Capítulo XX

Formalização Matemática de uma Linguagem de Domínio Específico Voltada à Integração de Aplicações Utilizando a Notação Z

Mauri J. Klein, Sandro Sawicki, Fabricia Roos-Frantz e Rafael Z. Frantz

Programa de Pós-Graduação Stricto Sensu em Modelagem Matemática

Universidade Regional do Noroeste do Estado do Rio Grande do Sul

Ijuí – RS – Brasil

Resumo

A integração de aplicações é de grande relevância para as empresas atuais, pois permite melhorar seus processos de negócio utilizando as aplicações já existentes no seu ecossistema de *software*. A tecnologia Guaraná é resultado de trabalhos de pesquisa realizados para oferecer suporte à modelagem, implementação e execução de soluções de integração de aplicações empresariais. Um elemento central desta tecnologia é a linguagem de modelagem. Essa linguagem permite aos engenheiros de *software* modelar de forma gráfica soluções de integração com alto nível de abstração. Guaraná, diferente de outras propostas, possui um sistema de monitoramento que pode ser configurado usando uma linguagem baseada em regras para dotar as soluções com tolerância a falhas. Atualmente, porém, não é possível validar as regras escritas pelos engenheiros de *software*, de forma que se possa garantir que as possibilidades de falha de uma dada solução estejam cobertas. Além disso, não é possível gerar automaticamente essas regras. Essas limitações devem-se à falta de formalização da linguagem de modelagem Guaraná, ou seja, à falta de uma definição rigorosa da semântica da linguagem. Neste trabalho, desenvolveu-se um modelo para a especificação formal da sintaxe abstrata da linguagem Guaraná, utilizando a notação matemática Z, a fim de

contribuir para a formalização completa dessa linguagem. A validação desse modelo ocorreu por meio da ferramenta Z/Eves, a qual proporciona a verificação de sintaxe, de tipo e de domínio, assim como prova de teoremas.

1 Introdução

A utilização de sistemas de informação por meio da Tecnologia da Informação (TI) vem ganhando cada vez mais destaque no mundo globalizado. Aplicações para *smartphones*, computadores pessoais, comércio eletrônico e prestação dos mais diversos tipos de serviço, são alguns exemplos do emprego destes sistemas.

Organizações empresariais normalmente dependem de aplicações computacionais para oferecer suporte aos seus processos de negócio. As aplicações utilizadas pelas organizações formam um ecossistema de *software* [1], em que geralmente cada aplicação fornece suporte a um setor ou departamento específico, muitas vezes sem que uma possa funcionar de forma integrada com a outra. Isto ocorre como consequência do desenvolvimento de aplicações pela própria organização, pelo reúso de aplicações legadas ou pela compra de pacotes de *softwares* de terceiros que visam a atender demandas específicas da organização em um determinado momento.

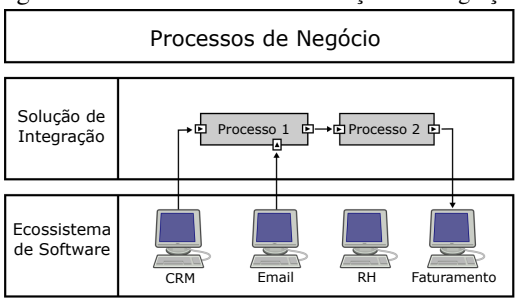
Este cenário, porém, exige que sejam empregados mais tempo e recursos para a manutenção das aplicações. Com a modularização dos sistemas de informação é necessário que dados e informações sejam replicados e atualizados em diversas bases de dados, aumentando a possibilidade de ocorrerem inconsistências (ex: dados desatualizados), além de elevar a demanda por recursos computacionais, tornando o sistema mais lento, podendo resultar em perdas para a organização. Neste contexto, a integração de aplicações torna-se necessária e fundamental.

A Integração de Aplicações Empresariais (do inglês *Enterprise Application Integration - EAI*) fornece metodologias, técnicas e ferramentas para projetar e implementar soluções de integração [2, 3]. O objetivo de uma solução de integração é manter uma série de dados das aplicações em sincronia ou desenvolver novas funcionalidades sobre as já existentes. Uma solução de integração parte do princípio de que as aplicações não devam (e em muitos casos nem

mesmo podem) ser alteradas, o funcionamento das aplicações seja minimamente impactado e estas não se tornem dependentes da solução de integração para poderem funcionar [2].

A Figura 1 ilustra um cenário típico de uma solução de integração, composta de dois processos que contêm a lógica da integração e por cinco portas de comunicação que permitem a interação entre os processos e com as aplicações.

Figura 1. Visão abstrata de uma solução de integração



Fonte: Adaptada de Inma Hernández et al. (2015) [4].

A grande demanda por soluções de integração se justifica pela presença maciça de ecossistemas de *software* heterogêneos nas organizações, o que constitui um ambiente desafiador para concepção e desenvolvimento de tecnologias para este fim. As tecnologias Camel [5], Spring Integration [6], Mule [7] e Guaraná [8], destacam-se por oferecer suporte aos padrões de integração documentados por Hohpe e Woolf (2003) [2] e que são amplamente utilizados pela comunidade de EAI. Essas tecnologias proporcionam uma linguagem de domínio específico para a modelagem das soluções de integração, bem como um *framework* de programação voltado para a implementação desses modelos em código executável.

