com a apresentação de uma linguagem declarativa baseada em lógica de

primeira ordem para definir estruturas e comportamentos de sistemas complexos. Em Alloy tudo é considerado como uma relação e, portanto, não propõe uma lógica especializada para máquinas de estado e concorrência, visando manter a simplicidade. Alloy também é baseado na utilização de contra-exemplos para detectar as falhas do sistema.

Duas outras características importantes em Alloy são destacadas por [54]. A primeira característica cita o suporte à extensão por adição de campos, semelhante à herança em uma linguagem orientada a objetos. A segunda refere-se a reutilização de fórmulas de parametrização explícita, semelhante às funções em uma linguagem de programação funcional. Além disso, o autor destaca que Alloy é uma notação ASCII puro e não requer ferramentas especiais para edição.

Com suporte a Alloy, Alloy Analyzer é uma ferramenta que possibilita a análise automatizada, baseada na lógica de primeira ordem. A ferramenta permite a análise das propriedades do sistema através da busca de instâncias do modelo, passíveis de verificar se algumas das propriedades do sistema são satisfeitas. Isto é realizado por meio de um demonstrador que aplica um procedimento sistemático de busca através de contra-exemplos para explorar o espaço de um determinado domínio, procurando atribuições que satisfaçam as fórmulas em questão (SAT). O utilizador, no entanto, tem que definir o domínio de alcance do modelo [54].

Outra ferramenta é a UML2Alloy que permite a tradução de diagramas de classes UML/OCL em modelos Alloy. Desenvolvida seguindo a metodologia MDA (Model Driven Architecture), emprega o conceito de metamodelo, sendo que os diagramas de classes UML/OCL são submetidos à ferramenta UML2Alloy a qual implementa um conjunto de regras de transformação que permitem produzir modelos Alloy [13].

4.3.4 RAISE - RSL

RAISE (Rigorous Approach to Industrial Software Engineering) é um método formal que fornece facilidades para o uso industrial de métodos formais no desenvolvimento de sistemas de software. A tecnologia RAISE foi desenvolvida como trabalho coletivo dos projetos RAISE e LaCoS [43].

Uma descrição das principais características do Método Raise (RSL) é encontrada em George e outros [43]. Segundo os autores, a linguagem RSL tem suporte à composição, reuso, concorrência e serialização. A modularização (criação de módulos reutilizáveis) e a orientação a objeto proporcionam características à linguagem como o polimorfismo, herança e encapsulamento.

Estas características são consideradas para a construção de softwares mais confiáveis com menos erros, facilidade de manutenção, suporte formal para o desenvolvimento e implementação, abstração, verificação e refinamento.

Este método tem suporte à diferentes estilos ou categorias. Como é uma linguagem mista, com formação composta de outras linguagens formais (como VDM, Z e CSP), além das características próprias ela possui caraterísticas comuns as outras linguagens e pode ser classificada como: orientada a modelo, orientado a processo, algébrica, imperativa e lógica [85].

O método de desenvolvimento RAISE compreende, além de outras características, a formulação de especificações abstratas. Outra característica importante é o refinamento, sendo que se sustenta a partir de uma especificação inicial, a qual pode ser evoluída em direção a algo que possa ser implementado em uma linguagem de programação[43].

Além da linguagem de especificação e do método de desenvolvimento, existem diversas ferramentas que oferecem suporte a utilização de RAISE. Elas são utilizadas para a edição, apresentação de justificativas (demonstrações), tradução em linguagens imperativas e suporte à documentação. George [41] apresenta a ferramenta *rsltc*. Fasie [30] descreve *eRaise*, um plugin para Eclipse com suporte a linguagem RSL. A linguagem e ferramentas focam no suporte às etapas de especificação, projeto e implementação do processo de desenvolvimento de software [40].

A linguagem formal RSL foi projetada para: dar suporte a especificações grandes, modulares; fornecer diversos estilos de especificação (axiomática e baseada em modelos, aplicativa e imperativa, sequencial e concorrente); e suportar especificações variando do abstrato (próximo aos requisitos) ao concreto (próximo à implementação) [43], [47].

RSL oferece uma notação rica, baseada em matemática, na qual requisitos, especificações e etapas do projeto de software podem ser formulados e verificados. RSL é uma linguagem de amplo espectro, podendo ser utilizada para expressar tanto especificações abstratas, de alto nível quanto, projetos concretos, de baixo nível. Permite ainda a especificação e projetos modularizados de sistemas de grande porte e o desenvolvimento separado de subsistemas [42].

4.3.5 Redes de Petri

Uma rede de Petri caracteriza-se pela utilização de uma técnica de modelagem que permite a representação de sistemas utilizando uma robusta base

matemática [92]. Segundo Maciel e outros [72], é possível, através da utilização de redes de Petri, modelar sistemas paralelos, concorrentes, assíncronos e não-determinísticos. Como a notação é formal, pode-se aplicar algoritmos de análise sobre a rede com o intuito de verificar e validar propriedades do sistema.

Apesar dos benefícios de se utilizar redes de Petri na modelagem de sistemas computacionais, a especificação de sistemas através de redes de Petri não oferece mecanismo ou notação que permita especificar a estrutura do sistema. Elas não consideram relações de associação, agregação e herança entre seus elementos [72]. Também não há medida de tempo ou fluxo de tempo numa rede de Petri, o que define uma característica assíncrona [84].

Neste sentido, algumas extensões são necessárias para suprir estas carências: as redes de Petri coloridas, hierárquica e temporizadas determinísticas. Além disso, alguns autores atribuem pontos negativos as redes de Petri, porque costumam gerar modelos extensos e complexos [72].

Em relação a ferramentas com suporte para a criação, edição, simulação e análise de redes de Petri, destacam-se as ferramentas TimeNET [117] e PIPE [27]. No entanto, um grande número de ferramentas está disponível em Heitmann e outros [48].

4.4 Analogia

Dentre as linguagens formais apresentadas, verifica-se que a notação Z apresenta maior representatividade no contexto da formalização de uma DSL. Por ser a primeira linguagem formal desenvolvida, ela apresenta um grande embasamento técnico e teórico, inclusive servindo como base para diversas outras linguagens, como o método B, Alloy e RSL, por exemplo.

Outro fator relevante que deve ser considerado é a facilidade de compreensão de uma especificação. Como uma DSL é construída para servir a uma diversidade de especialistas de diferentes áreas de estudo, ela deve ser compreendida por todos. Neste sentido a Notação Z está baseada na simplicidade da teoria dos conjuntos e na lógica de primeira ordem utilizando uma representação sintática muito próxima da simbologia usual da matemática.

Aliado a isto, observa-se algumas deficiências ou características indesejadas para a especificação de uma DSL, nas demais linguagens quando comparadas com a Notação Z.

O método B é baseado na Notação Z e foi desenvolvido para a representação de sistemas seguindo a noção de máquinas abstratas.

Alloy tem uma sintaxe muito próxima de uma linguagem de programação, apresentando uma maior complexidade de entendimento.

A linguagem formal RSL foi desenvolvida agrupando características de diversos outros métodos formais. Possui características interessantes, porém é menos difundida se comparada com as outras linguagens. Semelhante a Alloy, possui uma linguagem mais próxima das linguagens de programação.

Por fim, as Redes de Petri tem características especificas para a representação de sistemas, mostrando pouca usabilidade para especificação de linguagens. Neste sentido, algumas variações apresentadas no texto são necessárias para aumentar a abrangência da linguagem.

4.5 Resumo do Capítulo

Nesse capítulo, foi apresentado um breve histórico das linguagens formais. Além disso, discutiram-se algumas categorizações de acordo com a abordagem realizada por diferentes autores, cada qual fazendo a classificação dos métodos conforme características específicas. Em seguida foram apresentadas as principais linguagens e métodos formais, detalhando as suas características. Por fim, foi realizada uma analogia entre as linguagens apresentadas. Com isso, observou-se que a notação Z oferece maior usabilidade para aplicação ao problema.

Capítulo 5

Notação Z

O livro da natureza foi escrito exclusivamente com figuras e símbolos matemáticos.

Galileu Galilei, Matemático italiano (1564-1642)

A Notação Z oferece os recursos necessários para a formalização de linguagens. Sendo a primeira linguagem formal a ser estabelecida, ela é base de diversas outras linguagens formais hoje existentes. Neste capítulo estão descritas as propriedades da Notação Z. A Seção §5.1 introduz um breve histórico abordando as diferentes situações de aplicabilidade desta notação. A Seção §5.2 fornece o conceito de tipos abstratos e primitivos. A Seção §5.3 trata das relações e das funções como objeto de especificação. A Seção §5.4 introduz o conceito de agrupamento de declarações definidas como esquemas. A Seção §5.5 define a prova formal e detalha a sua importância para a validação de uma especificação. E a Seção §5.6 apresenta o resumo do capítulo.

5.1 Formalismo

A utilização de uma notação formal para descrever sistemas de informação, linguagens de domínio específico ou de propósito geral, é vista como um fator determinante para atribuir qualidade ao processo de desenvolvimento de um sistema ou para dar suporte formal a uma linguagem. A expressividade alcançada com a especificação formal é completa, sendo que a mesma é clara e precisa [105].

Por meio da especificação formal consegue-se alcançar modelos precisos em conformidade com as especificações e limitações pré-definidas. Com isso, temos um código mais preciso e confiável para a implementação real através dos modelos [44].

A notação Z é utilizada para descrever e modelar sistemas computacionais proporcionando rigor e precisão aos modelos que descreve. Esta linguagem formal foi proposta inicialmente por Jean-Raymond Abrial em 1977 e desenvolvida no Grupo de Pesquisa de Programação na Universidade de Oxford, Inglaterra [83].

Moura [83] destaca algumas vantagens na utilização da Notação Z. Segundo o autor, o fato da Notação Z estar baseada na teoria dos conjuntos e na lógica de primeira ordem, facilita a compreensão das especificações pelos engenheiros e cientistas sem conhecimento em linguagens de programação. Uma especificação formal com a Notação Z se aproxima dos conceitos matemáticos fundamentais utilizados no meio acadêmico e científico. Segundo Spivey [105], a mais complexa das definições realizadas em Z não é nada mais que teoria matemática organizada de uma maneira estruturada.

Outro fator importante destacado pelo mesmo autor é que formalidade proporcionada pela matemática preenche um requisito fundamental de uma formalização: rigor matemático para especificar as propriedades e prova de teoremas para validar a especificação.

Além disso, várias referências destacam que a Notação Z é base para outros métodos formais, o que lhe atribui maior conhecimento e usabilidade. Getir e outros [44] salientam que Alloy foi inspirado em Z; Brien e Martin [17] mostram que muitas características fundamentais de Z também estão presentes em B com a exceção de esquemas.

Getir e outros [44] atribuem a popularidade e a flexibilidade da Notação Z à sua noção de módulos por meio de esquemas. Este fator contribui para a

sua utilização na transformação de metamodelos UML, os quais possuem características similares. Estes esquemas são utilizados de forma button/up, ou seja, define-se os esquemas básicos com características mais especificas, os quais podem fazer parte dos esquemas mais abrangentes [83].

Portanto, a Notação Z apresenta-se como uma das principais referências para formalização. Baseada na Teoria dos Conjuntos e na Lógica de Predicados de Primeira Ordem, a Notação Z transfere às suas especificações as propriedades da matemática e o seu rigor formal é incorporado na construção dos modelos. O anexo §A apresenta os principais símbolos e expressões da Notação Z.

5.1.1 Teoria dos Conjuntos

A teoria dos conjuntos é a base da Notação Z. É obrigatório, assim, conhecer os conceitos envolvidos na definição de um conjunto, nas relações e nas operações entre eles.

Moura [83] define conjunto como agrupamento de elementos com características em comum. Em Z, define-se as características dos elementos atribuindo tipos abstratos. Assim, elementos de tipos abstratos iguais podem fazer parte do mesmo conjunto.

Na Figura §5.1 (a) observa-se um exemplo de conjunto bem definido em relação a um conjunto de tarefas. Isto não ocorre na Figura §5.1 (b), pois um elemento do tipo process não faz parte do conjunto de tarefas.

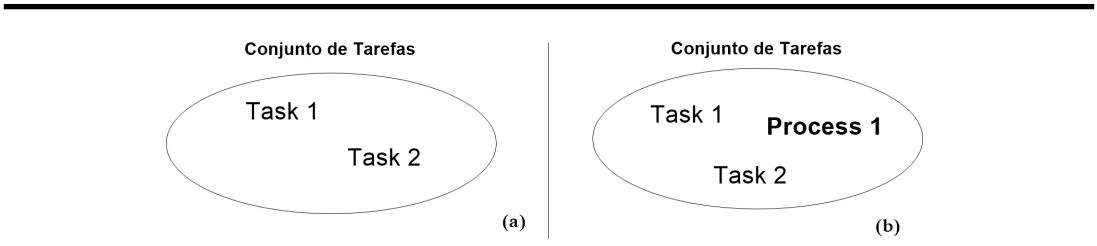


Figura 5.1: *Exemplos de conjuntos*

A partir da formação de conjuntos bem definidos, é possível estabelecer relações e operações sobre estes conjuntos. Uma relação, por exemplo, pode ocorrer entre elementos e conjuntos ou apenas entre conjuntos.

A relação entre elementos e conjuntos pode ser entendida quando definimos que o elemento *Task 1* pertence ao conjunto de tarefas conforme a Figura §5.1. Já uma relação entre dois conjuntos é caracterizada quando consideramos, por exemplo, que o conjunto de tarefas em (a) está contido no conjunto em (b) referente a Figura §5.1.

Existem ainda as operações que podem ser aplicadas sobre os conjuntos. A união e a intersecção são as principais operações. Moura [83] lembra que para que seja possível exercer qualquer operação sobre conjuntos em Z, é necessário que estejam bem definidos, e que sejam vinculados aos mesmos tipos abstratos. Portanto, não é possível fazer a união entre um conjunto de tarefas e um conjunto de processos. Também destaca-se que as operações ocorrem entre conjuntos e não entre elementos e conjuntos.

5.1.2 Lógica de Primeira Ordem

Além da teoria dos conjuntos a notação Z também utiliza-se da lógica de primeira ordem. A noção de lógica matemática em Z está basicamente relacionada com o fato de termos expressões que podem assumir um valor verdadeiro ou um valor falso [83].

Pode-se também escrever expressões que combinem certos predicados, elaborando novas expressões. Para estas expressões são utilizados operadores conectivos. Moura [83] destaca os conectivos mais utilizados: a conjunção, a disjunção, a implicação, a equivalência e a negação, e descreve a sua tabela verdade. Além disso, é possível ainda construir expressões utilizando vários conectivos formando predicados mais complexos [83].

5.2 Tipos

A linguagem Z se apoia na noção matemática de conjunto e em princípios de lógica de primeira ordem. No entanto, duas características que podem ser encontradas em conjuntos são indesejáveis para uma linguagem formal: elementos de naturezas distintas e alguns paradoxos [83].

Para evitar os problemas citados, a Notação Z é baseada na noção de TIPOS. Segundo Moura [83], a regra básica para esta abordagem é que cada elemento (variável) deve estar vinculado a um único tipo em Z. É o tipo de um objeto que determina um conjunto de dados ou valores a serem assumidos.

Assim, todo objeto utilizado em uma especificação deve ser previamente declarado, sendo-lhe associado um determinado tipo. Este objeto permanecerá associado ao tipo por toda a especificação, não sendo possível altera-lo.

Outro fator importante destacado por Moura [83] é que um objeto somente poderá participar de expressões ou predicados cujos operadores estão atuando sobre elementos do mesmo tipo. Contudo, tem-se o ponto de partida de uma especificação e esta deve conter um ou mais tipos iniciais, ou seja, tipos em que a estrutura interna não é detalhada. Assim, declarados os tipos, é possível desenvolver o restante da especificação, pois as variáveis (elementos) podem ser declaradas e devem ser associadas a um determinado tipo. Além disso, podem ser definidos predicados e expressões com restrições sobre sobre a parte declarativa.

5.3 Relações e Funções

Para aumentar a expressividade e a capacidade de trabalhar com sistemas mais complexos, a Notação Z também faz uso de relações entre elementos de conjuntos diferentes. A especialização destas relações pode ser definida como sendo uma dependência funcional.

