Criando uma Linguagem Específica de Domínio usando o Xtext: Uma Opção *Open Source* para Implementar a Canopus

Aníbal I. Neto, Luiz P. Franz, Jonnathan R. Lopes,
Elder Rodrigues, Maicon Bernardino

[1]Universidade Federal do Pampa (UNIPAMPA)
Código Postal 97.546-550 – Alegrete – RS – Brasil

{netoiiung, luizpaulofranz, jonnathan.riquelmo}@gmail.com
elderrodrigues@unipampa.edu.br, bernardino@acm.org

Abstract. *Domain-Specific Language (DSL) is the name given to languages used in specific aspects of a system. Implementing a DSL is not a trivial task, since they are programming languages and have a syntax that is defined by a grammar. In this way, it is necessary to use tools that support the definition of concepts for the new language. Tools with this ability are popularly known as Language Workbenches. This article presents a guide to assist on the development of a DSL using the Xtext Eclipse framework and aims to present an alternative to the definition of the textual notation and development of the Canopus DSL.*

Resumo. *Linguagem Específica de Domínio (Domain-Specific Language- DSL) é o nome dado a linguagens utilizadas em aspectos específicos de um sistema. O desenvolvimento de uma DSL não é tarefa trivial, pois são linguagens de programação e possuem uma sintaxe que é definida por uma gramática. Desta forma, faz-se necessário a utilização de ferramentas que suportem a definição dos conceitos para a nova linguagem. Ferramentas com essa capacidade são popularmente conhecidas como Language Workbenches. Este artigo apresenta um guia para auxiliar o desenvolvimento de uma DSL utilizando o framework Eclipse Xtext e tem como objetivo apresentar uma alternativa para a definição da notação textual e desenvolvimento da DSL Canopus.*

1. Introdução

As Linguagens Específicas de Domínio (*Domain-Specific Languages*-DSL), são linguagens de programação que exercem um papel de auxílio fornecendo maior nível de abstração para problemas específicos. Essas linguagens visam modelar e descrever um domínio ou uma solução distinta e tem diversas aplicações. A adoção de uma DSL pode trazer vários ganhos, como geração automática de código e reuso [Fowler 2010].

Um grande tema da Engenharia de Software em que a utilização de uma DSL pode auxiliar é a área de testes de software. É de senso comum que um software, quando testado de forma adequada, atinge um bom nível de qualidade. Existem diversas técnicas e estratégias de teste de software, que podem variar com o nível da aplicação testada e com os objetivos do teste, *e.g.* teste de desempenho. Uma solução amplamente adotada diz respeito aos testes automatizados. Um conjunto de testes automatizados possibilita,

hipoteticamente, a oportunidade de um software ser testado com maior frequência e rapidez em diversos aspectos [Ammann and Offutt 2008]. Entretanto, pode se tornar difícil manter um conjunto de testes automatizados. À medida que um software muda, ou ainda mesmo durante o seu desenvolvimento, é necessário que os testes sejam adaptados.

Nesta perspectiva é possível afirmar que a qualidade dos testes pode ficar mais frágil quando estes estão no nível de interface gráfica do usuário. Isso ocorre pois geralmente esses testes são criados por meio de ferramentas que utilizam a técnica de Captura e Repetição (*Capture and Replay* - CR). Nesta técnica o engenheiro deve executar os passos dos testes manualmente no software uma vez usando alguma ferramenta que dê apoio à captura das interações e, em seguida, executar um modo de repetição para simular as ações previamente realizadas. Estas ferramentas geram *scripts*, porém estes testes produzidos são difíceis de serem modificados ou atualizados, tornando assim necessário, muitas vezes, a reexecução do processo da técnica CR.