Considerando que uma solução de integração ao ser implementada nada mais é do que um programa de computador, tais soluções estão também sujeitas a falhas durante seu funcionamento. Assim, é importante que uma solução de integração possa ser tolerante a falhas, ou seja, que possa seguir funcio-

nando e mantendo seus serviços mesmo na ocorrência de falhas. Nesse contexto, o monitoramento e a detecção de erros em soluções de integração são tarefas fundamentais. Camel, Spring Integration e Mule fornecem um mecanismo de detecção de erros baseado principalmente na utilização do mecanismo *try-catch* [9] oferecido pelas linguagens de programação. A pesquisa apresentada nesse capítulo tem como foco a tecnologia Guaraná, em virtude de que a mesma se diferencia das demais por possuir um mecanismo para a detecção de erros de mais alto nível de abstração, baseado em um monitor externo, que pode ser configurado utilizando uma linguagem baseada em regras. Nessa tecnologia, tais regras devem ser definidas pelos engenheiros de software com base no modelo conceitual das soluções de integração [3], no entanto atualmente a tecnologia carece de uma validação dessas regras.

A validação das regras escritas ou geradas com a tecnologia Guaraná é uma forma de garantir que as possibilidades de falha em uma determinada solução de integração, modelada com Guaraná, estejam cobertas, no entanto, para a validação dessas regras, faz-se necessário um mecanismo formal de verificação. Além disso, a formalização da linguagem da tecnologia Guaraná viabiliza a geração automática do conjunto de regras, com base na semântica da solução de integração.

Na literatura existem inúmeros métodos formais que podem ser utilizados para formalização de sistemas e linguagens, tais como: Alloy, Raise, B, Notação Z e Redes de Petri. Observa-se que, no contexto da formalização de linguagens, a Notação Z é amplamente utilizada. A Notação Z tem como base a simplicidade da teoria dos conjuntos e a lógica de primeira ordem, facilitando a compreensão para especialistas de diferentes áreas de estudo. Além disso, apresenta um embasamento teórico que serve como base para diversos outros métodos como B, Alloy e RSL.

A especificação formal de linguagens por meio da Notação Z tem sido utilizada em diversos trabalhos. Mostafa et al. (2007) [10], Shroff e France (1997) [11] e Kim e Carrington (1999) [12] propõem a formalização da sintaxe de

diagramas UML (diagrama de caso de uso, diagrama de classe e diagrama de máquina de estados) com o objetivo de expressar precisamente o seu significado. Richters (1998) [13] apresentam uma semântica formal para a linguagem OCL [14] e para alguns aspectos centrais dos modelos de classe UML. Roe (2003) [15] propõem um mapeamento para traduzir sistemas modelados em UML com restrições especificadas em OCL. Jiang (2012) [16] e Jackson, Wang e Sztipanovitz (2009) [17] propõem a formalização da linguagem de modelagem de domínio específico XMML, além de proporcionar algoritmos para a análise de linguagens de modelagem de domínio específico e para a transformação de modelos.

O objetivo desse capítulo é propor um modelo, utilizando a Notação Z, para a especificação formal da sintaxe abstrata da linguagem de domínio específico da tecnologia Guaraná, a fim de contribuir para a formalização desta linguagem. Esta formalização viabiliza a construção de um mecanismo formal para a validação das regras escritas por engenheiros de *software*, bem como a sua geração de foram automática.

As demais seções deste capítulo estão estruturadas da seguinte forma. A Seção 2 fornece uma descrição simplificada das características da tecnologia Guaraná. A Seção 3 aborda os principais conceitos sobre a Notação Z. A Seção 4 apresenta a formalização da sintaxe abstrata para a linguagem da tecnologia Guaraná. A Seção 5 fornece a validação da especificação formal. Por fim, a Seção 6 descreve as conclusões e trabalhos futuros.

2 Tecnologia Guaraná

A tecnologia Guaraná proporciona construtores que permitem os engenheiros de *software* elaborar soluções de integração de forma simples e intuitiva. Ela oferece uma linguagem de domínio específico (do inglês *Domain-Specific Language - DSL*), baseada nos padrões de integração documentados por Hohpe e Woolf (2003) [2], com uma sintaxe concreta gráfica para projetar soluções de