Uma relação é caracterizada pela associação entre objetos de um conjunto origem e objetos de um conjunto destino. Uma função, no entanto, tem características mais específicas. Uma função relaciona um elemento do conjunto origem com exatamente um elemento do conjunto destino de acordo com uma determinada restrição [83].

Moura [83] destaca que a dependência funcional é extremamente útil em uma linguagem formal. Por meio dela consegue-se formalizar grande parte das ideias e propriedades dos mais diversos sistemas, sempre de maneira concisa e precisa. Um exemplo de relação simples pode ser observado na Figura §5.2 (a). Na Figura §5.2 (b) é mostrado um exemplo de função sobrejetora.

A especificação de relações e funções em Z é dada por pares de elementos. Estes pares são elementos de um novo conjunto sobre o qual podem ser aplicados todos os operadores já dispostos para conjuntos. Para caracterizar uma representação de uma função é utilizada uma simbologia específica, ou são utilizadas restrições impostas à elementos formados por relações simples [83].

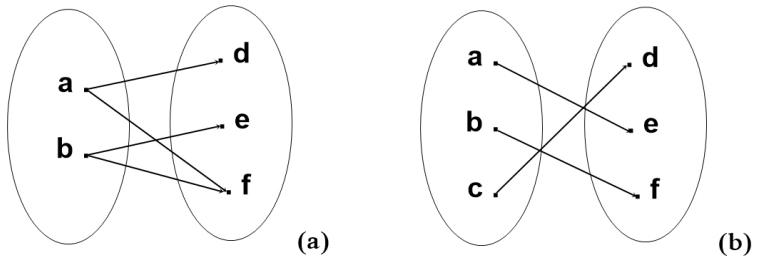


Figura 5.2: *Relação (a) e Função (b) entre dois conjuntos*

5.4 Esquemas

De acordo com Moura [83] ao realizar uma especificação tem-se uma estrutura formada por declarações e restrições. Ao especificar um sistema mais elaborado através de uma abordagem sequencial tem-se a repetição de muitas destas declarações e predicativos, gerando uma especificação extensa e confusa.

Neste contexto, a notação Z utiliza mecanismos notacionais para agrupar as declarações e as respectivas restrições, caso exista relação entre elas. Para obter este efeito, a Notação Z utiliza-se de esquemas.

Por meio dos esquemas proporcionados pela Notação Z é possível escrever trechos recorrentes na especificação e utilizá-los sem a necessidade de reescrevê-los. Além disso, é possível escrever especificações mais complexas partindo de especificações básicas, tornando-a mais enxuta e de fácil entendimento.

A Figura §5.3 apresenta a estrutura de um axioma. Neste sentido, é possível verificar que a Figura §5.3 (a) ilustra uma visão geral do axioma composto de uma parte declarativa (parte superior) a qual introduz a declaração das variáveis e de uma parte predicativa (parte inferior) com as cláusulas que apresentam as restrições das variáveis declaradas. Já a Figura §5.3 (b) apresenta um exemplo da aplicação de um axioma.

O esquema também é uma representação axiomática, porém com a definição de um identificador. Assim, como nos axiomas, pode-se agrupar as declarações e restrições dentro de um esquema. Isto é interessante, ao

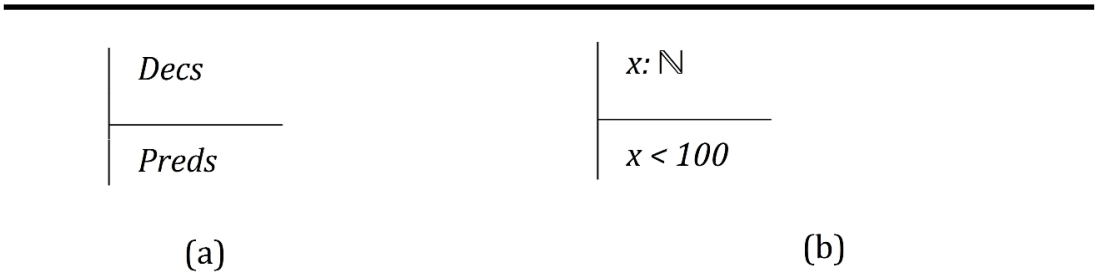


Figura 5.3: *Representação de axiomas*

passo que um esquema declarado pode ser reutilizado durante a especificação invocando-o por meio do identificador a ele atribuído. Assim, todas as declarações e restrições não precisam ser inseridas de forma repetida. Neste sentido, a utilização de esquemas, proporciona considerável clareza e economia de espaço e tempo, principalmente em especificações mais complexas.

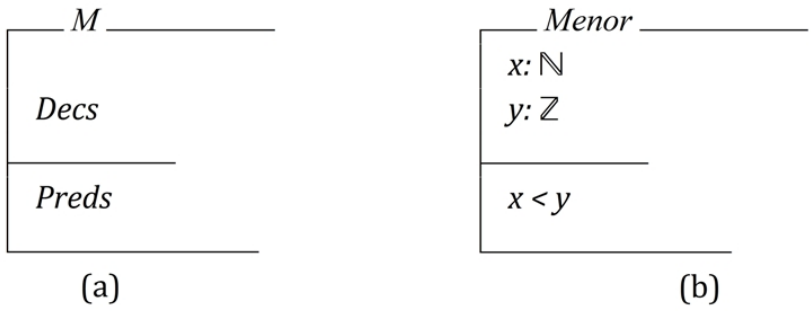


Figura 5.4: *Representação de esquemas*

A Figura §5.4 apresenta a estrutura básica de um esquema. A Figura §5.4 (a) introduz uma visão geral do esquema composto da parte declarativa (parte superior) e da parte predicativa (parte inferior). A Figura §5.4 (b) apresenta um exemplo da aplicação de um esquema com a sua identificação.

5.5 Provas

Além da importância da especificação formal para o desenvolvimento de sistemas e para a base formal de uma linguagem de modelagem, exis-

tem técnicas de prova matemática que tornam a especificação ainda mais interessante [83].

A prova matemática, segundo Moura [83], pode ser encarada como convencimento de alguns interlocutores. Uma prova é considerada boa quando convence o interlocutor daquilo que se quer demonstrar.

Outro fator importante citado pelo mesmo autor, refere-se ao rigor matemático a ser empregado na prova, pois ele pode ser aplicado em maior ou menor grau. No entanto, o ideal segundo Moura [83], é que a prova aplique o rigor matemático mínimo a partir do qual não reste dúvida sobre a sua validade.

Esta abordagem também é válida para uma especificação com a Notação Z. A prova pode ser aplicada em menor ou maior rigor dependendo das propriedades demostradas e do nível de detalhes em que estas são conduzidas [83].

O mesmo autor considera, no entanto, que na maioria das vezes apenas uma indicação envolvendo o argumento da prova é suficiente. Em outros casos, o próprio rigor empregado na especificação transmite a confiança necessária, concretizando, assim, a sua solidez. Assim, raramente é necessário providenciar uma prova através de argumentos explícitos [83].

A construção de provas a partir de uma especificação tem basicamente duas abordagens. A primeira está relacionada a prova usualmente realizada em textos matemáticos, ou seja, através do raciocínio sobre os objetos abstratos denotados pela especificação utilizando suas propriedades conhecidas. A segunda abordagem é realizada por meio da utilização de uma série de leis que possibilitam a manipulação mecânica da especificação. Com ela é possível manipular a forma simbólica dos objetos.

Ambientes computacionais apenas são capazes de trabalhar com a segunda abordagem, pois conseguem tratar apenas símbolos e não ideias. Estes ambientes são muito úteis pois proporcionam provas de teoremas ou de propriedades dentro de um escopo do sistema formal [83].

Como mencionado no capítulo §4.3.1 a ferramenta Z-EVES proporciona um ambiente computacional completo para edição das especificações e possibilita a geração dos principais teoremas para a validação do modelo.

5.6 Resumo do Capítulo

Com uma estrutura baseada na simplicidade da teoria dos conjuntos e na lógica de primeira ordem, a Notação Z apresenta-se como uma das principais

linguagens formais e pode ser utilizada, tanto para formalizar sistemas de informação quanto para formalizar linguagens. Além disso, a Notação Z herda as principais características da programação orientada a objeto, o que lhe atribui ainda mais recursos. Além disso, a ferramenta Z/EVES proporciona um ambiente para edição de especificações e fornece recursos para validação da especificação de forma automatizada.

Capítulo 6

Formalização

Os números são as regras dos seres
e a Matemática é o Regulamento do Mundo.

Francisco G. Teixeira, Matemático português (1869/1874)

Este capítulo apresenta a especificação formal da sintaxe abstrata da Linguagem de Domínio Específico da tecnologia Guaraná por meio da Notação Z. Na Seção §6.1 é apresentada uma introdução ao trabalho de especificação formal desenvolvido. Na Seção §6.2 são definidos os tipos básicos. Na Seção §6.3 são definidos os esquemas correspondentes as restrições OCL. Por fim, na Seção §6.4 é apresentado o resumo do capítulo.

6.1 Introdução

A formalização de uma linguagem de domínio específico é fundamental para estabelecer conceitos claros e concisos para cada propriedade da linguagem. Neste sentido, a formalização da linguagem de domínio específico da tecnologia Guaraná, teve como base as propriedades definidas no metamodelo UML, conforme a Figura §6.1, com restrições definidas em OCL.

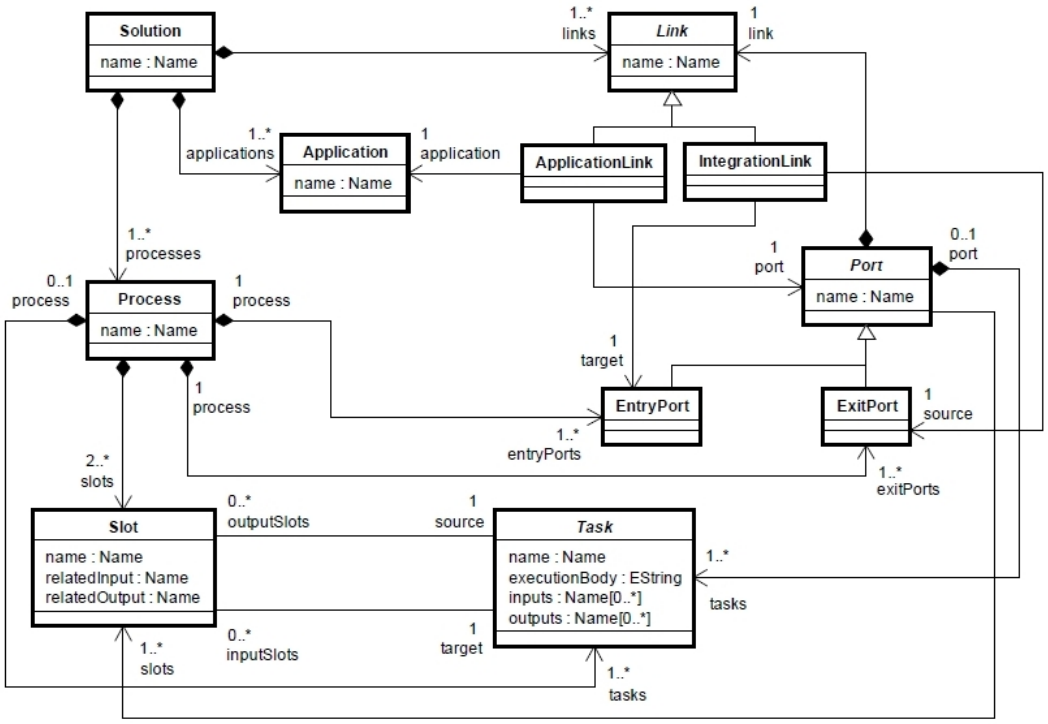


Figura 6.1: *Metamodelo UML (de Frantz e outros [36])*

Cada uma das propriedades definidas no metamodelo UML da tecnologia Guaraná, teve sua respectiva definição com a Notação Z. Algumas propriedades, no entanto, tiveram que ser adaptadas ao contexto do método formal, obedecendo conceitos rigorosos relacionados a operações sobre conjuntos e referência de tipos. Porém, procurou-se manter ao máximo a estruturação original fornecida para as propriedades da linguagem.

Para o processo de especificação formal e validação, foi utilizada a ferramenta Z/EVES. Esta definição está baseada na análise das ferramentas com suporte a Notação Z feitas por Dwivedi e Rath [29]. Conforme os autores o Z/EVES é uma ferramenta completa e interativa utilizada para edição, análise, verificação de tipo, cálculo de pré-condição, provas de refinamento e prova de teoremas. Além disso, a ferramenta é fácil de utilizar por meio de um editor gráfico.

Além do Z/EVES os autores consideram outras ferramentas disponíveis. A ferramenta HOL-Z é utilizada principalmente para fazer a prova de teoremas. Com ela é possível importar especificação em Z escritas em látex para a verificação de tipo e teoremas. A ferramenta Cadiz é utilizada para verificação de especificações e composição de tipos para Notação Z. As ferramentas Z type Checker (ZTC) e fuzz também fornecem suporte a verificação de tipos de especificações para a notação Z. Por fim a ferramenta RoZ é utilizada para gerar automaticamente os esquemas em Z correspondentes a um diagrama de classe UML.

Neste sentido, com base no metamodelo UML e utilizando a ferramenta Z/EVES, foi realizada especificação formal da sintaxe abstrata do linguagem de domínio específico da tecnologia Guaraná a partir da definição de tipos básicos e esquemas. A especificação formal pode ser encontrada no endereço eletrônico: http://www.gca.unijui.edu.br/mklein/Resources.

6.2 Tipos Básicos

Em uma solução de integração projetada com a tecnologia Guaraná, vários blocos de construção são utilizados. A especificação formal da sua sintaxe abstrata inicia com a definição e representação dos seus tipos básicos, que são utilizados para especificação dos conjuntos maiores.

Neste sentido foi definido o tipo básico *Char*, cf. Figura §6.2 para representar o conjunto de todos os caracteres formado pelos elementos [a..z][A..Z].

Além do tipo básico *Char* tem-se ainda um tipo *Text* que é formado por uma sequência de caracteres conforme representado na Figura §6.3. Este tipo é utilizado para representar os *Scripts* que implementam a lógica de negócios das tarefas que compõe uma solução.

Com a definição dos tipos básicos tem-se suporte para avançar na especificação visando estruturar as demais declarações e restrições.

$$[Char]$$

Figura 6.2: *Tipo básico para o conjunto de caracteres*

$$Text == \text{seq}\ Char$$

Figura 6.3: *Definição de Text*

Como pode ser observado no metamodelo UML da tecnologia Guaraná, apresentada na Seção §2.4, todos os blocos construtores são identificados através de um nome, que deve ser exclusivo em seu conjunto. Neste sentido, inicialmente definiu-se as restrições para a composição de *Name*.

$$Name == Char \times \mathbb{P}\ (\text{seq}\ Char \times \text{seq}\ (0\ ..\ 9))$$

Figura 6.4: *Definição de Name*

Conforme apresentado na Figura §6.4 um *Name* é formado por um par ordenado, onde o primeiro elemento é obrigatoriamente um *Char* e o segundo elemento é formado pelo conjunto potência formado pela relação entre uma sequência de caracteres e uma sequência de dígitos formada pelos números inteiros de 0 a 9. Como exemplos bem formados para o tipo *Name*, tem-se: n123abc, nabc123, n123, nabc. Por outro lado, elementos como 1abc e 12345, devem ser rejeitados, pois não contemplam as devidas restrições.

Dadas as restrições para o tipo *Name*, o qual identifica todas as instâncias dos blocos de construção em uma solução de integração, definiu-se alguns subconjuntos que são compostos por elementos deste tipo, conforme Figura §6.5.

Como a identificação é realizada por meio de um nome que pertence a um determinado subconjunto de nomes atribuídos à um bloco de construção,

$$Tasks\_Names, Processes\_Names, Slots\_Names, Applications\_Names, Ports\_Names, Links\_Names, Solutions\_Names, Gateway\_Names: \mathbb{P}\ Name$$

Figura 6.5: *Declaração dos elementos do tipo Name*

deve-se também, na especificação formal, separar estes subconjuntos ao nível em que a unicidade é exigida. Assim, tem-se a restrição de nome único para cada conjunto, ou seja, no conjunto de *Tasks_Names* não podem existir dois nomes iguais. Já em *Processes_Names*, por exemplo, pode existir um nome igual a um nome em *Tasks_Names*, sem prejuízo de identificação.

$$Names == \{Tasks\_Names, Processes\_Names, Slots\_Names, Applications\_Names, Ports\_Names, Links\_Names, Solutions\_Names, Gateway\_Names\}$$

Figura 6.6: *Conjunto de nomes da Solução.*

A Figura §6.6 representa o conjunto *Names*. Este conjunto é formado por todos os subconjuntos de nomes. Com isso, foram definidos os tipos básicos com as propriedades mas elementares desta especificação.