Uma alternativa a técnica CR são os Testes Baseados em Modelos, do inglês *Model-Based Testing* (MBT). Algumas vantagens que o MBT pode ocasionar são a facilidade de compreensão, o maior nível de reuso e geração de artefatos de teste. É possível afirmar também que a preocupação com o projeto de modelos de teste, desde estágios iniciais de desenvolvimento, é maior utilizando essa abordagem [Kramer and Legeard 2016]. Na abordagem MBT, o engenheiro projeta os modelos de teste, aponta informações de desempenho e, logo após, usa uma ferramenta de apoio para gerar automaticamente um conjunto de *scripts* e cenários de teste. Existem algumas notações, linguagens e modelos que podem ser aplicados para o teste de desempenho, *e.g.* *Unified Modeling Language* (UML), *User Community Modeling Language* (UCML) e a *Customer Behavior Modeling Graph* (CBMG).

O objetivo do presente artigo é apresentar uma alternativa para a definição e desenvolvimento do metamodelo da Canopus [Bernardino et al. 2016b], que é uma DSL que especifica e automatiza o processo de teste de desempenho para aplicações Web. No que tange o uso de uma DSL para auxiliar o processo de teste de desempenho de aplicações Web, várias soluções com esse propósito já foram propostas na literatura. Especialmente no contexto da avaliação de desempenho de aplicações na nuvem pode-se citar o caso dos ambientes Expertus [Jayasinghe et al. 2012], Cloudbench [Silva et al. 2013], Cloud Crawler [Cunha et al. 2013], e Cloud Work Bench [Scheuner et al. 2014]. Além disso, a automação das atividades de teste de desempenho é uma das práticas DevOps[1] que têm sido cada vez mais utilizadas pela indústria, a qual pode ser realizada com o apoio de ferramentas abertas, independentes de qualquer tecnologia proprietária, como Gatling [GatlingCorp 2017] e Locust [Heyman et al. 2017].

A Canopus foi proposta em 2016, utilizando o MetaEdit+ Workbench [MetaCase 2017] como ferramenta de metamodelagem. A ferramenta usada provê suporte à definição de DSL, sendo importante salientar que é um instrumento proprietário. A alternativa apresentada nesta proposta é o *framework* Xtext, uma opção *open source* que possibilita a definição de DSL de forma textual e oferece outras funções, entre elas a geração de código-fonte.

[1]DevOps: termo provêm da composição de "desenvolvimento" e "operações". É um processo de desenvolvimento e entrega de software que enfatiza comunicação e colaboração entre profissionais de gerenciamento de produtos, desenvolvimento de software e operações.

Este artigo está organizado da seguinte forma. A Seção 2 apresenta uma contextualização sobre Linguagem Específica de Domínio, bem como ferramentas que fornecem suporte a criação de DSL. Prosseguindo, na Seção 3 é demonstrado uma ideia geral do processo de implementação de uma DSL no Xtext. A Seção 4 apresenta os trabalhos relacionados. Por fim, a Seção 5 descreve a conclusão e trabalhos futuros.

2. Linguagem Específica de Domínio

Uma linguagem de programação possui uma sintaxe, que é definida por sua gramática. E gramática segundo [Russell and Norvig 2004] é composta por definições matemáticas rígidas, que definem a ordem dos lexemas da linguagem, ou seja, a ordem esperada das "palavras", enquanto que a semântica de uma linguagem é o significado de uma expressão válida nessa linguagem [Fowler 2010]. As linguagens de programação são classificadas em dois grupos: Linguagens de Propósito Geral (*General Purpose Languages* - GPL), que são linguagens de programação de uso geral, tais como: JAVA, C, C# e Python, e; Linguagens Específicas de Domínio (*Domain-Specific Languages* - DSL) [Fowler 2010] são linguagens de programação voltadas para um aspecto específico de um sistema, em que não podem ser utilizadas para o desenvolvimento de sistemas inteiros, pois o que define uma DSL é a expressividade limitada e o foco no domínio. As DSL são amplamente utilizadas no desenvolvimento de sistemas, são exemplos de DSL: SQL, HTML.