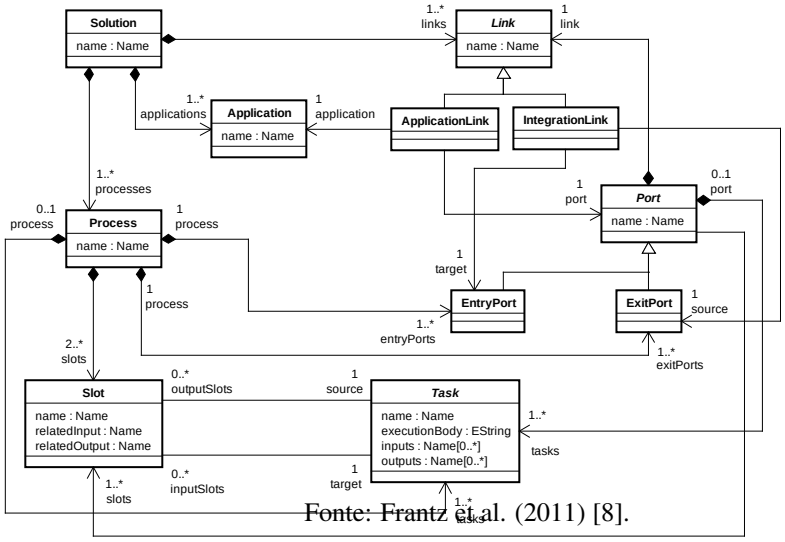
integração de aplicações empresariais com um alto nível de abstração. Guaraná DSL, assim denominada essa linguagem, tem também como característica que os modelos conceituais projetados com ela são modelos independentes de plataforma, ou seja, não possuem qualquer dependência com a tecnologia na qual serão posteriormente implementados e, portanto, podem ser utilizados para a geração de soluções de integração executáveis tanto com a tecnologia Guaraná como as demais mencionadas na introdução desse capítulo.

Essa tecnologia também oferece ferramentas para elaborar soluções de integração. Ela possui um Kit de Desenvolvimento de Software (do inglês *Software Development Kit - SDK*), denominado Guaraná SDK, que corresponde a uma implementação Java do Guaraná DSL. O Guaraná SDK está organizado em duas camadas: o *framework*, que possui uma série de classes e interfaces que fornecem a base para implementar tarefas, adaptadores e fluxos de integração, e o *kit* de ferramentas que estende o *framework* proporcionando um conjunto concreto de tarefas, adaptadores e um motor no qual implantar e executar as soluções de integração. Além deste *kit*, outros podem ser desenvolvidos para atender outros contextos de negócio específicos [18].

A Figura 2 apresenta um metamodelo UML que descreve a sintaxe abstrata do Guaraná DSL, ou seja, os elementos e a estrutura da linguagem. Como se pode observar na figura, a linguagem é composta por um conjunto de blocos construtores, tais como *ports, tasks, slots* e *links*, adiante designados por portas, tarefas, *slots* e *links*. Cada construtor representa um conceito da área de integração de aplicações empresariais. Assim, uma solução de integração modelada com o Guaraná DSL utiliza esses blocos construtores para definir sua lógica de integração.

Uma solução de integração define um conjunto de processos que se comunicam e integram um conjunto de aplicações do ecossistema de *software* das empresas. Processos são unidades centrais em uma solução de integração, e são compostos por portas e tarefas. Um processo pode ser visto de foram geral como um processador de mensagens, enquanto que uma mensagem representa a informação que é trocada e transformada ao longo do fluxo de integração de uma solução de integração. A estrutura e o tamanho dessas mensagens depen-

Figura 2. Metamodelo do Guaraná DSL



Fonte: Frantz et al. (2011) [8].

dem do problema de integração e das soluções de integração que as processa. Um modelo conceitual de uma solução de integração, portanto, é composto por um ou mais processos pelos quais fluem mensagens a serem processadas.

As tarefas representam operações atômicas que são executadas sobre mensagens dentro dos processos. O encadeamento de tarefas define o fluxo de integração e, portanto, o processamento a que as mensagens estarão submetidas ao longo da solução de integração. As tarefas se comunicam entre si por meio de *slots*, os quais funcionam como *buffers* de mensagens, o que torna possível o processamento assíncrono das mensagens ao longo do fluxo de integração.

Os processos utilizam portas para se comunicar uns com os outros e com as aplicações envolvidas em uma solução de integração. De maneira geral, as portas são responsáveis por abstrair os detalhes necessários para interagir com as aplicação integradas ou com outro processo da solução de integração. As

portas podem ser 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 também compostas de tarefas. Para mais detalhes sobre Guaraná DSL, o leitor pode consultar [8].

3 Notação Z

A utilização de uma notação formal para descrever sistemas, linguagens de domínio específico ou de propósito geral, é um fator determinante para o aumento da qualidade e eficiência das aplicações mediadas por computador. Os conceitos matemáticos utilizados em uma notação formal fazem com que a descrição de uma especificação seja representada de maneira clara e precisa, possibilitando ao projetista alcançar resultados que estejam em conformidade com as especificações e limitações pré-definidas [19, 20].

A Notação Z é um método formal utilizado para descrever e modelar linguagens e sistemas computacionais. Esta notação foi proposta, inicialmente, por Jean-Raymond Abrial em 1977 e desenvolvida, posteriormente, pelo Grupo de Pesquisa em Programação na Universidade de Oxford, Inglaterra. De acordo com Moura (2001) [21], o fato de a Notação Z basear-se na teoria dos conjuntos e na lógica de primeira ordem faz com que as suas especificações sejam mais simples e compreensíveis. Ela utiliza-se de conceitos fundamentais da matemática que são amplamente estudados no meio acadêmico e científico [19, 21] .

