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

Implementação da linguagem de metaprogramação Meta-CrySL utilizando o framework Xtext

Vinicius Costa e Silva

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientador
Prof. Dr. Rodrigo Bonifácio

Brasília
2020

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

Implementação da linguagem de metaprogramação Meta-CrySL utilizando o framework Xtext

Vinicius Costa e Silva

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof. Dr. Rodrigo Bonifácio (Orientador)
CIC/UnB

Pedro Henrique Teixeira Costa
CIC / UnB

Prof.a Dr.a Edna Dias Canedo
CIC / UnB

Prof. Dr. Marcelo Grandi Mandelli
Coordenadora do Bacharelado em Ciência da Computação

Brasília, 17 de dezembro de 2020

Dedicatória

Dedico esse trabalho aos meus pais, que me apoiaram ao longo de todo o caminho da minha graduação.

Agradecimentos

Agradeço ao meu orientador, Professor Rodrigo Bonifácio, por todo o apoio prestado ao longo de todo o processo de construção desse Trabalho de Graduação, e também a todos os colegas de curso que me auxiliaram de diversas maneiras ao longo dessa trajetória.

Resumo

O presente trabalho visa apresentar a implementação, via o *framework* Xtext, da linguagem de meta-programação Meta-CrySL. Meta-CrySL é uma linguagem utilizada para gerenciar famílias de regras CrySL, as quais por sua vez são uma linguagem de domínio específico utilizada para especificar o uso correto de API's criptográficas. A partir de uma implementação original baseada em Rascal-MPL, a linguagem Meta-CrySL foi implementada dentro do ecossistema *Xtext*, utilizando a linguagem de programação Xtend. Ao final do trabalho, foi avaliado o desempenho da nova implementação tendo como base um conjunto de especificações Meta-CrySL, e sugeridos caminhos para melhorias futuras tendo como base principal mecanismos de validação de corretude das regras geradas.

Palavras-chave: MetaCrySL, linguagens de domínio especifico, trabalho de conclusão de curso

Abstract

The goal of this work is to showcase the design and implementation of the Meta-CrySL meta-programming language, which is used to manage families of CrySL specifications. As CrySL rules are used to specify the correct usage of cryptographic API's, Meta-CrySL emerges as a layer above that to help modularize CrySL rules by using a meta-programming approach. From a previous Rascal-MPL implementation, this project aims to build the grammar and rules for the complete Meta-CrySL language by using the *Xtext* framework. This new version of Meta-CrySL will be evaluated against a set of available Meta-CrySL files to assess its correctness.

Keywords: MetaCrySL, DSL, cryptography

Sumário

1 Introdução **1**
1.1 Contexto 2
1.2 Objetivos 2
1.2.1 Objetivos Específicos 3
1.3 Metodologia 3
1.4 Organização 4

2 Revisão da literatura **5**
2.1 A Linguagem CrySL 5
2.1.1 A linguagem de metaprogramação Meta-CrySL 8
2.2 Implementação de linguagens 9
2.2.1 Engenharia de Linguagens 12
2.3 Solução em Rascal-MPL 13
2.4 O framework Xtext 14
2.5 A Linguagem de Programação Xtend 15

3 MetaCrySL em Xtext **16**
3.1 *MetaCrySL Parser* 17
3.1.1 A gramática CrySL estendida 17
3.1.2 A gramática de refinamentos 19
3.1.3 A gramática de configuração 20
3.1.4 Geração automatizada de testes 21
3.2 *MetaCrySL Preprocessor* 23
3.3 *MetaCrySL PrettyPrinter* 26
3.4 Avaliação dos Resultados 28

4 Conclusão **32**

Referências **34**

Apêndice **34**

A Dados Coletados **35**

Lista de Figuras

2.1 *Pipeline* de uma aplicação de linguagem de programação [1] 10

3.1 *Pipeline* para a linguagem Meta-CrySL 16
3.2 Arquitetura do gerador de testes unitários 23
3.3 Arquitetura do gerador de código 24

Lista de Tabelas

3.1 Arquivos de configuração utilizados nos testes 29

A.1 Resultados de testes para *Android0108* 36
A.2 Resultados de testes para *Android0116* 37
A.3 Resultados de testes para *Android25plus* 38

Capítulo 1

Introdução

A *Java Cryptography Architecture* (JCA) é a *Application Programming Interface* (API) criptográfica mais popular entre desenvolvedores da linguagem Java [2], fazendo parte da API de segurança nativa da linguagem. Estando disponível há mais de 20 anos, essa API é reconhecida pela sua flexibilidade, o que no entanto colabora para torná-la mais difícil de utilizar corretamente [2]. Dessa maneira, mesmo estando disponíveis há tanto tempo, uma série de vulnerabilidades em software podem ser pelo uso incorreto dessas APIs.