Existem duas classificações de DSL [Fowler 2010]: (i) DSL Interna: uma forma específica de escrever uma linguagem de propósito geral (*General Purpose Language* - GPL). É uma linguagem que utiliza a estrutura de uma GPL como base, também conhecida como Linguagem Host; (ii) DSL Externa: uma linguagem separada de uma GPL, possui sintaxe e semântica própria e precisa de um compilador/interpretador próprio para ser compilada/interpretada [Fowler 2010]. Uma DSL fornece um meio para comunicar e expressar mais claramente a definição de uma parte/todo do sistema, e tornar mais fácil a realização de determinadas tarefas, tais como: ler um trecho de código, encontrar erros e, modificar o sistema.

2.1. Ferramentas de Suporte a Criação de DSL

Tanto na indústria como na academia, o surgimento de ferramentas de suporte para DSL cresce consideravelmente. Ferramentas que dão suporte a uma ou várias etapas do processo de desenvolvimento de DSL são classificadas como frameworks, IDEs e as chamadas *Language Workbenchs* (LW), que segundo Fowler [Fowler 2010], são IDEs especializadas em desenvolvimento de DSL. Bons exemplos desse tipo de ferramentas são: **MPS**: LW desenvolvida pela JetBrains que suporta a criação de DSL textual [JetBrains 2017]; **MetaEdit+**: LW com suporte para notação gráfica, desenvolvida pela MetaCase [MetaCase 2017]. **EMFText**: Plugin do Eclipse que dá suporte a definição de sintaxe de linguagens [Wende et al. 2017]; **Whole**: LW baseada no Eclipse que suporta a criação de DSL textuais e gráficas [Solmi 2017]; **Xtext**: *framework* Eclipse que apoia a criação de linguagens textuais [Eysholdt and Behrens 2010], o qual é o foco deste estudo.

2.2. Xtext

É um framework voltado para o desenvolvimento de linguagens de propósito geral e DSLs [Eysholdt and Behrens 2010]. Esse framework permite um rápido desenvolvimento de ferramentas que dão suporte a criação de novas linguagens, como suporte a

implementação da gramática, gerador de compiladores e analisadores léxicos e sintáticos. O Xtext faz parte do grupo de ferramentas voltadas para a criação de linguagens do Eclipse (*Eclipse DSL Tools*), possui licença *open source* e cobre grande parte dos requisitos comuns no desenvolvimento de linguagens textuais.

3. Desenvolvendo um Exemplo de DSL com o Xtext

Ao criar um projeto de DSL no Xtext, algumas configurações são necessárias. Inicialmente, é definido um nome e qual a extensão dos arquivos que vai conter a DSL criada. Na sequência, um *wizard* oferece opções para a criação do produto (IDE) que dará suporte a nova DSL, o Xtext oferece a possibilidade de criação *plugins* para as IDE *Eclipse* e *IntelliJ*. Por padrão, no momento da criação, o Xtext gera três projetos com seus nomes baseados no atributo *project name*. O primeiro projeto, com o nome que foi determinado na configuração do *wizard*, é o principal. Nele é definida a gramática, os validadores e gerador automático de código. Todos os demais projetos são dependentes deste. Para DSL simples, esse é o único projeto requerido para edição de código [Bettini 2016].

O projeto que finaliza com ".ide" é um projeto que contém os recursos para a implementação da IDE para a nova DSL. Esse projeto é gerenciado pelo próprio Xtext. Não é recomendado nenhum tipo de alteração, pois esse projeto contém a implementação dos *parsers* gerados a partir da definição da gramática. O projeto com extensão ".ui" é o responsável pela integração da nova DSL com a IDE *Eclipse*. Esse é o projeto que se torna um produto Eclipse, portanto, informações ou funcionalidades adicionais devem ser implementadas neste projeto, dependendo dos objetivos. Porém nesse cenário, o mais recomendado é o desenvolvimento de um *plugin* externo a esses projetos, seguindo o princípio de separação de responsabilidades. Nesta última abordagem, é possível importar as novas funcionalidades nos outros *plugins* ou produtos Eclipse.