Na Notação Z, as características dos elementos são definidas por meio da criação de tipos abstratos. Com isso, os elementos com tipos abstratos iguais podem fazer parte do mesmo conjunto. Essa funcionalidade torna possível a relação entre elementos e conjuntos, ou apenas entre conjuntos. Moura (2001) [21] destaca que para exercer qualquer operação sobre conjuntos em

Z, é necessário que todos os elementos estejam bem definidos e que sejam vinculados aos mesmos tipos abstratos. Assim, não é possível fazer a união entre um conjunto de tarefas e um conjunto de processos. Para finalizar, o autor destaca que as operações ocorrem entre os conjuntos e não entre os elementos e os conjuntos.

Além da teoria dos conjuntos, a Notação Z também utiliza a lógica de primeira ordem para modelar e descrever linguagens e sistemas. A utilização de lógica matemática em Z está relacionada com a possibilidade de as expressões assumirem um valor verdadeiro ou falso. As expressões, por sua vez, podem combinar certos predicados, elaborando novas expressões. Para a criação das expressões são utilizados operadores conectivos, tais como: a conjunção, a disjunção, a implicação, a equivalência e a negação. Além disso, é possível construir expressões utilizando vários conectivos formando predicados mais complexos [21].

A Notação Z preenche o requisito fundamental da formalização, ou seja, o rigor matemático para especificar as propriedades e prova de teoremas para validar a especificação. Segundo Getir et al. (2014) [20] e Moura (2001) [21], a popularidade e a flexibilidade da Notação Z deve-se a utilização da noção de módulos, que em sua estrutura são chamados de esquemas. Estes esquemas são utilizados de forma *button/up*, ou seja, define-se os esquemas básicos com características específicas, com isso podem ser reaproveitados como parte de esquemas mais abrangentes.

4 Formalização da Sintaxe Abstrata

Com base no metamodelo UML apresentado na Seção 2, utilizou-se a ferramenta Z/Eves [22] para realizar a especificação formal da sintaxe abstrata da linguagem de domínio específico da tecnologia Guaraná. Cada propriedade definida no metamodelo UML da tecnologia Guaraná resultou em uma definição equivalente com a Notação Z. Cabe ressaltar que algumas propriedades, no

entanto, tiveram de ser adaptadas ao contexto do método formal, obedecendo conceitos rigorosos relacionados às operações sobre os conjuntos e referência de tipos.

4.1 Tipos Básicos

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

Nesse sentido, foi definido o tipo básico *Char* como "[*Char*]" 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* definido como *"Text == seq Char"* formado por uma sequência de caracteres. Este tipo é utilizado para representar os *Scripts* que implementam a lógica de negócios das tarefas que compõem uma solução.

Com a definição dos tipos básicos, tem-se suporte para avançar na especificação visando a estruturar as demais declarações e restrições. De acordo com o metamodelo UML da tecnologia Guaraná, todos os blocos construtores são identificados por um nome, que deve ser exclusivo em seu conjunto. Nesse sentido, inicialmente, definiram-se as restrições para a composição de *Name*.

O tipo *Name* é formado por um par ordenado, em que o primeiro elemento é, obrigatoriamente, um *Char* e o segundo elemento é formado pelo conjunto potência, que é formado pela relação entre uma sequência de caracteres e uma sequência de dígitos formada pelos números inteiros de 0 a 9, sendo então definido como "*Name == Char* $\times\ \mathbb{P}\ (seq\ Char \times seq\ (0..9))$."

Dada as restrições para o tipo *Name*, o qual identifica todas as instâncias dos blocos de construção em uma solução de integração, definiu-se os seguintes

subconjuntos que são compostos por elementos deste tipo: "*Tasks_Names, Processes_Names, Slots_Names, Applications_Names, Ports_Names, Links_Names, Solutions_Names, Gateway_Names*: $\mathbb{P}$ Name"

A identificação é realizada por meio de um nome que pertence a um determinado subconjunto de nomes atribuídos a um bloco de construção. Com isso, também deve-se separar estes subconjuntos ao nível de unicidade exigido. 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.

O conjunto *Names* é dado por "*Names* == *Tasks_Names, Processes_Names, Slots_Names, Applications_Names, Ports_Names, Links_Names, Solutions_Names, Gateway_Names*". Este conjunto é formado por todos os subconjuntos de nomes. Com isso, foram definidos os tipos básicos com as propriedades mais elementares da especificação.

4.2 Esquemas

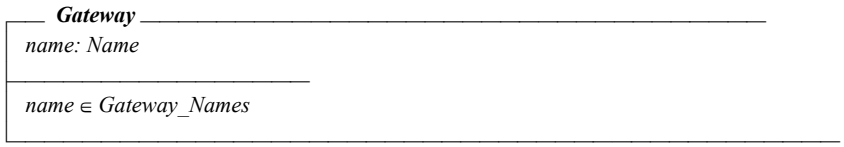
Na Notação Z os esquemas são utilizados para agrupar determinadas especificações. Esta estrutura possibilita que elas possam ser reutilizadas no decorrer da especificação. Um esquema é composto por duas parte: declarativa e predicativa. A parte declarativa é utilizada para que sejam descritas todas as declarações de tipos e variáveis. Já a parte predicativa é o conjunto de todas as restrições referentes ao conjunto de declarações.