6.3 Esquemas

Na Notação Z os esquemas são utilizados para agrupar determinadas especificações. Esta estrutura possibilita que estas especificações possam ser utilizadas no decorrer da especificação com facilidade. Um esquema é composto por duas partes, sendo a parte declarativa utilizada para fazer todas as declarações que se queira vincular e a parte predicativa com o conjunto de todas as restrições referentes ao conjunto de declarações.

6.3.1 Gateway

A relação entre *Task* e *Slot* ocorre a partir de seus pontos de ligação. A representação no metamodelo atribui propriedades aos dois lados da relação

que possibilita que os pontos de relação sejam identificados. Assim, *input* e *output* são propriedades de *Slot* e são do tipo *Task*. Já *relatedInput* e *relatedOutput* são propriedades de *Task* e são do tipo *Slot*.

Não é possível fazer esta representação na Notação Z de forma direta. Como a Notação Z está baseada na teoria dos conjuntos, esta representação seria equivalente a ter dois conjuntos A e B, por exemplo, onde A está contido em B e B está contido em A, sendo, no entanto, A e B conjuntos distintos. O que não é possível.

Assim, para representar esta relação foi utilizado um esquema definido como *Gateway*, apenas com a propriedade nome para identificação, conforme Figura §6.7

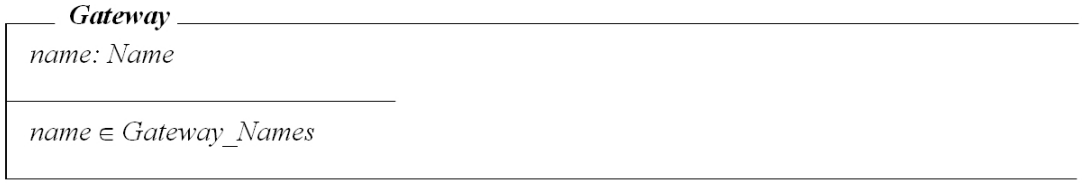


Figura 6.7: *Esquema Gateway*

Em seguida foram definidos os quatro conjuntos do tipo *Gateway*, conforme apresentado na Figura §6.8.

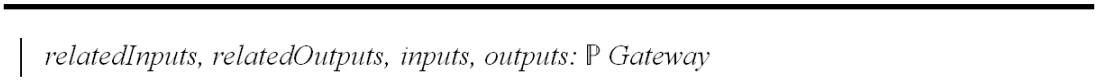


Figura 6.8: *Definição das variáveis de relação Task e Slot*

Assim, *Gateway* passa a ser um novo esquema. A relação entre tarefas e *slots* não consta de forma explicita no metamodelo, mas está expressa de forma implícita e pode ser representada desta forma.

6.3.2 Task

As tarefas são compostas por um conjunto de entradas, um conjunto de saídas, além de um *executionBody* que é um trecho de código java que im-

plementa as atividades que devem ser realizadas por aquela tarefa. As entradas e saídas são conectadas aos *slots* em tempo de execução e contêm mensagens. No *executionBody*, um engenheiro de software tem acesso às mensagens que trafegam nas entradas e saídas.

```
    context Task
(1)   inv: inputs->union(outputs)->isUnique(n: Name | n)
(2)   inv: inputSlots->collect(s: Slot | s.relatedInput) = inputs
(3)   inv: outputSlots->collect(s: Slot | s.relatedOutput) = outputs
```

Figura 6.9: *Restrições OCL para Task (de Frantz e outros [36])*

Conforme invariantes apresentadas na Figura §6.9 as entradas e saídas de uma tarefa devem ter nomes exclusivos e devem obrigatoriamente estar conectados a um slot. Além disso, jamais uma entrada ou saída poderá estar ligada a mais de um slot.

O esquema *Task* apresentado na Figura §6.10 trata destas restrições. A parte declarativa apresenta *name* do tipo *Name, executoinBody* do tipo *Text, taskGateway, inputs e outputs* do tipo *Gateway, inputSlots* e *outputSlots*. Estes dois últimos são utilizados para definir a relação entre as entradas e saídas de tarefas com entradas e saídas de Slot, sendo para tanto definidos como uma função bijetora.

A linha (1) indica que para cada instância de *Task*, o respectivo nome deve pertencer ao conjunto de *task_names*.

Na linha (2) e (3) está definido nome único para o conjunto de *inputs* e *outputs*, relativo a linha (1) das restrição OCL da Figura §6.9. Primeiramente declara-se uma variante adicional para agrupar os dois conjuntos, e em seguida define-se que para todo *tg1* e *tg2* do tipo *taskGateway*, *tg1.name* é igual a *tg2.name* se e somente se *tg1* for igual a *tg2*. Com isso, garante-se nome único dentro do conjunto de nomes de entradas e saídas de tarefa.

As linhas (4) e (5) definem as restrições de inputSlots como sendo uma função bijetora. Com isso, tem-se que seus elementos sejam formados pela relação entre *relatedInputs* e *inputs*, onde, para cada elemento existente no conjunto *relatedInput* deve, obrigatoriamente, existir um elemento no conjunto *inputs*. E para cada elemento no conjunto *inputs* deve existir um e apenas um elemento no conjunto *relatedInputs*.

Task

$name: Name$
$executionBody: Text$
$taskGateway: \mathbb{P}\ Gateway$
$inputs, outputs: \mathbb{P}\ Gateway$
$inputSlots: relatedInputs \rightarrowtail\!\!\!\!\!\!\rightarrow inputs$
$outputSlots: relatedOutputs \rightarrowtail\!\!\!\!\!\!\rightarrow outputs$

$name \in Tasks\_Names$ (1)
$taskGateway = inputs \cup outputs$ (2)
$\forall\ tg1, tg2: taskGateway \bullet\ tg1.name = tg2.name \Leftrightarrow tg1 = tg2$ (3)
$\forall\ ri: relatedInputs \bullet\ \exists\ i: inputs \bullet\ (ri, i) \in inputSlots$ (4)
$\forall\ i: inputs \bullet\ \exists\ ri: relatedInputs \bullet\ (ri, i) \in inputSlots$ (5)
$\forall\ ro: relatedOutputs \bullet\ \exists\ o: outputs \bullet\ (ro, o) \in outputSlots$ (6)
$\forall\ o: outputs \bullet\ \exists\ ro: relatedOutputs \bullet\ (ro, o) \in outputSlots$ (7)

Figura 6.10: *Esquema Task*

As linhas (6) e (7) atribuem a *outputSlots* a mesma condição de *inputSlots*, ou seja, para cada elemento do conjunto *relatedOutput* existe um elemento em *output*, e para cada elemento em *output* existe um elemento no conjunto *relatedOutput*.

As restrições apresentadas entre as linhas (4) e (7) são equivalentes as linhas (2) e (3) das restrições OCL, e garantem que cada entrada ou saída de tarefa esteja, obrigatoriamente, conectada a um e apenas um *Slot*.

6.3.3 Slot

Cada *slot* tem uma propriedade chamada *relatedInput* e uma propriedade chamada *relatedOutput*, as quais indicam a sua ligação com a entrada e com a saída da tarefa, respectivamente. Com isso, o conjunto de entradas de todas as tarefas deve ser igual ao conjunto de *relatedInput* de *inputSlots* e o conjunto de saídas de todas as tarefas deve ser igual ao conjunto de *relatedOutput* de *outputSlots*. Assim, temos a garantia de que cada entrada ou saída é ligada a um e apenas um slot.

```
        context Slot
(1)     inv: not (target = source)
(2)     inv: target.inputs->includes(relatedInput)
(3)     inv: source.outputs->includes(relatedOutput)
(4)     inv: let sourceProcess: Process = Process.allInstances()->any(p: Process |
                 p.tasks->union(p.entryPorts.tasks)->
                   union(p.exitPorts.tasks)->includes(self.source)) in
             let targetProcess: Process = Process.allInstances()->any(p: Process |
               p.tasks->union(p.entryPorts.tasks)->
                   union(p.exitPorts.tasks)->includes(self.target)) in
             sourceProcess = targetProcess
```

Figura 6.11: *Restrições OCL para Slot (de Frantz e outros [36])*

A Figura §6.11 apresenta as restrições para *slots*. Com elas, caracteriza-se que cada *slot* deve conectar duas tarefas diferentes, por meio das entradas e saídas correspondentes. Porém, os *slots* não podem conectar a tarefa de um processo com a tarefa de outro processo.

Para estabelecer as restrições para o conjunto de *Slots* que compõe uma solução de integração foram estabelecidos as variantes conforme Figura §6.12: *name, slots* e *interslots* que relacionam duas tarefas (*Task*), *source* e *target* do tipo *Task* e por fim, *slotGateway, relatedInputs, relatedOutputs* como sendo do tipo *Gateway*.

As restrições definidas no esquema *Slot* estabelecem que *name* deve pertencer ao conjunto de *Slot_Names*, conforme a linha (1).

As linhas (2) e (3) definem nome único para as entradas e saídas de *Slot*, ou seja, *reladetInput* e *relatedOutput* forma agrupados no conjunto *slotGateway*, que para toda instância *sg1* e *sg2* do tipo *slotGateway*, *sg1.name* é igual a *sg2.name* se e somente se *sg1* é igual a *sg2*.

Na linha (4) está definido que *target* não pode ser igual a *source*, ou seja, a entrada de um *Slot* não pode ser ao mesmo tempo a saída deste *Slot* conforme apresentado na linha (1) das restrições OCL.

As linhas (5) e (6) da especificação se referem as linhas (2) e (3) das restrições OCL da Figura §6.11. Com elas fica definido que para toda instância de

Slot

$name: Name$

$slots, interslots: Task \leftrightarrow Task$

$source, target: Task$

$slotGateway, relatedInputs, relatedOutputs: \mathbb{P}\ Gateway$

$name \in Slots\_Names$ (1)

$slotGateway = relatedInputs \cup relatedOutputs$ (2)

$\forall\ sg1, sg2: slotGateway \bullet\ sg1.name = sg2.name \Leftrightarrow sg1 = sg2$ (3)

$\neg\ target = source$ (4)

$\forall\ ri: relatedInputs \bullet\ ri \in target.inputs$ (5)

$\forall\ ro: relatedOutputs \bullet\ ro \in source.outputs$ (6)

Figura 6.12: *Esquema Slot*

relatedInputs ele deve, obrigatoriamente, pertencer ao conjunto *inputs* de tarefas definidas como *target*. Do mesmo modo, toda instância de *relatedOutputs* deve, necessariamente, pertencer ao conjunto *outputs* de tarefas definidas como *source*.

A restrição OCL definida na linha (4) que estabelece a impossibilidade de um *Slot* conectar duas tarefas em dois processos diferentes será especificada no esquema *Process*. Isto acontece, pois a restrição não ocorre totalmente dentro do conjunto *Slot*, mas dentro de um processo.

6.3.4 Port

Portas são compostas de tarefas e *slots* e são conectadas entre si por meio de *links*. Os *links* podem ser do tipo *ApplicationLinks*, caso façam a ligação entre Aplicações e portas; ou *IntegrationLinks*, caso façam a ligação entre uma porta de entrada de um processo e uma porta de saída de outro processo.

Além disso, as portas devem satisfazer as restrições especificadas na Figura §6.13, que determinam que as tarefas de uma porta devem ter nomes exclusivos. A porta de entrada deve ter um Comunicador do tipo *InCommunicator*, que é utilizado para ler as mensagens de um componente de

```
context Port
    inv: tasks->isUnique(name)

context EntryPort
    inv: tasks->one(oclIsKindOf(Communicator))
    inv: tasks->one(oclIsKindOf(InCommunicator))

context ExitPort
    inv: tasks->one(oclIsKindOf(Communicator))
    inv: tasks->one(oclIsKindOf(OutCommunicator))
```

Figura 6.13: *Restrições OCL para Port (de Frantz e outros [36])*

ligação; Já uma porta de saída também deve ter um Comunicador, porém do tipo *OutCommunicator*, o qual é utilizado para escrever mensagens para um componente de ligação.

A Figura §6.14 determina que o conjunto de *slots* de uma porta é formado pela união de seus *inputSlots* e *outputSlots*.

```
context Port::slots: Set(Slot)
    derive: tasks->collect(outputSlots)->union(tasks->collect(inputSlots))
```

Figura 6.14: *Restrições OCL para a relação Port e Slot (de Frantz e outros [36])*

As restrições para *links*, no entanto, não são especificadas dentro do esquema *Port*, mas dentro do esquema *Solution*. Isso ocorre porque o conjunto de *links* está contido dentro do conjunto *Solution*, sendo assim, não é possível ter uma "visão" completa a partir de esquema *Port*.

A especificação formal do conjunto de portas ocorre a partir de uma generalização de *Port* no contexto geral para *EntryPort* e *ExitPort* em um contexto mais específico. O esquema *Port*, cf. Figura §6.15 possui um *name* do tipo *Name*; um conjunto de *Task* e um conjunto de *slots* formado por *outputSlots* e *inputSlots*.

A linha (1) determina que o nome da porta deve pertencer ao conjunto *Ports_Names*.

Port

$name: Name$
$tasks: \mathbb{P}\ Task$
$slots, outputSlots, inputSlots: \mathbb{P}\ Slot$

$name \in Ports\_Names$ (1)
$slots = inputSlots \cup outputSlots$ (2)
$\forall\ s1, s2: slots \bullet\ s1.name = s2.name \Leftrightarrow s1 = s2$ (3)
$\forall\ t1, t2: tasks \bullet\ t1.name = t2.name \Leftrightarrow t1 = t2$ (4)
$\#\ tasks \geqslant 1$ (5)

Figura 6.15: *Esquema Port*

As linhas (2) e (3) estabelecem as restrições de *slots* dentro do esquema *Port*. Através delas é realizada a união dos conjuntos de *inputSlots* e *outputSlots* e atribuído a *slots* e posteriormente lhe é atribuída a condição de nome exclusivo.

As tarefas que compõe o conjunto de tarefas das portas também seguem o critério de nome exclusivo. Isto está definido na linha (4). E, por fim, atribui-se a cardinalidade para *tasks* conforme a linha (5), sendo que uma porta deve ter uma ou mais tarefas.

Uma porta de entrada (*EntryPort*) e uma porta de saída (*ExitPort*) herdam todas as características da porta (*Port*) e possuem características adicionais.

EntryPort

$port: Port$
$comunicator, inComunicator: \mathbb{P}\ Task$

Figura 6.16: *Esquema EntryPort*

Uma *EntryPort* possui tarefas específicas definidas como *comunicator* e

inComunicator, cf. Figura §6.16.

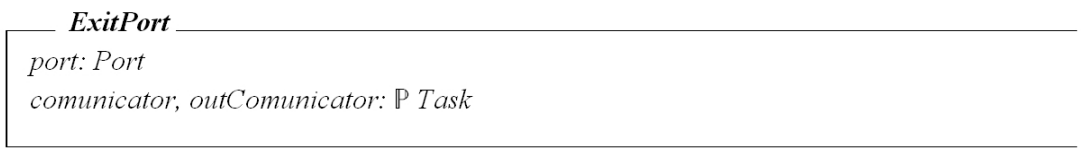


Figura 6.17: *Esquema ExitPort*

A *ExitPort* também possui tarefas específicas definidas como *comunicator* e *OutComunicator*, cf. Figura §6.17.

6.3.5 Application

Uma Aplicação (Application) não possui definições específicas de restrições OCL. No entanto, uma restrição OCL está definida em *Solution*, conforme Figura §6.22, onde lhe é atribuído um nome único para cada instância de *Application*, ou seja, todas as aplicações que fazem parte de uma solução são identificadas com um nome exclusivo.

Assim, a especificação formal para uma aplicação ocorre por meio da definição do esquema *Application* com a atribuição apenas de *name* do tipo *Name*, o qual deve, necessariamente, pertencer ao conjunto de *Application_Names*, cf. Figura §6.18.