Quando a opção *Testing Support* é marcada no *wizard*, são gerados dois projetos extras para fins de testes. O projeto com extensão ".tests", é usado para testar a nova DSL, em que um trecho da nova DSL é a entrada, comparando com a saída gerada. Já o projeto com a extensão ".ui.tests", é usado para testar o editor de código, *i.e.* a parte visual do produto Eclipse. Apresentados os projetos iniciais gerados, o artefato que merece mais atenção é o arquivo com a extensão ".xtext", localizado no projeto principal. É o arquivo em que é especificado a gramática da DSL a ser desenvolvida com a *Grammar Language*.

A *Grammar Language* apresenta uma sintaxe baseada em gramáticas, portanto, segue o conceito de regras. Cada regra da gramática da nova DSL deve ter: (i) um nome ou identificador válido, (ii) seguido do caractere ":", e (iii), na sequência as definições dessa regra. Essas definições podem conter outras regras, palavras reservadas e operadores. Estas regras são separadas em **regras terminais** e **regras não-terminais**. Regras terminais definem o léxico da linguagem por meio de expressões regulares e contém as palavras (ou *tokens*). As regras não-terminais por usa vez podem ser composições de outras regras terminais e não-terminais, além de poder conter palavras reservadas. As regras não-terminais são as responsáveis por definir a ordem dos lexemas e, por fim, definir a sintaxe da linguagem. O Xtext oferece um conjunto pré-programado com regras terminais, como identificadores válidos para variáveis, definições de comentários e *strings*. Esse conjunto é chamado de *Terminals*, e todo projeto de linguagem do Xtext já vem com esse recurso importado. O Xtext permite também a importação de outras linguagens definidas

por terceiros. O recurso que permite a mescla de DSL é chamado de *mix-in* de linguagens. A Figura 1 apresenta mais detalhes de como é uma *Grammar Language*.

```
1  grammar org.xtext.example.DomainModel with
2  org.eclipse.xtext.common.Terminals
3  generate domainModel "http://www.xtext.org/example/DomainModel"
4
5  Domainmodel: (elements+=Type)*;
6
7  Type: DataType | Entity;
8
9  DataType: 'datatype' name=ID;
10
11 Entity:
12     'entity' name=ID ('extends' superType=[Entity])? '{'
13         (features+=Feature)*
14     '}';
15
16 Feature: (many?='many')? name=ID ':' type=[Type];
```

Figura 1. *Grammar Language*

```
1  datatype String
2  entity Blog {
3      title: String
4      many posts: Post }
5
6  entity HasAuthor { author: String}
7
8  entity Post extends HasAuthor {
9      title: String
10     content: String }
```

Figura 2. Linguagem com Xtext

Nas duas primeiras linhas da Figura 1 ocorre o *mix-in* de linguagens, importando as regras terminais padrão. Em seguida a palavra reservada *generate* fornece instruções ao núcleo EMF, além de definir o nome da DSL proposta [Bettini 2016]. A primeira regra propriamente da DSL costuma ser bem genérica, é a raiz da gramática, ela define por onde o *parser* vai iniciar e é conhecida como regra inicial (linha 5 na Figura 1), [Bettini 2016]. A regra inicial neste exemplo é a regra *DomainModel*. Ela tem uma propriedade ou atributo (ou ainda variável) chamada *elements*. Esta propriedade contém uma coleção de ocorrências da regra *Type*. Os caracteres "(...)*" indicam que *Type* pode ocorrer zero ou muitas vezes. Essa coleção será armazenada na propriedade *elements*. Quando trata-se de coleções, o operador de atribuição "+=" é necessário para indicar que *elements* é uma coleção. Caso seja utilizado o operador de atribuição "=" apenas a última ocorrência de *Type* seria armazenada.