O plugin CogniCrypt, disponível para a IDE Eclipse, surgiu com a proposta de resolver o problema do uso incorreto de APIs criptográficas, não somente na biblioteca JCA mas também em outras APIs de criptografia comumente utilizadas em ambientes Java [3] [4]. O CogniCrypt oferece ferramentas de geração de código a partir de casos de uso e também faz a análise estática de código de modo a checar o uso correto das APIs suportadas.

A análise estática provida pelo CogniCrypt é possível graças a DSL (Domain Specific Language) CrySL, criada justamente com o intuito de facilitar o trabalho de descrição de uso correto das APIs criptográficas [3] [4]. A partir das especificações de uso correto de APIs descritas através da linguagem CrySL, o CogniCrypt pode realizar a análise estática do código e avisar ao usuário caso uma falha de segurança tenha sido introduzida no código.

No entanto, um dos desafios que se apresenta com o uso da linguagem CrySL é o reuso de tais especificações. APIs criptográficas variam bastante, sendo algumas mais flexíveis (e por consequência abrindo mais espaço para introdução de erros) enquanto outras são mais restritas e apresentam menos opções de configuração para o desenvolvedor. As fontes dessa variabilidade nas APIs podem prover de diversas fontes: desde padrões diferentes de segurança tais como FIPS/BSI a evoluções e mudanças no desenho das próprias APIs.

1.1 Contexto

Foi com o intuito de resolver o problema de reuso na linguagem CrySL que foi proposta a implementação de uma linguagem de metaprogramação chamada Meta-CrySL. Os requisitos iniciais no projeto da linguagem Meta-CrySL eram suportar as fontes de variabilidade mais comuns em APIs criptográficas e também que ela fosse compilada para CrySL, de modo que a estrutura de análise já existente no ecossistema do CogniCrypt pudesse ser aproveitada.

A implementação inicial da linguagem Meta-CrySL foi realizada em Rascal-MPL. A motivação para essa escolha foi o fato dessa linguagem ser voltada principalmente para o domínio da metaprogramação, contando com diversos construtos que facilitariam a implementação inicial do projeto. Essa implementação cumpriu os requisitos de resolver os problemas de variabilidade inerentes ao uso de APIs criptográficas e de geração de código final em CrySL.

O desenvolvimento em Rascal-MPL, no entanto, dificultou a incorporação da linguagem Meta-CrySL no meio do ecossistema CogniCrypt. O motivo principal dessa dificuldade vem da dificuldade prover suporte para o projeto, visto que basicamente toda a base de código do projeto CogniCrypt é escrita em Java. Identificou-se, dessa maneira, a oportunidade de implementar a estrutura definida para Meta-CrySL numa linguagem que pudesse ser mantida mais facilmente pela equipe do CogniCrypt. Para isso foi escolhido o framework Xtext [5], uma infraestrutura para geração de linguagens de programação que se integra facilmente com a IDE Eclipse e que é a base para a implementação da própria linguagem CrySL. Além disso, o Xtext utiliza a linguagem Xtend, que compila para Java 8 e utiliza as mesmas bibliotecas que o próprio Java utiliza, além de ser mais expressiva que Java e possibilitar um desenvolvimento mais rápido.

1.2 Objetivos

O objetivo principal desse trabalho é realizar o porte da implementação original de MetaCrySL, realizada em Rascal-MPL, para Xtext. Desse modo, as seguintes atividades devem ser realizadas:

1. Estudo das tecnologias Xtext e Xtend, bem como das melhores práticas para desenvolvimento de meta-programas, ou seja, programas que manipulam programas.

2. Implementação de Meta-CrySL em Xtext, reportando a experiência de uso das tecnologias Xtext e Xtend.

3. Avaliar a implementação, reproduzindo um dos estudos realizados com a implementação original de Meta-CrySL.

1.2.1 Objetivos Específicos

A primeira fase de desenvolvimento envolveu a implementação da gramática base CrySL e as extensões propostas para atender aos requisitos de Meta-CrySL. A escrita da gramática foi realizada na linguagem de gramáticas fornecida pelo framework Xtext, a qual provê uma ferramenta para escrita de gramáticas com geração automática de um parser.

A segunda etapa se trata da parte principal: no gerador de código são resolvidos problemas de variabilidade e reuso que são o foco principal de Meta-CrySL. A partir de uma única especificação Meta-CrySL podem ser geradas uma série de especificações CrySL diferentes, atendendo a diferentes padrões de segurança e versões de APIs. Esse processo ocorre na etapa de geração de código.