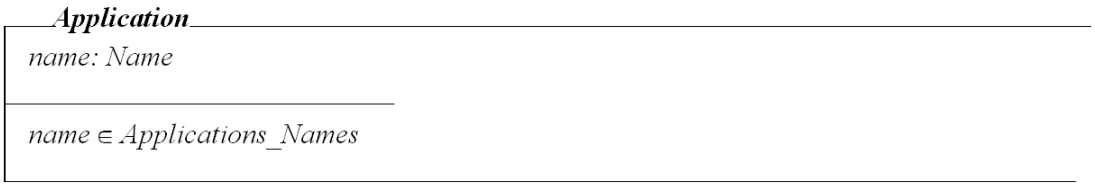


Figura 6.18: *Esquema Aplication*

6.3.6 Link

Do mesmo modo que uma aplicação, um link também não é definido a partir do seu contexto. As restrições OCL para link, estão dispostas no contexto de porta. A especificação formal das restrições para *applicationLink* e *integratioLink*, no entanto, são definidas dentro de esquema *Solution*.

Figura 6.19: *Esquema Link*

Com isso, no esquema *Link* apenas é definido *name* do tipo *Name* e lhe é atribuída a restrição de que este deve pertencer ao conjunto de *Link_Names*. Além disso, o esquema *Link* é composto por *integrationLink* e *applicationLink*. O primeiro faz a relação entre uma porta de entrada e uma porta de saída de um processo, ao passo que *applicationLink* conecta uma aplicação a um processo através de uma porta.

6.3.7 Process

A classe Processo contém a lógica de integração necessária para interagir com as aplicações dentro do ecossistema de software e para processar os dados que trafegam pela solução de integração. Um processo é composto por, pelo menos, uma porta de entrada, uma porta de saída, uma tarefa, e pelo menos dois slots. Na Figura §6.20 estão representadas as restrições impostas à um processo.

De acordo com as restrições, *tasks*, *slots* e *ports* devem ter nomes exclusivos. Um processo não pode ter tarefas do tipo *Communicator*, pois ela é uma tarefa específica para as portas. Além disso, deve haver apenas um *slot* do tipo interslot por porta.

```
        context Process
(1)       inv: tasks->union(entryPorts.tasks->union(exitPorts.tasks))->isUnique(name)
(2)       inv: slots->isUnique(name)
(3)       inv: entryPorts->union(exitPorts)->isUnique(name)
(4)       inv: tasks->select(oclIsKindOf(Communicator))->size() = 0
(5)       inv: let interslots: Set(Slot) = slots->select(s: Slot |
                  not self.tasks->includes(s.source) and self.tasks->includes(s.target) or
                  self.tasks->includes(s.source) and not self.tasks->includes(s.target)) in
             interslots->size() = self.entryPorts->size() + self.exitPorts->size()
```

Figura 6.20: *Restrições OCL para Process (de Frantz e outros [36])*

Em um processo existem tarefas próprias e tarefas das portas que compõe o processo, sendo que todas elas devem ter nomes diferentes. Assim, o conjunto de tarefas que compõe um processo é dado pela união das tarefas próprias com o conjunto de tarefas das portas de entrada e saída.

Em relação aos *slots*, é apenas lhe atribuído a restrição de nome único, uma vez que a composição do conjunto total de *slots* apenas é possível através análise das suas propriedades: *target* e *source*. Já os *slots* do tipo *interslot* são identificados por conectar uma tarefa própria do processo à uma tarefa de uma porta. Com isso, tem-se exatamente, um *interslot* por porta.

O esquema *Process* é mais abrangente que os demais, já que é nesse bloco construtor que está a lógica de integração. *O esquema Process* é composto por: *name* do tipo *Name*; *selTask* do tipo *Task* para representar as tarefas próprias de *Process*; *entryPortTasks* e *exitPortTasks* do tipo *Task* para representar as tarefas das portas de entrada e das portas de saída de *Process* respectivamente, além de uma variante adicional para agrupar todas as tarefas que pertençam a *Process*.

Além disso, ainda existem *entryPorts*, *exitPorts* e *ports* do tipo *Port* para representar o conjunto de portas de entrada e saída de *Process*. Por fim, duas variantes do tipo *Slot* denominadas de *slots* e *interslots*.

A restrição da linha (1) define que todo *name* deve pertencer ao conjunto de *Processes_Names*.

As restrições das linhas (2), (3) e (4) referem-se as tarefas de *Process*. A linha (2) é relativa a primeira parte da linha (1) das restrições OCL e atribui à variante *tasks* a união dos conjuntos de tarefas próprias do processo com as

Process

$name: Name$
$selfTasks, entryPortTasks, exitPortTasks, tasks: \mathbb{P}\ Task$
$entryPorts, exitPorts, ports: \mathbb{P}\ Port$
$slots, interslots: \mathbb{P}\ Slot$

$name \in Processes\_Names$ (1)
$tasks = selfTasks \cup entryPortTasks \cup exitPortTasks$ (2)
$selfTasks \cap (entryPortTasks \cup exitPortTasks) = \emptyset$ (3)
$\forall\ t1, t2: tasks \bullet\ t1.name = t2.name \Leftrightarrow t1 = t2$ (4)
$interslots \subseteq slots$ (5)
$\forall\ s1, s2: slots \bullet\ s1.name = s2.name \Leftrightarrow s1 = s2$ (6)
$ports = entryPorts \cup exitPorts \wedge entryPorts \cap exitPorts = \emptyset$ (7)
$\forall\ p1, p2: Port \bullet\ p1.name = p2.name \Leftrightarrow p1 = p2$ (8)
$\forall\ s: slots; t1, t2: Task$ (9)
$\bullet\ (t1, t2) \in s.slots$
$\Rightarrow t1 \in selfTasks \wedge t2 \in selfTasks \wedge t1 \neq t2 \vee (t1, t2) \in s.interslots$
$\Rightarrow t1 \in selfTasks \wedge t2 \in entryPortTasks \cup exitPortTasks \wedge t1 \neq t2$
$\#\ interslots = \#\ entryPorts + \#\ exitPorts$ (10)
$\#\ entryPorts \geqslant 1$ (11)
$\#\ exitPorts \geqslant 1$ (12)
$\#\ selfTasks \geqslant 1$ (13)
$\#\ slots \geqslant 2$ (14)

Figura 6.21: *Esquema Process*

tarefas das portas, ou seja, *tasks* é formado por *selfTasks* união *entryPortTasks* união *exitPortTasks*. A linha (3) refere-se a restrição OCL da linha (4), que determina que a interseção entre o conjunto de tarefas próprias do processo (*selfTasks*) com as tarefas das portas de entrada (*entryPortTasks*) e saída (*exitPortTasks*) forma um conjunto vazio. E a linha (4) define nome exclusivo para todas as tarefas referente a segunda parte da linha (1) das restrições OCL. Assim, para todo t1 e t2 do tipo *Tasks*, tenho que, *t1.name* é igual a *t2.name* se e somente se *t1* é igual a *t2*.

As linhas (5) e (6) definem que o conjunto de *interslots* deve estar contido

no conjunto *slots* e sua identificação através de *name* exige que ele seja exclusivo, conforme linha (2) das restrições OCL. Com isso, tem-se que para todo *s1* e *s2* do tipo *slots*, *s1.name* é igual a *s2.name* se e somente se *s1* for igual a *s2*.

Na linha (7) o predicado atribui ao conjunto de portas todas as instâncias de portas de entrada e portas de saída. Além disso, define-se que a intersecção entre estes dois conjuntos deve formar um conjunto vazio. Em seguida, na linha (8) é definido nome exclusivo para todo o conjunto de portas conforme a restrição OCL da linha (3).

A linha (9) refere-se as restrições de *slots* apresentadas em OCL conforme a Figura §6.20 na linha (5). Esta restrição define as propriedades para slots e interslots, sendo que ambos fazem a relação entre duas tarefas. Assim, para todo *s* do tipo *slots*, *t1* e *t2* do tipo *Task*, tem-se que para a relação (*t1, t2*) pertencer ao conjunto de *slots* implica que *t1* e *t2* devem, obrigatoriamente, pertencer ao conjunto de tarefas próprias do processo. E para que a relação (*t1, t2*) pertença ao conjunto de *interslots* implica que *t1* deve pertencer ao conjunto de tarefas próprias do processo e *t2* deve pertencer ao conjunto formado pela união das tarefas das portas de entrada com as tarefas das portas de saída.

Através da restrição para *interslots* fica estabelecido que este relaciona a tarefa do processo com a tarefa de uma porta. Com isso, tem-se exatamente um *interslot* para cada porta. Portanto, pode-se definir a cardinalidade de *interslots*, ou seja, o número de *interslots* que pertencem a um processo é formado pela soma do número de portas de entrada com o número de portas de saída conforme apresentado na linha (10).

As restrições entre as linhas (11) e (14) definem a cardinalidade para os blocos construtores que compõe um processo. Em um processo deve existir ao menos uma porta de entrada (entryPorts), ao menos uma porta de saída (exitPorts), as menos uma tarefa própria do processo (selfTask) e ao menos dois slots.

6.3.8 Solution

Conforme o metamodelo, a Solução é a classe raiz, e representa uma solução de integração. Possui uma propriedade nome para a sua identificação e é composta por um ou mais processos, uma ou mais aplicações e um ou mais links. A solução deve satisfazer as invariantes apresentadas na Figura §6.22.

As restrições condicionam que os nomes das aplicações, processos e links devem ser únicos.

```
context Solution
(1)  inv: applications->isUnique(name)
(2)  inv: processes->isUnique(name)
(3)  inv: links->isUnique(name)
```

Figura 6.22: *Restrições OCL para Solution (de Frantz e outros [36])*

Além das restrições OCL estabelecidas da perspectiva de Solution, são especificadas no esquema Solution, as restrições para *integrationLink* e *applicationLink* estabelecidas a partir da perspectiva de Port. Por esse motivo as restrições foram inseridas nesse ponto do texto.

Conforme a Figura §6.23, uma porta pode estar ligada com uma aplicação através de um *aplicationLink*, ou a um processo através de um *integrationLink*.

```
context Port::link: Link derive:
  let appLink: Link = ApplicationLink.allInstances()->any(port = self) in
  let intLink: Link = IntegrationLink.allInstances()->any(source = self or target = self) in
    if not appLink.oclIsUndefined() then appLink else intLink endif
```

Figura 6.23: *Restrições OCL para a relação Port e Link (de Frantz e outros [36])*

Por fim, a restrição apresentada na Figura §6.24 determina que um *integrationlink* jamais pode ligar uma porta de saída de um processo com uma porta de entrada do mesmo processo.

```
context IntegrationLink:
  inv: not (source.process = target.process)
```

Figura 6.24: *Restrições OCL para link (de Frantz e outros [36])*

Conforme descrito acima, *Solution* é a Classe principal e representa uma solução de integração como um todo. No esquema *Solution* tem-se a identifi-

cação através de *name* do tipo *Name*; *applications* do tipo *Applications*, *processes* do tipo *Process* e *integrationLinks, applicationLinks, links* do tipo *links*.

Solution

name: Name
applications: $\mathbb{P}$ *Application*
processes: $\mathbb{P}$ *Process*
integrationLinks, applicationLinks, links: $\mathbb{P}$ *Link*

$name \in Solutions\_Names$ (1)
$links = integrationLinks \cup applicationLinks$ (2)
$\forall\ a1, a2: applications \bullet\ a1.name = a2.name \Leftrightarrow a1 = a2$ (3)
$\forall\ p1, p2: processes \bullet\ p1.name = p2.name \Leftrightarrow p1 = p2$ (4)
$\forall\ l1, l2: links \bullet\ l1.name = l2.name \Leftrightarrow l1 = l2$ (5)
$\forall\ l1: links; port, entryPort, exitPort: Port; p1, p2: processes;$ (6)
$application: applications$
$\bullet\ (entryPort, exitPort) \in l1.integrationLinks$
$\Rightarrow entryPort \in p1.entryPorts \wedge exitPort \in p2.exitPorts \wedge p1 \neq p2$
$\vee\ (port, application) \in l1.applicationLinks$
$\forall\ sourceProcess, targetProcess: processes; s: Slot$ (7)
$\bullet\ s.source \in sourceProcess.tasks \wedge s.target \in targetProcess.tasks$
$\#\ processes \geqslant 1$ (8)
$\#\ applications \geqslant 1$ (9)
$\#\ integrationLinks \geqslant 1$ (10)

Figura 6.25: *Esquema Solution*

A parte predicativa é composta, inicialmente, com a definição de *name* que deve pertencer ao conjunto de *Solution_Names*.

A linha (2) atribui para *links* a união entre os conjuntos de *integration-Links* e *applicationlinks* e lhe atribui nome exclusivo de acordo com a especificação da linha (3).

As linhas (4) e (5) definem a restrição de nome exclusivo para todos as instâncias de *Application* e *Process*, respectivamente.

A relação entre uma porta de entrada e uma porta de saída de um processo está expressa através do *integrationLink*. Já a relação entre uma porta de um processo e uma aplicação é expressa por um *applcationLink*. As restrições que especificam estas relações estão definidas na linha (6). Para toda instância *l1* de *links*; *port, entryPort, exitPort* do tipo *Port*; *p1* e *p2* do tipo *processes*; e *application* do tipo *applications*, tem-se que:

Para que a relação (*entryPort, exitPort*) pertença ao conjunto de *integrationLinks*, implica que a porta de entrada (*entryPort*) pertença ao conjunto de portas de entrada do processo *p1* e que a porta de saída (*exitPort*) pertença ao conjunto de portas de saída do processo *p2*. Além disso, *p1* deve ser diferente de *p2*, isto é, devem ser dois processos distintos. Já a relação (*port, application*) pertence ao conjunto de *applicationLinks*.

A restrição da linha (7) determina que, para toda instância *sourceProcess* e *targetProcess* do tipo *Process* e s do tipo *Slot*, tem-se que, *s.source* e *s.target* pertencem ao conjunto de tarefas de *Process*.

As restrições (8), (9) e (10) definem a cardinalidade para o esquema *Solution*. Com eles fica estabelecido que uma solução de integração deve ter, obrigatoriamente, um ou mais processos, uma ou mais aplicações e um ou mais *integrationLinks*.

6.4 Resumo do Capítulo

Este capítulo apresenta a especificação formal para a sintaxe abstrata do Guaraná DSL, com base nas definições do metamodelo UML com as restrições expressas em OCL. Com a especificação fornecida é possível fazer a verificação das propriedades através da prova de teoremas.

Capítulo 7

Validação

A ciência nunca resolve um problema
sem criar pelo menos outros dez.

George B. Shaw, Jornalista irlandês (1856-1950)

Por meio da especificação formal da sintaxe abstrata do Guaraná DSL, é possível obter maior conhecimento sobre a linguagem, diminuir ambiguidades no entendimento, além de favorecer a estruturação com maior facilidade de compreensão e visibilidade. Esta especificação, no entanto, precisa ser provada. Na Seção §7.1 são apresentados conceitos de provas de diferentes autores e como ela foi aplicada neste trabalho. Na Seção §7.2 é apresentada uma análise dos teoremas gerados pela ferramenta para a validação. A Seção §7.3 apresenta as etapas para a validação de uma especificação. Na Seção §7.4 é apresentada a validação do modelo apresentado. A Seção §7.5 apresenta um resumo do capítulo.

7.1 Prova Formal

De acordo com Moura [83] a motivação para provar a correção de um sistema é o desejo de entender melhor suas especificidades e fazê-las entendíveis a outros interlocutores. Deste modo, o autor destaca que as provas de correção podem ser produzidas no estilo de prova matemática, ou seja, através da utilização da linguagem natural com um certo nível de informalidade, fazendo com que seja legível e inteligível para outros leitores humanos.

Neste sentido, Moura [83] complementa que a própria especificação, através de uma linguagem formal, pode ser considerada uma prova para um sistema, desde que ela convença os seus interlocutores.

A especificação formal da sintaxe abstrata do Guaraná DSL apresentada neste trabalho, foi envolvida por uma descrição detalhada de suas propriedades. Com isso, pretende-se deixar claro que a especificação formal corresponde rigorosamente com as propriedades fornecidas pelo metamodelo e pelas restrições OCL.