Na linha 7 da Figura 1 a regra *Type* é definida. Ela apenas informa que um *Type* pode ser um *DataType* **ou** *Entity*. A definição da regra *DataType* é simples, pois apenas apresenta a palavra reservada "datatype" (essa é a forma como palavras reservadas são definidas com *grammar-language*) e uma propriedade *name* que deve receber um identificador válido, definido na regra padrão ID, oferecida pelo Xtext.

A primeira orientação que a regra *Entity* define na linha onze é sua palavra reservada "entity". Após, a regra espera um identificador para a entidade em questão, que será atribuída à propriedade *name*. Uma entidade pode ou não estender outra entidade. A palavra reservada "extends" está contida em um parênteses, e esse parênteses é seguido do operador "?" que indica opcionalidade. É importante enfatizar o trecho "superType=[Entity]", onde os colchetes são usados para realizar a chamada **referência cruzada**. Este é um dos recursos mais poderosos do Xtext. Isso significa que a propriedade *superType* só receberá uma entidade já declarada. Isso se justifica no paradigma de orientação a objetos, em que é preciso declarar a classe mãe antes das classes filhas. O uso dos colchetes informa ao *parser* que nesse ponto é esperado o nome de uma *Entity* já declarada. Para a referência cruzada funcionar, a regra deve possuir a propriedade *name*.

A regra *Entity* ainda define que uma entidade pode conter uma lista de *Features* (linha trêze) e que essa lista pode ser vazia, por meio do operador "*" (zero ou muitos). Por fim, a regra *Feature*, linha dezesseis, inicia com a opcionalidade na palavra reservada "many". O operador de atribuição "?=", indica que essa é uma propriedade *booleana*,

portanto, quando houver a **palavra reservada** "many" no código de uma *Feature*, a **propriedade** *many* vai receber *true*, indicando que se trata de um *array*. Após isso, a regra espera um nome seguido da palavra reservada ":". Então, é indicado o tipo dessa *Feature*, utilizando novamente referência cruzada.

Essa DSL contém as seguintes palavras reservadas: "datatype", "entity", "extends", "{", "}", ":" e "many". Um trecho de código que respeita essa gramática é apresentado na Figura 2. Ela traz o *highlight* das palavras reservadas. O Xtext ainda gera uma série de artefatos, inclusive os *parsers* a partir da gramática descrita. Com a gramática definida e os artefatos criados é possível testar a DSL em um editor *Eclipse* por meio da opção "Eclipse Application".

As análises léxicas, sintáticas e ferramental de apoio da IDE são gerados automaticamente a partir da gramática. Com isso, obtêm-se um editor funcional para a DSL. Porém, é importante notar que a geração de códigos precisa ser implementada. Após a geração dos artefatos, um pacote "**generator**" é criado no projeto principal. Esse pacote já contém um esboço do gerador de código, e é nele onde são codificadas as regras para a geração de código e persistência em arquivos, o que seria a "compilação" de uma DSL. Além do pacote *generator*, outro pacote importante criado é o "**validation**". A corretude de um código não pode ser determinado unicamente pelas análises léxicas e sintáticas, sendo necessário muitas vezes uma etapa extra de **análise semântica**. No Xtext as análises semânticas são implementadas na forma de *validators* [Bettini 2016].

O Xtext ainda cria dois projetos separados para os testes. Um é responsável pelo *parser* e *generator* enquanto o outro testa a interface gráfica. Os testes para os *parsers* são relativamente simples. No exemplo gerado, basicamente, deve ser informado uma entrada (código da nova DSL) e uma saída. Esta pode ser um objeto do EMF ou uma *string* contendo os códigos gerados esperados. Por fim, ainda é possível empacotar o produto como um *plugin* Eclipse, ou Eclipse RCP, criando um produto Eclipse personalizado contendo apenas a nova DSL criada [Bettini 2016].