4.2.1 *Gateway*

A relação entre *Task* e *Slot* ocorre a partir de seus pontos de ligação. A representação no metamodelo que representa a sintaxe abstrata da tecnologia Guaraná atribui propriedades para as duas extremidades do relacionamento.

Assim, *input* e *output* são propriedades de *Slot* do tipo *Task*. Já *relatedInput* e *relatedOutput* são propriedades de *Task* e são do tipo *Slot*. Não é possível fazer esta representação com a Notação Z de forma direta. Como a Notação Z está baseada na teoria dos conjuntos, esta representação é equivalente, por exemplo, a dois conjuntos A e B, onde A está contido em B e B está contido em A, sendo, no entanto, A e B são conjuntos distintos, o que, segundo essa teoria, não é possível. Assim, para representar este tipo de relação foi utilizado um esquema definido como *Gateway*, apenas com a propriedade nome para a identificação, como ilustra a Figura 3

Figura 3. Esquema *Gateway*



Fonte: Elaborada pelos autores.

Os quatro conjuntos do tipo *Gateway* foram definidos como "*relatedInputs, relatedOutputs, inputs, outputs*: $\mathbb{P}$ Gateway"

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

4.2.2 *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 põe em prática as atividades que devem ser realizadas pela tarefa. As entradas e saídas são conectadas aos *slots* em tempo de execução e contêm mensagens.

O esquema *Task* apresentado na Figura 4 define as respectivas restrições. A parte declarativa descreve *name* do tipo *Name, executoinBody* do tipo *Text*, *taskGateway, inputs e outputs* do tipo *Gateway, inputSlots* e *outputSlots*. Estes

Figura 4. Esquema *Task*

Task

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

$name \in Tasks\_Names$	*(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)*

Fonte: Elaborada pelos autores.

dois últimos são utilizados para definir a relação entre as entradas e saídas da tarefas com entradas e saídas de *Slot*, sendo, nesse contexto, definidos como uma função bijetora.

A linha (1) indica que, para cada instância de *Task*, o seu respectivo nome deve pertencer ao conjunto de *task_names*. Nas linhas (2) e (3) está definido um nome único para o conjunto de *inputs* e *outputs*, garantindo-se um nome único dentro do conjunto de nomes de entradas e saídas da 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*, em que, 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*. 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*.

4.2.3 *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*. Já o conjunto de saídas de todas as tarefas deve ser igual ao conjunto de *relatedOutput* de *outputSlots*. Assim, tem-se a garantia de que cada entrada ou saída está ligada a um, e apenas um, *slot*.

Figura 5. Esquema *Slot*

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)

Fonte: Elaborada pelos autores.

Para estabelecer as restrições para o conjunto de *Slots* que compõe uma solução de integração, foram declaradas as variantes, conforme ilustra a Figura 5: *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 um nome único para as entradas e saídas de *Slot*. Na linha (4) define-se que *target* não pode ser igual a *source*, ou seja, a entrada de um *Slot* não pode ser ao mesmo tempo a saída do *Slot*. As linhas (5) e (6) da especificação definem que, para toda instância de *relatedInputs*, deve-se, 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*.

4.2.4 *Port*

Portas são compostas de *tarefas* e *slots* e são conectadas entre si por meio de *links*. Os *links* podem ser de dois tipos: *ApplicationLinks* e *IntegrationLinks*. Caso façam a ligação entre aplicações e portas, pertencem ao tipo *ApplicationLinks*. Se, no entanto, realizam ligação entre uma porta de entrada de um processo e uma porta de saída de outro processo pertencem ao tipo *IntegrationLinks*. 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*, conforme a Figura 6, 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*. As linhas (2) e (3) estabelecem as restrições de *slots* dentro do esquema *Port*. A partir dela é realizada a união dos conjuntos de *inputSlots* e *outputSlots* e atribuído para *slots*. As tarefas que compõem o conjunto de tarefas das portas seguem o critério de nome exclusivo. Isto está definido na linha (4). E, por fim, atribui-se a cardinalidade para *tasks* como mostra 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 propriedades adicionais. Uma *EntryPort* possui tarefas específicas, definidas como *Comunicator* e *InComunicator*. A *ExitPort* também possui tarefas específicas, tais como *Comunicator* e *OutComunicator*, conforme ilustra a Figura 6.

Figura 6. Esquema *Port*

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)

EntryPort

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

ExitPort

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

Fonte: Elaborada pelos autores.

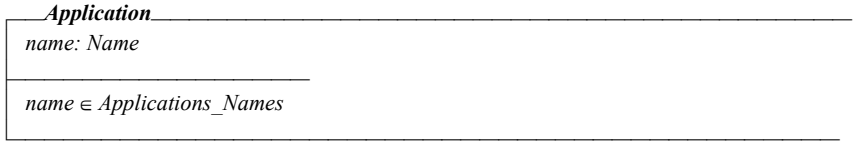
4.2.5 *Application*

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*, como ilustra a Figura 7.