No entanto, outros autores são críticos desta abordagem. Freitas [38] afirma que podem ocorrem ambiguidades na utilização da linguagem natural. Bertot e Castéran [10] consideram que o tamanho e a complexidade de uma prova completa torna imprescindível a utilização da verificação automatizada para garantir que todas as regras da especificação estão corretamente aplicadas. Complementa que é impraticável a prova de uma especificação de forma manual.

Neste sentido verifica-se que diversos autores e linhas de pensamento defendem provas verificadas por máquinas em seus padrões, princialmente em ambientes que lidam com sistemas críticos. Bowen e Hinchey [15] destaca que a Agência Espacial Europeia, por exemplo, defende a prova formal, e sugere que as provas sejam verificadas de forma automatizada com o objetivo de reduzir a possibilidade de erro humano.

Com o surgimento de ferramentas de apoio à linguagens formais, é possível realizar a prova de especificações de forma automatizada. Esta forma de validação é uma prática amplamente utilizada em engenharia de software. Os trabalhos apresentados por [37, 91, 106, 108], por exemplo, utilizam a prova automatizada por meio da ferramenta Z/EVES com o objetivo de provar suas especificações.

De acordo com Freitas [38], duas técnicas automatizadas de prova podem ser utilizadas: a prova de teoremas e a verificação de modelo. A provas de teoremas consiste na construção de um conjunto de teoremas a partir da especificação, capazes de verificar e validar todas as suas propriedades. Já a verificação de modelo consiste em verificar as propriedades por meio de uma análise exaustiva de todos os estados possíveis que um sistema poderia assumir durante a sua execução.

Considerando a especificação formal da sintaxe abstrata do Guaraná DSL, a prova por meio da verificação de modelo não é a mais apropriada. Isso ocorre pois a tecnologia Guaraná é uma linguagem de modelagem e não um modelo propriamente dito. Nesse sentido ela é utilizada para gerar diferentes modelos, cada qual com propriedades específicas. Assim, cada modelo gerado com o Guaraná tem um conjunto distinto de estados possíveis. Esses modelos poderiam ser submetidos a esse estilo de prova.

No entanto existem restrições mesmo em casos onde a aplicabilidade da verificação de modelo poderia ser aplicada. Baier e outros [9] consideram que a análise exaustiva de todos os estados possíveis de um sistema (modelo) é praticamente inviável e complementa que, na prática, dependendo do tamanho e da complexidade do sistema, apenas um pequeno subconjunto destes estados é tratado, deixando a verificação incompleta.

Bouajjani e outros [14] também argumentam que todos os algoritmos de verificação de modelo tem complexidade exponencial ao tamanho do sistema dificultando, assim, a sua prova completa se aplicados à sistemas grandes.

Já a prova de teoremas pode ser utilizada com bastante propriedade, validando as especificidades da linguagem. Moura [83] destaca que uma vez validadas através de teoremas, elas sempre serão válidas em todos os casos. Assim, todos modelos gerados com o Guaraná DSL sempre estarão sujeitados às regras especificadas e validadas por esse estilo de prova.

Freitas [38], no entanto, considera que, independente de ferramenta, a prova de teoremas sempre é difícil. Ledru [68] cita duas limitações importantes para a prova de teoremas: a curva de aprendizado para demonstrar teoremas e o desempenho quando inúmeros esquemas estão envolvidos pelo teorema.

7.2 Análise dos Teoremas

De acordo com Ledru [68], o Z/EVES fornece suporte para prova de teoremas de especificações com a notação Z. O autor também destaca que a

ferramenta pode ser utilizada no modo interativo ou automático sendo que o modo automático pode provar teoremas mais simples. Para provar teoremas mais complexos, é necessário a intervenção do desenvolvedor, através da inserção de comandos de prova. Contudo, teoremas simples nem sempre são sinônimo de teoremas curtos. Neste sentido, teoremas longos, porém simples, muitas vezes podem gerar provas longas que podem ser manipulados automaticamente.

Saaltink [98] também destaca que o Z/EVES é capaz de fazer a verificação de sintaxe e de tipo, além de gerar inúmeros teoremas de forma automática sobre especificação, e, com isso, permitir a prova de suas propriedades. Os teoremas são gerados automaticamente ou especificados pelo desenvolvedor.

A Figura §7.1 ilustra um exemplo do teorema de Saaltink. O rótulo definido para o teorema indica o tipo de prova. Um rótulo pode ser definido como: *grule, frule, rule* ou *axiom*. O parâmetro *ability* que precede o rótulo é usado para habilitar ou desabilitar o uso automático do teorema durante uma prova, caso o mesmo seja uma *grule, frule* ou *rule* [37]. Além disso, o teorema é identificado por um nome e possui um predicado com o comando da prova.

---

theorem *ability* [rótulo] theorem-name
Predicate

Figura 7.1: *Exemplo de teorema (de Saaltink [98])*

Cabe ressaltar que os teoremas são classificados de acordo com o rótulo que precede o nome do teorema. As definições para cada rótulo de teorema estão descritas em Freitas e Woodcock [37]:

Uma *assumption rule* é um teorema gerado automaticamente pela ferramenta para todas as definições axiomáticas e para cada um dos elementos que compõem um tipo livre. Este teorema é rotulado como *grule* e é usado para expressar informações sobre tipos de elementos de uma especificação.

Uma *forward rule* é um teorema que é etiquetado como *frule* e que possui o padrão $P \Rightarrow Q$. O Z/EVES gera automaticamente uma *frule* para cada esquema definido na especificação, onde Q é composto pela parte declarativa do esquema de nome P. Desta forma, durante uma prova os tipos dos componentes de um esquema são conhecidos mesmo que o esquema não tenha sido expandido. Sempre que P está no contexto da prova a *frule* $P \Rightarrow Q$ é disparada e o predicado Q é adicionado ao contexto da prova.

Os teoremas etiquetados como *rule* são uma regra de reescrita. Uma *rule* permite reescrever predicados transformando-os em predicados de primeira ordem, permitindo que sejam facilmente verificados. A Ferramenta Z/EVES gera várias *rules* automaticamente para uma especificação em que um predicado "A" é substituído por um predicado "B" em uma sentença lógica em que A é uma sub-fórmula.

Assim, o Z/EVES gera outra classe de teoremas etiquetados como *axiom*. Estes teoremas expressam propriedades e fatos de uma especificação. Além disso, os teoremas que estabelecem pré-condições e inicialização de estados do sistema também são expressos como axiomas, porém estes dois últimos não são gerados de forma automática.

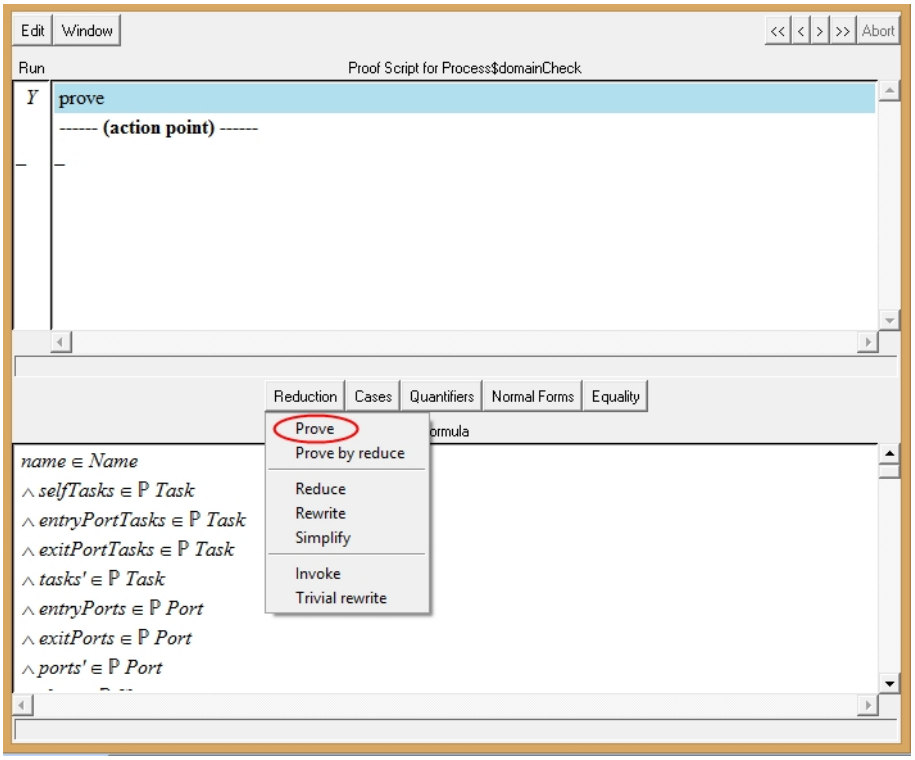


Figura 7.2: *Prova de teoremas rotulados como axiom.*

Para os teoremas etiquetados como *axiom*, é necessário a intervenção do desenvolvedor para concluir a prova. Para isto o Z/EVES proporciona várias maneiras de tratar uma prova de um teorema etiquetado com o rótulo *axiom*. Um menu com as opções está disponível na janela de provas da ferra-

menta, cf.Figura §7.2. Assim, dependendo do tipo e da complexidade da prova, o desenvolvedor pode utilizar uma ou mais opções específicas com o objetivo de validar a especificação.

7.3 Etapas para Validação

Para provar uma especificação, inicialmente o Z/EVES traduz automaticamente as especificações realizadas por meio da Notação Z em predicados em predicados de primeira ordem, tornando-os manipuláveis e de fácil entendimento. A partir desta tradução todas as manipulações lógicas podem ser realizadas. A Figura §7.3 mostra uma parte do código que representa a especificação da sintaxe abstrata do Guaraná DSL realizado na ferramenta Z/EVES.

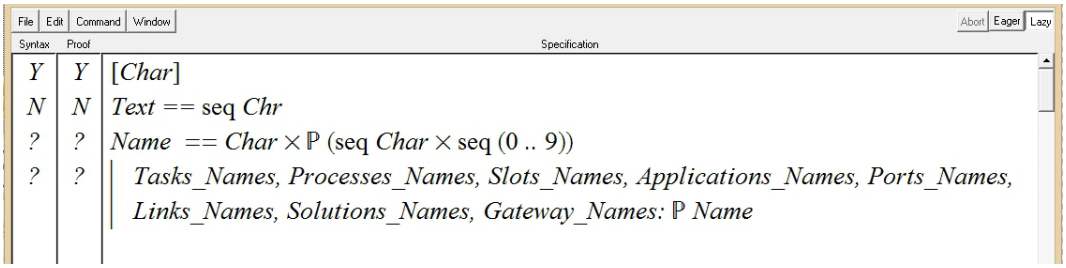


Figura 7.3: *Validação dos parágrafos da especificação.*

Verifica-se, na Figura 7.3, que o primeiro parágrafo é representado pela definição do tipo livre *Char* definido entre um par de *colchetes*. Esta é uma definição bem formada de acordo com a linguagem, ou seja, utiliza as regras de expressão da Notação Z. A letra "Y" nas colunas *Sintax* e *Proof* mostram isto. Já o segundo parágrafo não pode ser validado, pois ocorreu uma inconsistência de tipos, ou seja, o tipo *Chr* atribuído a *Text* não existe. Neste caso a ferramenta sinaliza com a letra "N" para *Sintax* e *Proof*. Os demais parágrafos estão sinalizados com "?", o que significa que ainda não foram verificados.

Woodcock e Davies [113] destacam várias etapas que são seguidas para provar uma especificação:

a) A verificação de sintaxe e de tipos

Ciancarini e outros [23] consideram que Z é uma notação muito bem estruturada e possui uma sintaxe e uma semântica complexa. Segundo os autores, este fator agrega um alto grau de dificuldade para prover a escrita

(desenvolvimento da especificação), a verificação e geração de documentos por meio da especificação Z. Por conta disto é comum que até mesmo usuários experientes cometam erros ao escrever uma especificação.

No entanto, com o surgimento das ferramentas de apoio aos métodos formais, este problema vem sendo contornado. O Z/EVES, além do suporte para edição de especificações que podem servir de documentação, oferece, também, suporte à verificação de sintaxe e de tipos.

De acordo com o manual de referência do Z/EVES [99], a verificação da sintaxe e de tipos é realizada de forma automática pela ferramenta, logo após a leitura do parágrafo. Com isso é garantido que os tipos dos operandos e operadores estão em concordância e que a especificação pertence a linguagem gerada pela gramática aceita pelo ferramenta.

Na Figura §7.4 são apresentados três exemplos de erros de sintaxe e de tipos.

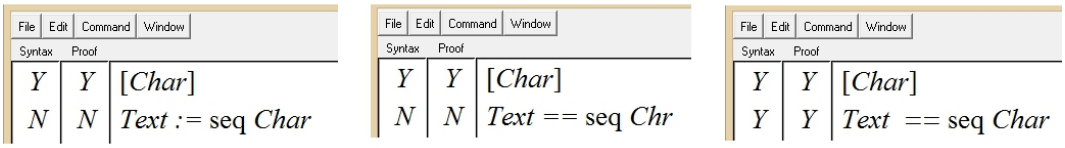


Figura 7.4: *Verificação de tipo e de sintaxe.*

A Figura §7.4 (a) mostra um erro de sintaxe. A atribuição de igualdade escrita através de " := " não é aceita pelo provador, o qual faz atribuição de igualdade utilizando " = = ". Na Figura §7.4 (b) pode ser observado um erro de tipo, onde o tipo *Chr* não existe e deve ser substituído por um tipo válido. Por fim, na Figura §7.4 (c) as correções para a sintaxe e tipo foram feitas e o parágrafo pôde ser validado.

b) Checagem de domínio

A checagem de domínio nem sempre é realizada de forma automática pelo Z/EVES. Quando isto não acontece, cabe ao desenvolvedor intervir para completar a prova. Contudo, Saaltink [99] destaca que está verificação é bastante útil, pois é ela que verifica se as funções parciais foram aplicadas para valores definidos no seu domínio.

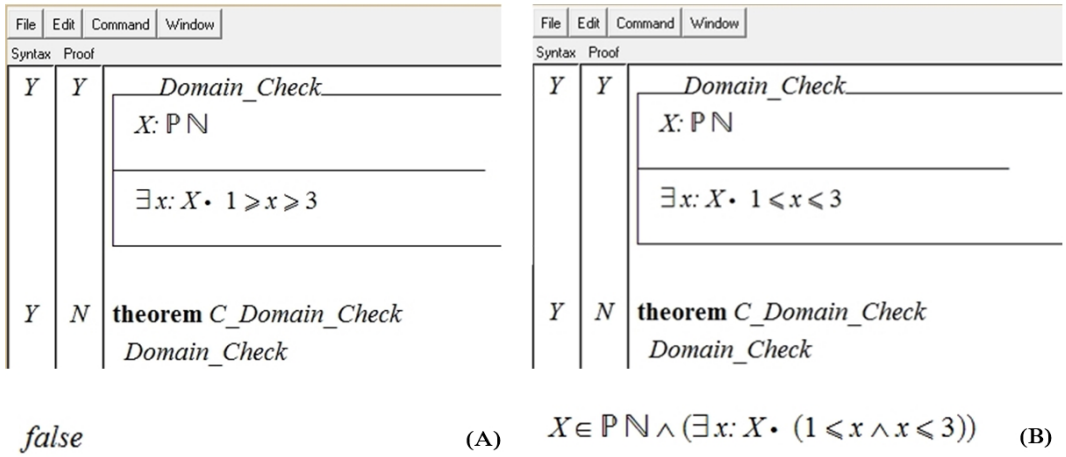


Figura 7.5: *Checagem de Domínio.*

A Figura §7.5 apresenta um exemplo em que é realizada a verificação de domínio de uma restrição simples. Na Figura §7.5 (a) a restrição apresentada está mal formada e é rejeitada pela ferramenta por meio da expressão *false* quando colocada a prova. Isto ocorre, pois a restrição define que **existe** um valor inteiro menor que 1 e maior que 3. Assim a restrição não pode ser satisfeita, pois não apresenta nenhuma possibilidade válida em seu domínio. Já o exemplo apresentado na Figura §7.5 (b) possui, ao menos, uma opção válida para o domínio, sendo que a restrição determina que existe um número inteiro maior que 1 e menor que 3. Com isto a restrição é validada pela ferramenta.

c) Verificação de inconsistências

De acordo com Saaltink [99], uma inconsistência pode ser caracterizada quando uma especificação não possui nenhum modelo válido. Este fator inviabiliza a especificação. Conforme destacado pelo autor, a inconsistência pode ser global tendo seu efeito propagado por toda a especificação ou local quando restrito a uma definição predicativa de um determinado esquema.