4. Trabalhos Relacionados

[Arkin and Tekinerdogan 2014] utilizam o Xtext para implementar duas DSL para aplicações paralelas em plataformas de computação paralela. Ambas as DSL criadas são externas, a primeira foi criada para descrever a configuração física e a segunda para descrever os componentes e a construção deles.

[Demirkol et al. 2013] apresentam uma DSL chamada *Semantic Web Enabled Agent Language* (SEA-L). Essa DSL foi criada utilizando o Xtext junto a alguns outros *plugins* de suporte a notação gráfica, tendo em vista o fato do Xtext dar suporte apenas a notação textual, também pertencentes ao grupo Eclipse. A DSL foi criada para diminuir a complexidade do desenvolvimento de agentes de software para sistemas multiagentes. Assim, os autores realizaram um estudo de caso para avaliar na prática os ganhos obtidos com o uso da DSL no desenvolvimento de um sistema multiagente. Nesse caso, trataram de um sistema de trocas eletrônicas baseado em agentes.

[Nakamura et al. 2012] apresenta uma DSL externa para resolver um problema recorrente na área de MSR (*Mining Software Repositories*). A DSL criada foi chamada de *QORAL*, a qual tem por objetivo facilitar o aumento do desempenho de múltiplas

iterações. O autor utilizou questionários com pesquisadores da área, sem explicar a gramática da *QORAL* para avaliar a curva de aprendizado da linguagem e a capacidade de modificar o código de acordo com o entendimento. Os resultados indicaram que a *QORAL* facilitou o entendimento e a fácil modificação do código.

Por fim, [Marand et al. 2015] utilizaram o Xtext para implementar a "parte textual" da DSML4CP, uma DSL voltada para a programação concorrente. Essa DSL foi criada para ser utilizada em um nível mais alto de abstração que o código fonte. Desta forma, como a DSL proposta é, ao mesmo tempo, gráfica e textual, isso facilita a modelagem do domínio, reduzindo complexidades no desenvolvimento desse tipo de sistema.

5. Conclusão

Este artigo resume parte de um levantamento iniciado pelo *Laboratory of Empirical Studies in Software Engineering* (LESSE), um grupo de pesquisa da UNIPAMPA. O grupo busca, entre outros estudos, encontrar meios para desenvolver uma solução *open source* que auxilie no processo de criação de testes de desempenho baseado por modelos. A Canopus foi selecionada como solução padrão. A escolha tem como apoio estudos que apresentam, entre outros resultados, vantagens da DSL em relação à UML quando aplicada para modelagem de testes de desempenho para aplicações Web [Bernardino et al. 2016a]. É importante salientar que mediante investigação não foram encontradas na literatura propostas com as mesmas características. A condução do desenvolvimento da Canopus usada como exemplo de uso para apresentar na prática o Xtext, nos possibilita afirmar que ele é um *framework* que contempla/suporta os requisitos de criação da notação textual da DSL.

Em outra perspectiva da pesquisa do LESSE está em andamento o estudo sobre o *framework* Sirius. O Sirius é um projeto *open source* que fornece meios de criação de metamodelos gráficos. O Sirius, assim como o Xtext, tem compatibilidade com o *Eclipse Modeling Framework* (EMF), um conjunto de recursos do Eclipse para representar modelos e gerar código equivalente. O objetivo final da pesquisa é a integração de ambos os metamodelos da Canopus, convergindo em um *plugin* com os metamodelos textuais e gráficos no Eclipse. Desta forma, o produto final será uma solução *open source* para modelagem bidirecional das notações textual e gráfica para o teste de desempenho, com capacidade de geração de *scripts* e cenários de desempenho para outras ferramentas. Assim, espera-se que esta solução proposta permita viabilizar a adoção por parte da indústria, bem como prover o envolvimento de outros grupos de pesquisa e/ou comunidade para futuras contribuições e evolução do projeto.