4.2.6 Link

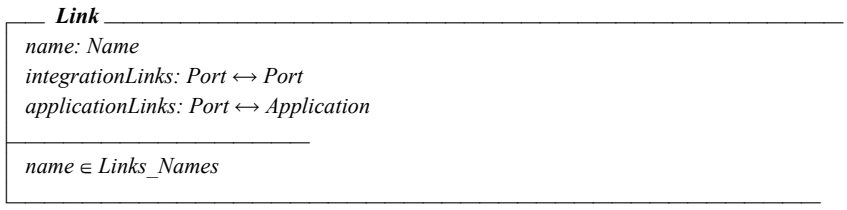
O esquema *Link* define *name* do tipo *Name*, atribuído-lhe a restrição de que deve pertencer ao conjunto de *Link_Names*. Além disso, o esquema *Link* é composto por *integrationLinks* e *applicationLinks*, conforme ilustra a Figura 8. O primeiro compõe a relação entre uma porta de entrada e uma porta de saída de um processo, ao passo que *applicationLink* conecta uma aplicação a um processo através de uma porta.

Figura 7. Esquema *Application*



Fonte: Elaborada pelos autores.

Figura 8. Esquema *Link*



Fonte: Elaborada pelos autores.

4.2.7 *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*, conforme ilustra a Figura 9.

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 às tarefas de *Process*. A linha (2) atribui à variante *tasks* a união dos conjuntos de tarefas próprias do processo com as tarefas das portas, ou seja, *tasks* é formado por *selfTasks* união *entryPortTasks* união *exitPortTasks*. A linha (3) 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

Figura 9. Esquema *Process*

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)
$selfTasks \cap (entryPortTasks \cup exitPortTasks) = \varnothing$ (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)
$ports = entryPorts \cup exitPorts \land entryPorts \cap exitPorts = \varnothing$ (7)
$\forall\ p1, p2: Port \bullet\ p1.name = p2.name \Leftrightarrow p1 = p2$ (8)
$\forall\ s: slots; t1, t2: Task$ (9)
$\quad \bullet\ (t1, t2) \in s.slots$
$\quad\quad \Rightarrow t1 \in selfTasks \land t2 \in selfTasks \land t1 \neq t2 \lor (t1, t2) \in s.interslots$
$\quad\quad\quad \Rightarrow t1 \in selfTasks \land t2 \in entryPortTasks \cup exitPortTasks \land t1 \neq t2$
$\#\ interslots = \#\ entryPorts + \#\ exitPorts$ (10)
$\#\ entryPorts \geqslant 1$ (11)
$\#\ exitPorts \geqslant 1$ (12)
$\#\ selfTasks \geqslant 1$ (13)
$\#\ slots \geqslant 2$ (14)

Fonte: Elaborada pelos autores.

de *Process*. As linhas (5) e (6) definem que o conjunto de *interslots* deve estar contido no conjunto *slots* e sua identificação, por meio de *name*, exige que ele seja exclusivo. 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 um nome exclusivo para todo o conjunto de portas. A linha (9) refere-se às restrições para *slots* e *interslots*. *Slots* relacionam duas tarefas que devem, obrigatoriamente, pertencer ao conjunto de tarefas próprias do processo. Já *interslots* relacionam duas tarefas, em que uma

delas deve pertencer ao conjunto de tarefas próprias do processo e a outra deve pertencer ao conjunto formado pela união das tarefas das portas de entrada com as tarefas das portas de saída. Por meio desta restrição para *interslots* fica estabelecido que o relacionamento entre a tarefa do processo com a tarefa de uma porta. Com isso, tem-se exatamente um *interslot* para cada porta. Assim, 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 compõem um processo.

4.2.8 *Solution*

No metamodelo apresentado na Figura 2, a classe raiz é chamada de Solution e 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*. Deste modo, no esquema *Solution*, tem-se a identificação por meio de *name* do tipo *Name*; *applications* do tipo *Applications*, *processes* do tipo *Process* e *integrationLinks, applicationLinks, links* do tipo *links*, conforme ilustra a Figura 10.

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 *integrationLinks* e *applicationlinks* e lhe atribui um nome exclusivo de acordo com a especificação da linha (3). As linhas (4) e (5) definem a restrição de nome exclusivo para todas as instâncias de *Application* e *Process*, respectivamente. A linha (6) define que a relação entre uma porta de saída de um processo e uma porta de entrada de outro processo está expressa pelo *integrationLink*. Já a relação entre uma porta de um processo e uma aplicação é expressa por um *applicationLink*. A restrição da linha (7) determina que *source* e *target*, elementos de *slot*, pertencem ao conjunto de tarefas de um processo. As restrições (8), (9) e (10) definem a cardinalidade para o esquema *Solution*.

Figura 10. Esquema *Solution*

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)*
$application: applications$
$\bullet\ (entryPort, exitPort) \in l1.integrationLinks$
$\Rightarrow entryPort \in p1.entryPorts \land exitPort \in p2.exitPorts \land p1 \neq p2$
$\lor\ (port, application) \in l1.applicationLinks$
$\forall\ sourceProcess, targetProcess: processes;\ s: Slot$ *(7)*
$\bullet\ s.source \in sourceProcess.tasks \land s.target \in targetProcess.tasks$
$\#\ processes \geqslant 1$ *(8)*
$\#\ applications \geqslant 1$ *(9)*
$\#\ integrationLinks \geqslant 1$ *(10)*

Fonte: Elaborada pelos autores.