Além destas três etapas ainda pode-se considerar a etapa da **Verificação de estado inicial**. Esta etapa garante que existe ao menos um estado que satisfaça as invariantes do sistema. Por fim, a etapa do **Cálculo de pré-condições**, que define o conjunto de estados iniciais e estados intermediários, para os quais existe a definição de um estado posterior.

7.4 Validação do Modelo

A validação da proposta da especificação formal da sintaxe abstrata da linguagem de domínio especifico da tecnologia Guaraná, foi realizada de forma automatizada utilizando como recurso, um notebook Acer, Intel Core 2 Duo processador T5550 (1.83 GHz, 667 MHz, 2 MB L2 cache). Para o trabalho de validação do modelo com a especificação formal foram utilizados os recursos da ferramenta Z-EVES. Com isso, a análise de tempo de processamento para a realização da prova, está baseada nestes recursos.

Foram analisados os 16 parágrafos correspondentes a especificação, os quais estão divididos em 4 parágrafos simples e 12 esquemas. Os esquemas estão subdivididos em dois grupos: 2 parágrafos do tipo *axiom box* e 10 parágrafos do tipo *schema box*.

A validação dos parágrafos simples é realizada de forma implícita pela ferramenta e ocorre de forma totalmente transparente para o desenvolvedor. Já a validação dos esquemas é realizada através da análise de teoremas. Estes são gerados e exibidos pela ferramenta após a validação possibilitando sua análise.

Freitas e Woodcock [37] destacam, em seu trabalho, que o Z/EVES gera um conjunto específico de teoremas para cada um de seus esquema. Além disso, os autores exemplificam um conjunto de teoremas gerados para um determinado esquema, e os descrevem em ordem de relevância para a validação automatizada.

Para os 12 esquemas da especificação formal do Guaraná DSL foram gerados 104 teoremas totalizando mais de 300 linhas de código. Neste sentido torna-se inviável apresentar o conjunto completo de teoremas gerados neste trabalho.

No entanto, baseado na abordagem apresentada por Freitas e Woodcock [37], pode-se apresentar o conjunto de teoremas gerados para o esquema *Task*, com o objetivo de exemplificar a geração e validação da especificação formal da sintaxe abstrata do Guaraná DSL.

Inicialmente, cf. Figura §7.6 é apresentado um teorema para definição do conjunto de relações para o esquema *Task* expandindo os tipos associados até o primeiro nível.

Na Figura §7.7 é apresentado um teorema para inferir sobre o tipo de cada componente declarado que faz parte do esquema *Task*.

theorem grule *Task$declaration*
Task
$\in \mathbb{P}\ \langle executionBody: \mathbb{P}\ (\mathbb{Z} \times Char);$
$inputSlots: \mathbb{P}\ (\langle name: Char \times \mathbb{P}\ (\mathbb{P}\ (\mathbb{Z} \times Char) \times \mathbb{P}\ (\mathbb{Z} \times \mathbb{Z}))\rangle \times$
$\langle name: Char \times \mathbb{P}\ (\mathbb{P}\ (\mathbb{Z} \times Char) \times \mathbb{P}\ (\mathbb{Z} \times \mathbb{Z}))\rangle);$
$inputs: \mathbb{P}\ \langle name: Char \times \mathbb{P}\ (\mathbb{P}\ (\mathbb{Z} \times Char) \times \mathbb{P}\ (\mathbb{Z} \times \mathbb{Z}))\rangle;$
$name: Char \times \mathbb{P}\ (\mathbb{P}\ (\mathbb{Z} \times Char) \times \mathbb{P}\ (\mathbb{Z} \times \mathbb{Z}));$
$outputSlots: \mathbb{P}\ (\langle name: Char \times \mathbb{P}\ (\mathbb{P}\ (\mathbb{Z} \times Char) \times \mathbb{P}\ (\mathbb{Z} \times \mathbb{Z}))\rangle \times$
$\langle name: Char \times \mathbb{P}\ (\mathbb{P}\ (\mathbb{Z} \times Char) \times \mathbb{P}\ (\mathbb{Z} \times \mathbb{Z}))\rangle);$
$outputs: \mathbb{P}\ \langle name: Char \times \mathbb{P}\ (\mathbb{P}\ (\mathbb{Z} \times Char) \times \mathbb{P}\ (\mathbb{Z} \times \mathbb{Z}))\rangle;$
$taskGateway: \mathbb{P}\ \langle name: Char \times \mathbb{P}\ (\mathbb{P}\ (\mathbb{Z} \times Char) \times \mathbb{P}\ (\mathbb{Z} \times \mathbb{Z}))\rangle\rangle$

Figura 7.6: *Teorema com expansão dos tipos.*

theorem frule *Task$declarationPart*
Task
$\Rightarrow name \in Name$
$\wedge\ executionBody \in Text$
$\wedge\ taskGateway \in \mathbb{P}\ Gateway$
$\wedge$ **local** $inputs \in \mathbb{P}\ Gateway$
$\wedge$ **local** $outputs \in \mathbb{P}\ Gateway$
$\wedge\ inputSlots \in relatedInputs \rightarrowtail\!\!\!\!\!\rightarrow inputs$
$\wedge\ outputSlots \in relatedOutputs \rightarrowtail\!\!\!\!\!\rightarrow outputs$

Figura 7.7: *Teorema para os tipos declarados em Task*

O Z/EVES também gera expressões *Theta* (θ) que permitem a representação das restrições de forma resumida e concisa. Estes teoremas constroem uma associação entre os nomes das declarações do esquema e os valores das variáveis de mesmo nome contidas no escopo da expressão.

A Figura §7.8 apresenta o teorema que descreve a relação e associação das variáveis declaradas para o esquema *Task* com as suas respectivas variáveis de estado.

A Figura §7.9 apresenta o teorema de inclusão do esquema *Task* e suas propriedades. Com isso, garante-se que o esquema *Task* faça parte e possa

```
theorem rule Task$thetaInSet
  θ Task
    ∈ ⟨executionBody: executionBody';
      inputSlots: inputSlots';
      inputs: inputs';
      name: name';
      outputSlots: outputSlots';
      outputs: outputs';
      taskGateway: taskGateway'⟩
  ⇔ executionBody ∈ executionBody'
    ∧ inputSlots ∈ inputSlots'
    ∧ local inputs ∈ inputs'
    ∧ name ∈ name'
    ∧ outputSlots ∈ outputSlots'
    ∧ local outputs ∈ outputs'
    ∧ taskGateway ∈ taskGateway'
```

Figura 7.8: *Teorema com as expressões de relação.*

```
theorem rule Task$thetaMember
  θ Task ∈ Task ⇔ Task
```

Figura 7.9: *Teorema para inclusão do esquema Task.*

ser reutilizado no decorrer da especificação.

O teorema apresentado na Figura §7.10 estabelece regras de igualdade para as variáveis do esquema. Tais regras servem para estabelecer a relação entre as variáveis declaradas e as suas variáveis de estado correspondentes.

A Figura §7.11 apresenta sete teoremas que representam as restrições de seleção das sete variáveis declaradas no esquema *Task*.

O teorema apresentado na Figura §7.12 define a condição de existência para um elemento válido para o conjunto *Task*. O mesmo acontece no teorema descrito na Figura §7.13. No entanto com a expansão do esquema *Task*.

Por fim, o último teorema gerado para o esquema *Task*, cf. Figura §7.14

```
theorem rule Task$thetasEqual
  θ Task = θ Task'
  ⇔ executionBody = executionBody'
    ∧ inputSlots = inputSlots'
    ∧ local inputs = inputs'
    ∧ name = name'
    ∧ outputSlots = outputSlots'
    ∧ local outputs = outputs'
    ∧ taskGateway = taskGateway'
```

Figura 7.10: *Teorema para igualdade de variáveis.*

```
theorem grule Task$select$executionBody
  θ Task.executionBody = executionBody
theorem grule Task$select$inputSlots
  θ Task.inputSlots = inputSlots
theorem grule Task$select$inputs
  θ Task.inputs = local inputs
theorem grule Task$select$name
  θ Task.name = name
theorem grule Task$select$outputSlots
  θ Task.outputSlots = outputSlots
theorem grule Task$select$outputs
  θ Task.outputs = local outputs
theorem grule Task$select$taskGateway
  θ Task.taskGateway = taskGateway
```

Figura 7.11: *Teorema para seleção de variáveis.*

apresenta todas as variáveis declaradas para o esquema e define suas restrições de existência limitando seus valores ao conjunto potência da sua variável de estado correspondente.

Além dos teoremas gerados diretamente para o esquema Task, o teorema apresentado na Figura §7.15 define a regra para a declaração do

theorem disabled rule *Task$member*

$$x\$ \in Task \Leftrightarrow (\exists\, Task \bullet x\$ = \theta\, Task)$$

Figura 7.12: *Teorema que define a condição de existência.*

theorem disabled rule *Task$inSet*

$$\begin{aligned} & x\$ \\ & \quad \in \langle executionBody: executionBody'; \\ & \qquad inputSlots: inputSlots'; \\ & \qquad inputs: inputs'; \\ & \qquad name: name'; \\ & \qquad outputSlots: outputSlots'; \\ & \qquad outputs: outputs'; \\ & \qquad taskGateway: taskGateway' \rangle \\ & \Leftrightarrow (\exists\, executionBody: executionBody'; inputSlots: inputSlots'; \\ & \qquad inputs: inputs'; name: name'; outputSlots: outputSlots'; \\ & \qquad outputs: outputs'; taskGateway: taskGateway' \bullet x\$ = \theta\, Task) \end{aligned}$$

Figura 7.13: *Teorema expandido para a condição de existência.*

conjunto Task_Names.

Da mesma forma, foram gerados teoremas para todos os demais esquemas. Os teoremas do tipo *grule, frule* e *rule* foram invocados e provados automaticamente quando a especificação foi executada, como mostra a Figura §7.16.

No entanto, a inserção de restrições de cardinalidade fez com que esquemas do tipo *axiom* fossem gerados. Considerando a especificação formal da sintaxe abstrata do Guaraná DSL apresentada no Capítulo §6, os esquemas *Port, process* e *Solution* consideram esse tipo de restrição. Com isso, três teoremas do tipo *axiom* são gerados pela ferramenta.

Para a validação destes teoremas foi utilizou-se o comando *Prove* da janela de provas da ferramenta Z/EVES. Algumas estatísticas da prova destes três teoremas são apresentadas na Tabela §7.1. Como pode ser observado, os teoremas gerados são bastante extensos e por esse motivo não foram incluídos no corpo deste trabalho.

```
theorem rule Task$setInPowerSet
 ⟨executionBody: executionBody;
  inputSlots: inputSlots;
  inputs: local inputs;
  name: name;
  outputSlots: outputSlots;
  outputs: local outputs;
  taskGateway: taskGateway⟩
  ∈ ℙ ⟨executionBody: executionBody';
      inputSlots: inputSlots';
      inputs: inputs';
      name: name';
      outputSlots: outputSlots';
      outputs: outputs';
      taskGateway: taskGateway'⟩
 ⇔ executionBody ∈ ℙ executionBody'
  ∧ inputSlots ∈ ℙ inputSlots'
  ∧ local inputs ∈ ℙ inputs'
  ∧ name ∈ ℙ name'
  ∧ outputSlots ∈ ℙ outputSlots'
  ∧ local outputs ∈ ℙ outputs'
  ∧ taskGateway ∈ ℙ taskGateway'
  ∨ executionBody = {}
  ∨ inputSlots = {}
  ∨ local inputs = {}
  ∨ name = {}
  ∨ outputSlots = {}
  ∨ local outputs = {}
  ∨ taskGateway = {}
```

Figura 7.14: *Teorema para o conjunto potência.*

Conforme a tabela §7.1, os três teoremas gerados totalizam 198 linhas de código distribuídas ao longo de 6 páginas. No entanto, para a prova dos teoremas, a ferramenta reescreve o código de forma expandida, resultando em um código de 11.176 linhas distribuídas em 276 páginas. O tempo gasto para a prova destes três teoremas é de 8 minutos e 10 segundos.

theorem grule *Tasks_Names$declaration*
$Tasks\_Names \in \mathbb{P}\ Name$

Figura 7.15: *Teorema para Task_Name.*

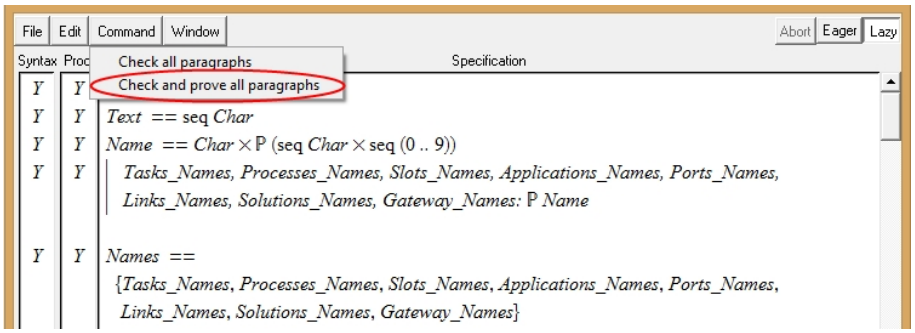


Figura 7.16: *Verificação e prova das propriedades da especificação.*

ESQUEMAS	TEOREMA GERADO			PROVA EXPANDIDA			Duração da Prova
	Páginas	Linhas	Palavras	Páginas	Linhas	Palavras	
Port	1	12	67	1	20	183	14 s
Process	3	118	742	37	1.473	7.242	4 m e 09 s
Solution	2	68	364	238	9.683	17.625	3 m e 47 s
Total	6	198	1.175	276	11.176	25.050	8 m e 10 s

Tabela 7.1: *Estatistica sobre a prova de teoremas do tipo axiom.*

A Tabela §7.2 apresenta a estatística completa de todos os teoremas gerados e provados automaticamente pela ferramenta Z/EVES para a especificação formal da sintaxe abstrata do Guaraná DSL.

Os dados contidos na Tabela §7.2 mostram que a validação da especificação da sintaxe abstrata do Guaraná DSL ocorreu por meio da geração de 104 teoremas. Destes, 52 teoremas são rotulados como sendo do tipo *Grule*, 39 teoremas do tipo *rule*, 10 teoremas do tipo *frule* e 3 teoremas do tipo *axiom*.

ESQUEMAS	TEOREMAS				Total
	Grule	Rule	Frule	Axiom	
Gateway	3	6	1	0	10
Inputs	1	0	0	0	1
Outputs	1	0	0	0	1
RelatedInputs	1	0	0	0	1
RelatedOutputs	1	0	0	0	1
Slot	10	6	1	0	17
Task	9	6	1	0	16
EntryPort	0	0	1	0	1
ExitPort	0	0	1	0	1
Port	7	6	1	1	15
Application	1	3	1	0	5
Link	5	6	1	0	12
Process	12	6	1	1	20
Solution	1	0	1	1	3
Total	52	39	10	3	104

Tabela 7.2: *Estatística sobre os teoremas gerados e provados.*

7.5 Resumo do Capítulo

Esse capítulo apresentou a validação do modelo com a especificação formal da sintaxe abstrata do Guaraná DSL através da ferramenta Z/EVES. Neste sentido, realizou-se a verificação de sintaxe, de tipo e de domínio, além da prova de teoremas gerados automaticamente pela ferramenta. Juntamente com descrição detalhada da especificação formal fornecida por esta dissertação, verificou-se que o modelo apresentado está em conformidade com a definição do metamodelo UML com as restrições OCL definidas para o Guaraná DSL.

Parte IV

Considerações Finais

Capítulo 8

Conclusões

A ciência nos traz conhecimento;
a vida, sabedoria.