Referências

Ammann, P. and Offutt, J. (2008). *Introduction to Software Testing*. Cambridge Press.

Arkin, E. and Tekinerdogan, B. (2014). Domain specific language for deployment of parallel applications on parallel computing platforms. Vienna, Austria.

Bernardino, M., Rodrigues, E. M., and Zorzo, A. F. (2016a). Performance Testing Modeling: An Empirical Evaluation of DSL and UML-based Approaches. In *31st Annual ACM Symposium on Applied Computing*, pages 1660–1665, New York, USA. ACM.

Bernardino, M., Zorzo, A. F., and Rodrigues, E. M. (2016b). Canopus: A domain-specific language for modeling performance testing. In *Int. Conf. on Software Testing, Verification and Validation*, pages 157–167.

Bettini, L. (2016). *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd.

Cunha, M., Mendonca, N., and Sampaio, A. (2013). A declarative environment for automatic performance evaluation in iaas clouds. In *6th Int. Conf. on Cloud Computing*, pages 285–292. IEEE.

Demirkol, S., Challenger, M., Getir, S., Kosar, T., Kardas, G., and Mernik, M. (2013). A DSL for the Development of Software Agents working within a Semantic Web Environment. *Computer Science and Information Systems*, 10(4):1525–1556.

Eysholdt, M. and Behrens, H. (2010). Xtext: Implement your language faster than the quick and dirty way. In *Int. Conf. Companion on Object Oriented Programming Systems Languages and Applications Companion*, pages 307–309, New York, USA. ACM.

Fowler, M. (2010). *Domain-specific languages*. Pearson Education.

GatlingCorp (2017). Gatling. `http://gatling.io/`.

Heyman, J., Byström, C., and Hamrén, J. (2017). Locust. `http://locust.io/`.

Jayasinghe, D., Swint, G., Malkowski, S., Li, J., Wang, Q., Park, J., and Pu, C. (2012). Expertus: A generator approach to automate performance testing in iaas clouds. In *5th Int. Conf. on Cloud Computing*, pages 115–122. IEEE.

JetBrains (2017). MPS - Create your own domain-specific language. `https://www.jetbrains.com/mps/`.

Kramer, A. and Legeard, B. (2016). *Model-Based Testing Essentials - Guide to the ISTQB Certified Model-Based Tester: Foundation Level*. Wiley.

Marand, E. A., Marand, E. A., and Challenger, M. (2015). DSML4CP: a domain-specific modeling language for concurrent programming. *Computer Languages, Systems & Structures*, 44:319–341.

MetaCase (2017). MetaEdit+ Modeler – Supports your modeling language. `http://www.metacase.com/mep/`.

Nakamura, H., Nagano, R., Hisazumi, K., Kamei, Y., Ubayashi, N., and Fukuda, A. (2012). QORAL: An External Domain-Specific Language for Mining Software Repositories. In *4th Int. Workshop on Empirical Software Engineering in Practice*, pages 23–29.

Russell, S. and Norvig, P. (2004). *Inteligência artificial*. CAMPUS - RJ.

Scheuner, J., Leitner, P., Cito, J., and Gall, H. (2014). Cloud Work Bench–Infrastructure-as-Code Based Cloud Benchmarking. In *6th Int. Conf. on Cloud Computing Technology and Science*, pages 246–253. IEEE.

Silva, M., Hines, M. R., Gallo, D., Liu, Q., Ryu, K. D., and Da Silva, D. (2013). Cloudbench: Experiment automation for cloud environments. In *Int. Conf. on Cloud Engineering*, pages 302–311. IEEE.

Solmi, R. (2017). Whole. `https://whole.sourceforge.io/`.

Wende, C., Seifert, M., Heidenreich, F., Karol, S., and Johannes, J. (2017). EMFText. `http://www.emftext.org`.