5 Validação

Com as ferramentas de apoio às linguagens formais é possível realizar a prova das especificações de forma automatizada. Esta forma de validação é uma prática amplamente utilizada na área de formalização de sistemas e linguagens. Os trabalhos apresentados por [23, 24, 25, 26], por exemplo, utilizam a prova automatizada por meio da ferramenta Z/Eves com o objetivo de provar suas especificações. Além disso, a complexidade de uma prova completa torna imprescindível a utilização da verificação automatizada para garantir que todas as regras da especificação estejam corretamente desenvolvidas. É impraticável

a prova de uma especificação de forma manual [27] .

Desta forma, a validação da especificação formal da sintaxe abstrata da tecnologia Guaraná foi realizada de forma automatizada por meio da ferramenta Z/Eves. Esta ferramenta é capaz de fazer a verificação de sintaxe e de tipo, além de gerar os teoremas necessários de forma automática sobre especificação e, com isso, permitir a prova de suas propriedades.

Foram analisados os 16 parágrafos correspondentes à especificação, os quais estão divididos em 4 parágrafos simples e 12 esquemas. Os esquemas dividem-se 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 modo totalmente transparente para o desenvolvedor. Já a validação dos esquemas é realizada por meio da análise de teoremas. Os teoremas são gerados e exibidos pela ferramenta após a execução da validação, possibilitando a análise por parte do desenvolvedor.

Inicialmente o Z/Eves traduz automaticamente as especificações realizadas pela Notação Z em predicados de primeira ordem, tornando-os manipuláveis e de fácil entendimento. A partir desta tradução, todas as manipulações lógicas são realizadas.

A verificação da sintaxe e de tipos é realizada de forma automática pela ferramenta, logo após a leitura da especificação. Com isso, é garantido que os tipos dos operandos e operadores estejam em concordância e que a especificação pertença a uma linguagem gerada pela gramática aceita pela ferramenta. Em relação a prova de teoremas, o Z/Eves gera um conjunto específico de teoremas para cada um dos esquemas da especificação rotulados como: *grule, frule, rule e axiom*, dependendo do tipo de prova [23].

Para confirmar a prova foi necessário seguir as etapas de verificação de sintaxe e de tipos, checagem de domínio e verificação de inconsistências. Na verificação de sintaxe e de tipos, considerou-se a complexidade da notação. As-

sim, percebeu-se que a verificação, nessa etapa, necessitava de uma ferramenta de apoio para evitar erros na escrita da especificação. Assim, a verificação de sintaxe e de tipos foi realizada de forma automática pela ferramenta logo após a leitura do parágrafo. Isso garantiu que os tipos dos operandos e operadores estivessem em concordância e que a especificação pertencesse à linguagem que foi gerada pela gramática. Por exemplo, se a atribuição de igualdade escrita por meio de ":=" não é válida pelo provador, gera-se um erro de sintaxe, o qual realiza a atribuição utilizando "==". Na chegagem de domínio, verificou-se quais foram as funções parciais aplicadas para valores definidos no seu domínio. Por exemplo, se uma restrição não for satisfeita, significa que ela não apresenta nenhuma possibilidade válida em seu domínio. Já a ocorrência da verificação de inconsistências foi realizada sempre que a especificação não apresentava nenhum modelo válido. A inconsistência, nesse caso, poderia ser global, com seu efeito propagado por toda a especificação, ou local, quando restrito a uma definição predicativa de um determinado esquema.

Além dessas três etapas, pode-se considerar a etapa de verificação de estado inicial, que garante que existe ao menos um estado que satisfaça às invariantes do sistema e o cálculo de précondições, que define o conjunto de estados iniciais e estados intermediários, que define a existência de um estado posterior.

Para os 12 esquemas da especificação formal da tecnologia Guaraná foram gerados 104 teoremas totalizando mais de 300 linhas de código. De acordo com a Tabela 1, do total de teoremas, 52 são rotulados como sendo do tipo *Grule*, 39 do tipo *rule*, 10 do tipo *frule* e 3 do tipo *axiom*.

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.

Tabela 1. Estatística sobre os teoremas gerados e provados

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

Fonte: Elaborada pelos autores.

6 Conclusões e Trabalhos Futuros

Este capítulo apresentou a especificação formal da sintaxe abstrata da linguagem de domínio específico da tecnologia Guaraná. Cada propriedade definida no metamodelo UML para o Guaraná DSL teve sua especificação formal correspondente estabelecida. Verificou-se a necessidade de fornecer a declaração de tipos adicionais. Como uma operação de relação ou função entre dois conjuntos deve formar um elemento de um novo conjunto, não é possível associar este elemento a um dos dois conjuntos participantes desta operação.