Will Durant, Filósofo estadunidense (1885-1981)

O avanço da tecnologia da informação tem proporcionado importantes ferramentas de apoio gerencial para as organizações empresariais. Os recursos computacionais, por meio de aplicações, têm grande capacidade de armazenar e analisar dados e auxiliar na tomada de decisões que podem alavancar o crescimento empresarial. No entanto, alguns problemas ainda atenuam a real capacidade destes recursos. Geralmente, o conjunto de aplicações que compõe o ecossistema de software das organizações não oferece suas funcionalidades de forma integrada. Além disso, muitas destas aplicações foram desenvolvidas sem considerar critérios para integração. Neste contexto, a utilização de soluções de integração proporciona meios para que as organizações consigam fornecer suporte aos processos de negócio de maneira eficiente e precisa, utilizando suas próprias aplicações.

Neste sentido, esta dissertação apresentou a tecnologia Guaraná DSL. Esta tecnologia fornece suporte para o desenvolvimento e implementação de soluções de integração e possui uma linguagem específica de domínio e um mecanismo de tolerância a falhas baseado em regras. No entanto, para que estas regras sejam validadas, ou para que as mesmas possam ser geradas de forma automática, a linguagem da tecnologia Guaraná precisa estar formalizada.

Vários trabalhos encontrados na literatura apresentam a formalização de sistemas de informação. Porém, um número reduzido de trabalhos apresentam a formalização de linguagens. Contudo, alguns trabalhos foram apresentados nesta dissertação. Foram referenciados trabalhos relacionados aos diferentes métodos e linguagens formais apresentados. Notou-se, no entanto, uma incidência maior de trabalhos relacionados que utilizaram a Notação Z para formalizar linguagens. Além disso, vários estudos abordam a especificação formal da linguagem orientada a modelo UML associada com restrições definidas em OCL.

Na sequência, apresentou-se a categorização das linguagens e métodos formais, considerando abordagens de diferentes autores. Partindo desta abordagem foram apresentadas as principais propriedades da Notação Z, Alloy, Método B, RSL e Redes de Petri. Dentre estas linguagens, destacou-se a Notação Z. Isto ocorre, pois a Notação Z apresenta as características necessárias para todo o processo de formalização da tecnologia Guaraná, além de apresentar uma sintaxe próxima à sintaxe usual proveniente da matemática, amplamente conhecida por especialistas das mais diversas áreas de conhecimento.

A Notação Z tem sua estrutura baseada na teoria dos conjuntos e na lógica de primeira ordem. Estas características são exploradas por muitos autores, pois fornece grande capacidade para compor especificações de sistemas ou linguagens. Além disso, a Notação Z possui as principais características da programação orientada a objeto.

Em relação a ferramentas com suporte para edição e validação de uma especificação formal utilizando a Notação Z, destaca-se a ferramenta Z/EVES. Esta ferramenta proporciona um ambiente para edição e fornece recursos para validação da especificação de forma automatizada. Com recursos disponibilizados de forma estável a vários anos, o Z/EVES tem sido importante na formalização de linguagens, considerando que um conjunto considerável dos trabalhos relacionados destacados nesta dissertação, fizeram uso desta ferramenta.

Portanto, a partir da ampla contextualização do problema abordado nesta dissertação, da definição da Notação Z como linguagem formal e da escolha da ferramenta Z/EVES, proporcionou-se a especificação formal da sintaxe abstrata da linguagem de domínio específico da tecnologia Guaraná. A especificação formal teve por base as propriedades definidas para a tecnologia Guaraná DSL, expressa por meio de um metamodelo UML com restrições OCL. Cada propriedade definida no metamodelo UML para o Guaraná DSL

teve sua especificação formal correspondente estabelecida. No entanto, algumas propriedades tiveram que ser adaptadas, pois a especificação formal segue conceitos rigorosos relacionados a operações sobre conjuntos e referência de tipos. Assim, verificou-se a necessidade de fornecer a declaração de tipos adicionais. Além disto, uma operação de relação ou função entre dois conjuntos deve formar um elemento de um novo conjunto. Com isso, não é possível associar este elemento a um dos dois conjuntos participantes desta operação. No entanto, apesar destes fatores, procurou-se manter ao máximo a estruturação original fornecida para as propriedades da linguagem.

A partir do modelo com especificação formal da sintaxe abstrata do Guaraná DSL, realizou-se uma criteriosa descrição das propriedades definidas. De acordo com Moura [83], a descrição textual com linguagem informal é parte integrante de uma especificação. Por meio dela é possível orientar o leitor visando facilitar o entendimento. O mesmo autor ainda considera que esta descrição pode atender os conceitos de prova, isto é, convencer o interlocutor da correta definição das propriedades. Além disso, realizou-se a validação do modelo com a especificação formal, que ocorreu de forma automatizada com recursos proporcionados pela ferramenta Z/EVES. Inicialmente foi realizada a verificação de sintaxe, de tipo e de domínio, seguindo com a validação do modelo por meio da prova de teoremas.

Desta forma, concluiu-se a primeira etapa do processo de formalização da tecnologia Guaraná atribuindo maior rigor e clareza a sintaxe abstrata da linguagem através da Notação Z. Esta especificação formal servirá como base para as etapas de formalização da sintaxe concreta e da semântica. Uma vez concluído todo o processo de formalização, espera-se, por meio dele, tornar possível validar as regras escritas por engenheiros de software utilizando a linguagem baseada em regras do mecanismo de tolerância a falhas da tecnologia Guaraná. Outra possibilidade vislumbrada com a formalização, é gerar automaticamente o conjunto de regras para soluções de integração.

Capítulo 9

Trabalhos Futuros

Procure ser um homem de valor,
em vez de ser um homem de sucesso.

Albert Einstein, físico teórico alemão (1879-1955)

O processo de formalização da tecnologia Guaraná DSL passa por várias etapas. A especificação formal e validação da sintaxe abstrata apresentada nesta dissertação é a primeira delas. Neste sentido, é necessário o desenvolvimento das outras etapas para que a formalização esteja completa.

Inicialmente, é necessário fazer o refinamento da especificação formal da sintaxe abstrata fornecida nesta dissertação. Com o refinamento, é verificada a possibilidade de tornar ainda mais legível a especificação formal por meio da simplificação de alguma propriedade.

Além disso é preciso fornecer uma especificação formal e a validação para a sintaxe concreta e para semântica da tecnologia guaraná DSL. Conforme apresentado na dissertação, uma linguagem de domínio especifico é composta por uma sintaxe abstrata, sintaxe concreta e pela semântica.

Neste sentido, associando todos os trabalhos, será possível estabelecer a formalização da linguagem da tecnologia Guaraná.

Parte V

Apêndice

Apêndice A

Símbolos e Expressões da Notação Z

Simbologia de representação da notação Z conforme o padrão internacional ISO/IEC 13568:2002 [1].

Simbologia Básica

Os símbolos básicos da linguagem de especificação Z estão estritamente vinculados a simbologia matemática usual e podem ser divididos em três grupos. Na tabela §A.1 são apresentados os operadores relacionais.

Símbolos	Descrição
$=$	IGUALDADE
$\neq$	NEGAÇÃO DE IGUALDADE
$\leq$	MENOR OU IGUAL
$<$	MENOR
$\geq$	MAIOR OU IGUAL
$>$	MAIOR

Tabela A.1: *Operadores Relacionais em Z*

Na tabela §A.2 é apresentado o conjunto de operadores lógicos.

Por último, na tabela §A.3 apresenta-se os operadores que atuam sobre os conjuntos.

Ambos os símbolos dos três grupos são muito utilizados em qualquer especificação Z, sendo que, a partir deles, pode-se desenvolver relações e operações mais complexas.

Símbolos	Descrição
$\wedge$	E
$\vee$	OU
$\rightarrow$	IMPLICA
$\leftrightarrow$	SE SOMENTE SE
$\neg$	NEGAÇÃO

Tabela A.2: *Operadores lógicos em Z.*

Símbolos	Descrição
$\forall$	PARA TODO
$\exists$	EXISTE (ao menos um)
$\in$	PERTENCE
$\notin$	NÃO PERTENCE
$\subset$	SUBCONJUNTO
$\subseteq$	SUBCONJUNTO OU IGUAL
$\cup$	UNIÃO
$\cap$	INTERSECÇÃO

Tabela A.3: *Operadores sobre Conjuntos em Z.*

Simbologia complementar

Além dos símbolos básicos apresentados, existem outros símbolos complementares que podem ser utilizados na especificação formal Z. Na tabela §A.4 são apresentados os operadores relacionados aos conjuntos, com a sua descrição, definição e um exemplo de aplicação.

Simbologia para Funções

A utilização de símbolos para representação de funções se faz necessário em Z, pois a função relaciona elementos de conjuntos. De acordo com Spivey (1989)[104] cada elemento de um conjunto pode estar relacionado a no máximo um elemento de outro conjunto. Na tabela §A.5 estão representados os símbolos, a sua descrição e a definição de atua-

Símbolo	Descrição	Definição	Exemplo
$\varnothing$	CONJUNTO VAZIO	Representação de conjunto vazio.	X=$\varnothing$
$\mathbb{P}$	CONJUNTO POTÊNCIA	Representa um conjunto com todos os subconjuntos de X.	Se X={a,b} $\mathbb{P}$X = {$\varnothing$, {a}, {b}, {a,b}}
$\mathbb{F}$	CONJUNTO FINITO	Representa um conjunto finito de elementos de um tipo x.	Se X = N { X : FN | 1 ≤ X ≤ 10}
$\cup$	UNIÃO	Realiza a união entre conjuntos.	Se X = F$\mathbb{N}$ X = X $\cup$ {1,2,3}
\	REMOÇÃO DE CONJUNTO	Remove conjunto de entradas y sobre um conjunto x.	Se X = F$\mathbb{N}$ X = X \ {1,2,3}
#	CONTADOR	Obtém o tamanho de elementos de um determinado conjunto.	#{a,b,c} = 3 (três).
$\longrightarrow$	MAPEAMENTO	Relaciona um elemento X a outro elemento Y, produz um par ordenado (X,Y).	X={1,2}; Y={3,4} X$\longrightarrow$Y = {1,3}
$\times$	PRODUTO CARTESIANO	Representação de um produto cartesiano entre dois conjuntos.	A={1,2}; B={3,4} A$\times$B={{1,3},{1,4},{2,3},{2,4}
$\longleftrightarrow$	RELAÇÃO	Conjunto de todas as relações entre x e y.	A relação é uma abreviação de: X $\longleftrightarrow$Y = = $\mathbb{P}${ X $\times$Y }

Tabela A.4: *Operadores complementares para conjuntos.*

Símbolo	Descrição	Definição
$\rightarrow\!\!\!\!\!+$	FUNÇÃO PARCIAL	A função é parcial quando nem todos os elementos de X podem estar associados a elementos de Y.
$\longrightarrow$	FUNÇÃO TOTAL	A função é total quando todos os elementos de X estão associados a elementos de Y.
$\rightarrowtail\!\!\!\!\!+$	FUNÇÃO PARCIAL INJETORA	A função é parcial injetora quando os elementos de X que possuem associação estão associados a apenas um elemento distinto em Y.
$\rightarrowtail$	FUNÇÃO TOTAL INJETORA	A função é total injetora quando todos os elementos de X estão associados a apenas um elemento distinto em Y.
$\twoheadrightarrow\!\!\!\!\!+$	FUNÇÃO PARCIAL SOBREJETORA	A função é parcial sobrejetora quando todos os elementos em Y que possuam associação estão associados a apenas um elemento distinto em X. Pode haver elementos em Y que não estão relacionados com elementos em X.
$\twoheadrightarrow$	FUNÇÃO TOTAL SOBREJETORA	A função é total sobrejetora quando todos os elementos em Y estão associados a um elemento distinto em X.
$\rightarrowtail\!\!\!\!\!\twoheadrightarrow$	FUNÇÃO TOTAL BIJETORA	A função é total bijetora quando todos os elementos de X possuem apenas um elemento distinto associado em Y.

Tabela A.5: *Símbolos para funções em Z.*

ção de cada função, além de um exemplo de aplicação. Para a descrição são considerados dois conjuntos: X e Y;

Simbologia para Declaração e Predicados

Outro grupo de símbolos utilizados na Notação Z, proporciona a declaração de tipos primitivos e abstratos, além de variáveis, para que possam ser utilizados durante a especificação. Outros símbolos são utilizados para articular a estrutura da parte predicativa dos esquemas. Na tabela §A.6 são

Símbolo	Descrição	Definição	Exemplo
$\mathbb{Z}$	INTEIROS	Representação dos números inteiros.	Int == {x: $\mathbb{Z}$}
$\mathbb{N}$	NATURAIS	Representação dos números naturais.	Nat == {x: $\mathbb{Z}$ | x≥0}
==	ABREVIAÇÃO	Declara um nome para abreviar a referência a um determinado conjunto.	Par == {0, 2, 4, 6, 8}
::=	ABREVIAÇÃO LIVRE	Declara um conjunto com elementos sem que estejam previamente especificados.	Vogais ::={a, e, i, o, u}
|	SEPARADOR DECLARATIVO	Separa a parte declarativa da parte predicativa.	Nat == {x: $\mathbb{Z}$ | x≥0}
{ }	DELIMITADORES DECLARATIVOS	Delimita a declarativa da parte predicativa.	Nat == {x: $\mathbb{Z}$ | x≥0}
[ ]	DECLARAÇÃO DE TIPO	Delimita a declaração de Tipos.	[Task, Process, Solution]
•	SEPARADOR	Separa a declaração de uma expressão, dentro da parte predicativa de um esquema.	y: $\mathbb{N}$ $\exists$ x: $\mathbb{N}$ • (y=2*x)

Tabela A.6: *Símbolos para tipos, declaração e predicados em Z.*

encontrados os principais símbolos deste grupo, com a apresentação da sua descrição e um exemplo de aplicação.

Bibliografia

[1] *Iso/iec 13568: Information technology – z formal specification notation – syntax, type system and semantics*. 2002.

[2] J.-R. Abrial, M. K. Lee, D. Neilson, P. Scharbach e I. H. Sørensen. *The b-method*. Em *VDM'91 Formal Software Development Methods*, páginas 398–405. Springer, 1991.

[3] J. B. Almeida, M. J. Frade, J. S. Pinto e S. M. de Sousa. *An overview of formal methods tools and techniques*. Em *Rigorous Software Development*, páginas 15–44. Springer, 2011.

[4] K. Anastasakis, B. Bordbar, G. Georg e I. Ray. *Uml2alloy: A challenging model transformation*. Em *Model Driven Engineering Languages and Systems*, páginas 436–450. Springer, 2007.

[5] T. Anderson e P. A. Lee. *Fault tolerance*. Prentice-Hall, 1981.

[6] U. Assmann, S. Zschaler e G. Wagner. *Ontologies, meta-models, and the model-driven paradigm*. Em *Ontologies for Software Engineering and Software Technology*, páginas 249–273. Springer, 2006.

[7] C. Attiogbe. *Tool-assisted multi-facet analysis of formal specifications (using alelier-b and prob). arXiv preprint arXiv:0910.1690*, 2009.

[8] J. L. N. Audy, G. K. de Andrade e A. Cidral. *Fundamentos de sistemas de informação*. Bookman, 2007.

[9] C. Baier, J.-P. Katoen e outros. *Principles of model checking*, volume 26202649. MIT press Cambridge, 2008.

[10] Y. Bertot e P. Castéran. *Interactive theorem proving and program development*. Springer Science & Business Media, 2004.

[11] D. Bjørner. *Domain engineering: A software engineering discipline in need of research*. Em *SOFSEM 2000: Theory and Practice of Informatics*, páginas 1–17. Springer, 2000.

[12] D. Bjørner e M. C. Henson. *Logics of specification languages*, volume 18. Springer, 2008.

[13] B. Bordbar e K. Anastasakis. *Uml2alloy: A tool for lightweight modelling of discrete event systems*. Em *IADIS AC*, páginas 209–216, 2005.

[14] A. Bouajjani, J. Esparza e O. Maler. *Reachability analysis of pushdown automata: Application to model-checking*. Em *CONCUR'97: Concurrency Theory*, páginas 135–150. Springer, 1997.

[15] J. P. Bowen e M. G. Hinchey. *Ten commandments of formal methods*. *Computer*, 28(4):56–63, 1995.

[16] J. P. Bowen e M. G. Hinchey. *Ten commandments of formal methods... ten years later*. *Computer*, 39(1):40–48, 2006.

[17] S. Brien e A. Martin. *A calculus for schemas in z*. *Journal of Symbolic Computation*, 30(1):63–91, 2000.

[18] S. M. Brien, J. E. Nicholls e outros. *Z base standard: Version 1.0*. Oxford University Computing Laboratory, Programming Research Group, 1992.

[19] R. H. Campbell e B. Randell. *Error recovery in asynchronous systems*. *Software Engineering, IEEE Transactions on*, (8):811–826, 1986.

[20] K. Channabasavaiah, K. Holley e E. Tuggle. *Migrating to a service-oriented architecture*. *IBM DeveloperWorks*, 16, 2003.

[21] D. Chappell. *Enterprise service bus*. "O'Reilly Media, Inc.", 2004.

[22] D. Chiorean, M. Paşca, A. Cârcu, C. Botiza e S. Moldovan. *Ensuring uml models consistency using the ocl environment*. *Electronic Notes in Theoretical Computer Science*, 102:99–110, 2004.

[23] P. Ciancarini, F. Vitali e C. Mascolo. *Managing complex documents over the www: a case study for xml*. *Knowledge and Data Engineering, IEEE Transactions on*, 11(4):629–638, 1999.

[24] S. Cranefield e M. Purvis. *Uml as an ontology modelling language*. 1999.

[25] F. A. Cummins. *Enterprise integration: an architecture for enterprise application and systems integration*. John Wiley & Sons, Inc., 2002.

[26] N. Debnath, A. Funes, A. Dasso, G. Montejano, D. Riesco e R. Uzal. *Integrating ocl expressions into rsl specifications*. Em *Electro/Information Technology, 2007 IEEE International Conference on*, páginas 158–162. IEEE, 2007.

[27] N. J. Dingle, W. J. Knottenbelt e T. Suto. *Pipe2: a tool for the performance evaluation of generalised stochastic petri nets*. *ACM SIGMETRICS Performance Evaluation Review*, 36(4):34–39, 2009.

[28] D. Dossot, J. D'Emic e V. Romero. *Mule in action*. Manning, 2009.

[29] A. K. Dwivedi e S. K. Rath. *Model to specify real time system using z and alloy languages: A comparative approach*. 2012.

[30] M. V. Fasie. *An eclipse based development environment for raise*. Tese Doutoral, Masters thesis, DTU Compute, Technical University of Denmark, to appear May, 2013.

[31] M. Fisher, J. Partner, M. Bogoevici e I. Fuld. *Spring integration in action*. Manning Publications Co., 2012.

[32] M. Fowler. *Domain-specific languages*. Pearson Education, 2010.

[33] R. Z. Frantz e R. Corchuelo. *A software development kit to implement integration solutions*. Em *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, páginas 1647–1652. ACM, 2012.

[34] R. Z. Frantz. *Enterprise application integration: An easy-to-maintain model-driven engineering approach*. 2012.

[35] R. Z. Frantz, R. Corchuelo e C. Molina-Jiménez. *A proposal to detect errors in enterprise application integration solutions*. *Journal of Systems and Software*, 85(3):480–497, 2012.

[36] R. Z. Frantz, A. M. R. Quintero e R. Corchuelo. *A domain-specific language to design enterprise application integration solutions*. *International Journal of Cooperative Information Systems*, 20(02):143–176, 2011.

[37] L. Freitas e J. Woodcock. *Mechanising mondex with z/eves*. *Formal Aspects of Computing*, 20(1):117–139, 2008.

[38] L. Freitas. *Proving theorems with z/eves*. *Appendix A*, 1(1), 2004.

[39] H. Fuks e M. Pimentel. *Sistemas colaborativos*. Elsevier Brasil, 2012.

[40] C. George. *Tutorial on the raise language, method and tools*. Em *Formal Methods and Software Engineering*, páginas 3–4. Springer, 2004.

[41] C. George. *Raise tool user guide*. *UNU/IIST, Macau, Tech. Rep*, 227, 2008.

[42] C. George e A. E. Haxthausen. *The logic of the raise specification language*. *Computing and Informatics*, 22(3-4):323–350, 2012.

[43] C. George, A. E. Haxthausen, S. Hughes, R. Milne, S. Prehn e J. S. Pedersen. *The raise development method*. Prentice Hall Int., 1995.

[44] S. Getir, M. Challenger e G. Kardas. *The formal semantics of a domain-specific modeling language for semantic web enabled multi-agent systems*. *International Journal of Cooperative Information Systems*, 23 (03), 2014.

[45] J. B. Goodenough. *Exception handling: issues and a proposed notation*. *Communications of the ACM*, 18(12):683–696, 1975.

[46] D. Harel e B. Rumpe. *Meaningful modeling: what' s the semantics of "semantics"?* *Computer*, 37(10):64–72, 2004.

[47] A. E. Haxthausen, J. S. Pedersen e S. Prehn. *Raise: a product supporting industrial use of formal methods*. *Technique et Science Informatiques*, 12(3), 1993.

[48] F. Heitmann, D. Moldt, K. Mortensen e H. Rölke. *Petri nets tools database quick overview*, 2008.

[49] G. Hohpe e B. Woolf. *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley Professional, 2004.

[50] C. Holloway. *Epistemology, software engineering and formal methods*. *The Role of Computers in Research and Development at Langley Research Center*, páginas 570–595, 1994.

[51] P. Hudak. *Domain-specific languages. Handbook of Programming Languages*, 3:39–60, 1997.

[52] C. Ibsen e J. Anstey. *Camel in action*. Manning Publications Co., 2010.

[53] D. Jackson. *Lightweight formal methods*. Em *FME 2001: Formal Methods for Increasing Software Productivity*, páginas 1–1. Springer, 2001.

[54] D. Jackson. *Software abstractions: logic, language, and analysis*. MIT press, 2012.

[55] E. Jackson e J. Sztipanovits. *Formalizing the structural semantics of domain-specific modeling languages*. *Software & Systems Modeling*, 8 (4):451–478, 2009.

[56] T. Jiang e X. Wang. *Formalizing domain-specific metamodeling language xmml based on first-order logic*. *Journal of Software*, 7(6): 1321–1328, 2012.

[57] S. Kelly e J.-P. Tolvanen. *Domain-specific modeling: enabling full code generation*. John Wiley & Sons, 2008.

[58] S.-K. Kim e C. David. *Formalizing the uml class diagram using object-z*. Em *«UML»'99 – The Unified Modeling Language*, páginas 83–98. Springer, 1999.

[59] M. J. Klein, S. Sawicki, F. Roos-Frantz e R. Z. Frantz. *On the formalisation of an application integration language using z notation*. *In Proceedings of the 16th International Conference on Enterprise Information Systems*, páginas 314–319, 2014.

[60] R. Kling. *Computerization and controversy: value conflicts and social choices*. Morgan Kaufmann, 1996.

[61] T. Kosar, N. Oliveira, M. Mernik, V. J. M. Pereira, M. Crepinsek, C. D. Da e R. P. Henriques. *Comparing general-purpose and domain-specific languages: An empirical study*. *Computer Science and Information Systems*, 7(2):247–264, 2010.

[62] K. Lano e H. Haughton. *Specification in b: An introduction using the b toolkit*. World Scientific, 1996.

[63] J. Laprie. *Dependability handbook. laas report n 98-346. Toulouse: Laboratory for Dependability Engineering*, 1998.

[64] K. C. Laudon e J. P. Laudon. *Sistemas de informação: com internet*. LTC Editora, 1999.

[65] H. Ledang e J. Souquières. *Integration of uml and b specification techniques: Systematic transformation from ocl expressions into b*. Em *Asia-Pacific Software Engineering Conference*, páginas 495–495. IEEE Computer Society, 2002.

[66] H. Ledang, J. Souquieres e outros. *Formalizing uml behavioral diagrams with b*. Em *the Tenth OOPSLA Workshop on Behavioral Semantics: Back to Basics*, páginas 162–171, 2001.

[67] Á. Lédeczi, A. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, J. Sprinkle e G. Karsai. *Composing domain-specific design environments*. *Computer*, 34(11):44–51, 2001.

[68] Y. Ledru. *Identifying pre-conditions with the z/eves theorem prover*. Em *Automated Software Engineering, 1998. Proceedings. 13th IEEE International Conference on*, páginas 32–41. IEEE, 1998.

[69] P. A. Lee e T. Anderson. *Fault tolerance principles and practice, volume 3 of dependable computing and fault-tolerant systems*, 1990.

[70] D. S. Linthicum. *Enterprise application integration*. Addison-Wesley Professional, 2000.

[71] J. P. López-Grao, J. Merseguer e J. Campos. *From uml activity diagrams to stochastic petri nets: application to software performance engineering*. Em *ACM SIGSOFT software engineering notes*, volume 29, páginas 25–36. ACM, 2004.

[72] P. R. Maciel, R. D. Lins e P. R. Cunha. *Introdução às redes de petri e aplicações*. UNICAMP-Instituto de Computacao, 1996.

[73] R. Marcano e N. Levy. *Using b formal specifications for analysis and verification of uml/ocl models*. Em *In Workshop on consistency problems in UML-based software development. 5th International Conference on the Unified Modeling Language*, páginas 91–105. Citeseer, 2002.

[74] M. V. Mauco, M. Leonard, D. Riesco, G. Montejano e N. Debnath. *Formalising a derivation strategy for formal specifications from natural language requirements models*. Em *Signal Processing and Information Technology, 2005. Proceedings of the Fifth IEEE International Symposium on*, páginas 646–651. IEEE, 2005.

[75] M. V. Mauco, M. C. Leonardi e D. Riesco. *Deriving formal specifications from natural language requirements*. *Encyclopedia of Information Science and Technology*, páginas 1007–1015, 2009.

[76] I. Meisels e M. Saaltink. *The z/eves reference manual (for version 1.5). Reference manual, ORA Canada*, 1997.

[77] S. J. Mellor. *Mda distilled: principles of model-driven architecture*. Addison-Wesley Professional, 2004.

[78] P. B. Menezes. *Linguagens formais e autômatos*, volume 5. Bookmann, 2008.

[79] F. Menge. *Enterprise service bus*. Em *Free and open source software conference*, volume 2, páginas 1–6, 2007.

[80] D. G. Messerschmitt e C. Szyperski. *Software ecosystem. Understanding an Indispensable Technology and Industry. Massachusetts Institute of Technology, Cambridge, MA*, 2003.

[81] E. Meyer e J. Souquières. *A systematic approach to transform omt diagrams to a b specification*. Em *FM 99 Formal Methods*, páginas 875–895. Springer, 1999.

[82] A. M. Mostafa, M. A. Ismail, H. El-Bolok e E. Saad. *Toward a formalization of uml2.0 metamodel using z specifications*. Em *Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2007. SNPD 2007. Eighth ACIS International Conference on*, volume 1, páginas 694–701. IEEE, 2007.

[83] A. V. Moura. *Especificações em z: uma introdução*. Unicamp, 2001.

[84] T. Murata. *Petri nets: Properties, analysis and applications*. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[85] M. R. Nami e F. Hassani. *A comparative evaluation of the z, csp, rsl, and vdm languages*. *ACM SIGSOFT Software Engineering Notes*, 34 (3):1–4, 2009.

[86] A. Palmisano e A. M. Rosini. *Administração de sistemas de informação ea gestão do conhecimento*. Cengage Learning Editores, 2003.

[87] D. K. Pradhan. *Fault-tolerant computer system design*. Prentice-Hall, 1996.

[88] R. S. Pressman. *Engenharia de software*. McGraw Hill Brasil, 2002.

[89] A. Raja e D. Lakshmanan. *Domain specific languages*. *International Journal of Computer Applications*, 1(21):99–105, 2010.

[90] M. V. M. Ramos, J. J. Neto e Í. S. Veja. *Linguagens formais: teoria, modelagem e implementação*. Bookman, 2009.

[91] A. Regayeg, A. H. Kacem e M. Jmaiel. *Specification and design of multi-agent applications using temporal z*. Em *Intelligent Agents and Multi-Agent Systems*, páginas 228–242. Springer, 2005.

[92] W. Reisig. *Petri nets: an introduction*. Springer-Verlag New York, Inc., 1985.

[93] D. A. Rezende. *Tecnologia da informação integrada à inteligência empresarial*. Atlas, 2002.

[94] D. A. Rezende. *Engenharia de software e sistemas de informação*. Brasport, 2005.

[95] M. Richters e M. Gogolla. *On formalizing the uml object constraint language ocl*. Em *Conceptual Modeling–ER'98*, páginas 449–464. Springer, 1998.

[96] M. Richters e M. Gogolla. *Validating uml models and ocl constraints*. Em *«UML»'2000 – The Unified Modeling Language*, páginas 265–277. Springer, 2000.

[97] D. Roe, K. Broda e A. Russo. *Mapping uml models incorporating ocl constraints into object-z*. Imperial College of Science, Technology and Medicine, Department of Computing, 2003.

[98] M. Saaltink. *The z/eves system*. Em *ZUM'97: The Z Formal Specification Notation*, páginas 72–85. Springer, 1997.

[99] M. Saaltink. *The z/eves 2.0 userâs guide*. Ora Canada, 1999.

[100] P. Schobbens, P. Heymans e J.-C. Trigaux. *Feature diagrams: A survey and a formal semantics*. Em *Requirements Engineering, 14th IEEE international conference*, páginas 139–148. IEEE, 2006.

[101] R. W. Sebesta. *Conceitos de linguagens de programação*. Bookman, 2010.

[102] M. Shroff e R. B. France. *Towards a formalization of uml class structures in z*. Em *Computer Software and Applications Conference, 1997. COMPSAC'97. Proceedings., The Twenty-First Annual International*, páginas 646–651. IEEE, 1997.

[103] C. Snook e M. Butler. *Uml-b: Formal modeling and design aided by uml*. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(1):92–122, 2006.

[104] J. M. Spivey. *An introduction to z and formal specifications*. *Software Engineering Journal*, 4(1):40–50, 1989.

[105] J. M. Spivey. *The z notation: a reference manual. international series in computer science*, 1992.

[106] J. Sun, H. Zhang, Y. Fang e H. Wang. *Formal semantics and verification for feature modeling*. Em *Engineering of Complex Computer Systems, 2005. ICECCS 2005. Proceedings. 10th IEEE International Conference on*, páginas 303–312. IEEE, 2005.

[107] A. Svendsen, B. Møller-Pedersen, Ø. Haugen, J. Endresen e E. Carlson. *Formalizing train control language: automating analysis of train stations*. Em *Comprail*, páginas 245–256, 2010.

[108] S. Tarkan. *The formal specification of a kitchen environment. Masterâs scholarly paper, University of Maryland*, 2009.

[109] S. L. Tonsig. *Engenharia de software: análise e projeto de sistemas*. Ciência Moderna, 2008.

[110] I. Toyn e J. A. McDermid. *Cadi: An architecture for z tools and its implementation*. *Software: Practice and Experience*, 25(3):305–330, 1995.

[111] I. van den Berk, S. Jansen e L. Luinenburg. *Software ecosystems: a software ecosystem strategy assessment model*. Em *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, páginas 127–134. ACM, 2010.

[112] A. Van Deursen, P. Klint e J. Visser. *Domain-specific languages*. *Centrum voor Wiskunde en Informatika*, 5:12, 2000.

[113] J. Woodcock e J. Davies. *Using z: specification, refinement, and proof*. Prentice-Hall, Inc., 1996.

[114] L. Yu, S. Ramaswamy e J. Bush. *Symbiosis and software evolvability*. *IT Professional*, 10(4):56–62, 2008.

[115] T. Zhang, X. Xiao e L. Qian. *Modeling object-oriented framework with z*. Em *Computer Science and Computational Technology, 2008. ISCSCT'08. International Symposium on*, volume 2, páginas 165–170. IEEE, 2008.

[116] P. Ziemann e M. Gogolla. *An extension of ocl with temporal logic*. Em *Critical Systems Development with UML–Proceedings of the UML*, volume 2, páginas 53–62. Citeseer, 2002.

[117] A. Zimmermann e M. Knoke. *Timenet 4.0: A software tool for the performability evaluation with stochastic and colored petri nets; user manual*. Fakultätsdr., 2007.

This document was typeset on April 22, 2015 using class RG-BOOK α2.14 for LaTeX$_{2\varepsilon}$. As of the time of writing this document, this class is not publicly available. Only members of The Distributed Group (TDG) and the Applied Computing Research Group (GCA) are allowed to typeset their documents using this class.