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 uma etapa do processo de formalização da tecnologia Guaraná atribuindo maior rigor e clareza a sintaxe abstrata da linguagem por meio da Notação Z. Esta especificação formal serve como base para as etapas de formalização da sintaxe concreta e da semântica. Além disso é

possível fazer o refinamento da especificação formal da sintaxe abstrata fornecida. Com o refinamento pode-se tornar ainda mais legível a especificação formal por meio da simplificação de alguma propriedade. Uma vez concluído todo o processo de formalização espera-se, por meio dele, tornar possível validar ou gerar automaticamente o conjunto de regras para as soluções de integração implementadas com a tecnologia Guaraná.

Agradecimentos

Os autores agradecem à CAPES (Coordenação de Aperfeiçoamento de Pessoal de Ensino Superior) e ao CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico).

Referências

[1] D. G. Messerschmitt and C. Szyperski, *Software ecosystem - Understanding an Indispensable Technology and Industry*. The MIT Press, London, 2003.

[2] G. Hohpe and B. Woolf, *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley Professional, Boston, 2003.

[3] R. Z. Frantz, *Enterprise Application Integration: An Easy-to-maintain Model-Driven Engineering Approach*. University of Seville, Seville, 2012.

[4] I. Hernández, S. Sawicki, F. Roos-Frantz, and R. Z. Frantz, "Cloud configuration modelling: A literature review from an application integration deployment perspective," *Procedia Computer Science*, vol. 64, pp. 977–983, 2015.

[5] C. Ibsen and J. Anstey, *Camel in action*. Manning Publications Co., Shelter Island, NY, 2010.

[6] M. Fisher, J. Partner, M. Bogoevici, and I. Fuld, *Spring Integration in action*. Manning Publications Co., Shelter Island, NY, 2012.

[7] D. Dossot, J. D'Emic, and V. Romero, *Mule in action*. Manning Publication Co., Shelter Island, NY, 2009.

[8] R. Z. Frantz, A. M. R. Quintero, and R. Corchuelo, "A domain-specific language to design enterprise application integration solutions," *International Journal of Cooperative Information Systems*, vol. 20, no. 02, pp. 143–176, 2011.

[9] J. B. Goodenough, "Exception handling: issues and a proposed notation," *Communications of the ACM*, vol. 18, no. 12, pp. 683–696, 1975.

[10] A. M. Mostafa, M. A. Ismail, H. El-Bolok, and E. M. Saad, "Toward a formalization of UML 2.0 metamodel using Z specifications," in *Proceedings of the 8th International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*, 2007, pp. 694–701.

[11] M. Shroff and R. B. France, "Towards a formalization of UML class structures in Z," in *Proceedings of the 21st Annual International Computer Software and Applications Conference*, 1997, pp. 646–651.

[12] S.-K. Kim and C. David, "Formalizing the UML class diagram using Object-Z," in *≪UML≫'99 – The Unified Modeling Language*, 1999, pp. 83–98.

[13] M. Richters and M. Gogolla, "On formalizing the UML object constraint language OCL," in *International Conference on Conceptual Modeling*, 1998, pp. 449–464.

[14] A. G. Kleppe and J. B. Warmer, *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Professional, 2003.

[15] D. Roe, K. Broda, and A. Russo, *Mapping UML models incorporating OCL constraints into Object-Z*. Imperial College of Science, Technology and Medicine, Department of Computing, London, UK, 2003.

[16] T. Jiang and X. Wang, "Formalizing domain-specific metamodeling language XMML based on first-order logic," *Journal of Software*, vol. 7, no. 6, pp. 1321–1328, 2012.

[17] E. Jackson and J. Sztipanovits, "Formalizing the structural semantics of domain-specific modeling languages," *Software & Systems Modeling*, vol. 8, no. 4, pp. 451–478, 2009.

[18] R. Z. Frantz, R. Corchuelo, and F. Roos-Frantz, "On the design of a maintainable software development kit to implement integration solutions," *Journal of Systems and Software*, vol. 111, pp. 89–104, 2016.

[19] J. M. Spivey, *The Z notation: a reference manual. International Series in Computer Science*. Prentice-Hall, Upper Saddle River, New Jersey, 1992.

[20] S. Getir, M. Challenger, and G. Kardas, "The formal semantics of a domain-specific modeling language for semantic web enabled multi-agent systems," *International Journal of Cooperative Information Systems*, vol. 23, no. 03, 2014.

[21] A. V. Moura, *Especificações em Z: uma introdução*. Editora Unicamp, 2001.

[22] A. K. Dwivedi and S. K. Rath, "Model to specify real time system using Z and alloy languages: A comparative approach," in *International Conference on Software Engineering and Mobile Application Modelling and Development*, 2012, pp. 1–6.

[23] L. Freitas and J. Woodcock, "Mechanising mondex with Z/Eves," *Formal Aspects of Computing*, vol. 20, no. 1, pp. 117–139, 2008.

[24] J. Sun, H. Zhang, Y. Fang, and H. Wang, "Formal semantics and verification for feature modeling," in *Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems*, 2005, pp. 303–312.

[25] A. Regayeg, A. H. Kacem, and M. Jmaiel, "Specification and design of multi-agent applications using temporal Z," in *Intelligent Agents and Multi-Agent Systems*, 2005, pp. 228–242.

[26] S. Tarkan, "The formal specification of a kitchen environment," University of Maryland, College Park, Maryland," Master's scholarly paper, 2009.

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