Por último, o Pretty Printer é o componente que irá receber uma representação em forma de árvore sintática de uma especificação CrySL e imprimir um arquivo correspondente no sistema de arquivos. O arquivo gerado pela ferramenta poderá ser consumido pela infraestrutura do CogniCrypt.

Como resultado final, essa implementação será entregue para ser mantida pela equipe do CogniCrypt. Desse modo, espera-se que a linguagem Meta-CrySL possa fazer parte do ecossistema do CogniCrypt, facilitando o reuso de especificações de uso correto de API's criptográficas através de versões diferentes dessas mesmas APIs e padrões distintos de segurança tais como FIPS e BSI.

1.3 Metodologia

De modo a atingir os objetivos da nova implementação, foi feito primeiramente um estudo das tecnologias Xtext e Xtend, de modo a possiblitar a familiarização com o ecossistema Xtext e com as práticas de desenvolvimento de linguagens dentro da plataforma. A própria documentação de Xtext e Xtend serviram como base desse estudo, assim como o livro *Implementing Domain-Specific Languages with Xtext and Xtend* [5]. Também foi feito um estudo de técnicas de implementação de linguagens, baseada principalmente no livro *Language Implementation Patterns* [1]. A implementação de Meta-CrySL foi bastante influenciado por esses dois trabalhos.

Para avaliar a nova implementação, foram replicados testes realizados com a versão original de Meta-CrySL. A partir dos mesmos arquivos de entrada, foram comparadas as saídas geradas pela nova implementação com as saídas geradas pela implementação em Rascal-MPL, e as diferenças foram comparadas de modo a avaliar o desempenho.

1.4 Organização

Esse trabalho está organizado da seguinte maneira: em primeiro lugar será feita uma descrição detalhada a respeito da arquitetura de alto nível da linguagem Meta-CrySL e sobre a implementação anterior feita em Rascal-MPL. Além disso, será feita também uma revisão da arquitetura a respeito de Grammarware, com foco nos princípios de engenharia de linguagens que foram utilizados na implementação.

Em seguida, será discutida a implementação em Xtext e Xtend da linguagem Meta-CrySL, fazendo-se primeiro uma avaliação dos benefícios do uso do Xtext nesse contexto. Além disso, serão discutidas as metodologias de engenharia de software utilizadas para se implementar a arquitetura Meta-CrySL através da linguagem Xtend.

Por último será feita uma avaliação dos resultados obtidos, testando-se a nova implementação em Xtext contra uma série de testes que já foram testados anteriormente pela implementação em Rascal-MPL, e também algumas sugestões sobre avanços futuros em cima do trabalho aqui descrito.

O código-fonte dessa implementação encontra-se no GitHub em https://github.com/PAMunb/MetaCrySL[6].

Capítulo 2

Revisão da literatura

Conforme foi mencionado no capítulo anterior, o principal objetivo desse trabalho de graduação é construir uma nova implementação da linguagem Meta-CrySL. A linguagem Meta-CrySL é uma extensão de uma linguagem de especificação chamada CrySL, a qual possibilita a escrita de regras para o uso correto de API's de criptografia.

Desse modo, esse trabalho requer um conhecimento tanto na linguagem CrySL como também em técnicas de implementação de linguagens.

2.1 A Linguagem CrySL

É comum que desenvolvedores façam o uso de bibliotecas criptográficas nativas dos ambientes de desenvolvimento que estejam utilizando: uma série de API's e bibliotecas criptográficas existem para as principais linguagens de programação disponíveis no mercado. No entanto, existem trabalhos que reportam a dificuldade dos desenvolvedores em utilizar essas API's [2] [7].

Um exemplo disso pode ser observado na API JCA/JCE (Java Cryptography Architecture), a qual foi implementada de modo que o algoritmo de criptografia a ser utilizado possa ser facilmente modificado pelo desenvolvedor sem que muitas linhas de código tenham de ser alteradas. A classe *MessageDigest* da JCA, por exemplo, necessita que o desenvolvedor especifique o algoritmo criptográfico a ser utilizado, assim como a sequência de chamadas de métodos que são necessárias sobre os dados de entrada. Essa flexibilidade no uso da biblioteca, no entanto, abre espaço para que erros sejam cometidos. Nesse caso, é comum que desenvolvedores especifiquem algoritmos inseguros (como MD5 ou SHA-1). Desenvolvedores também podem esquecer de chamar alguns métodos, ou mesmo chamá-los na ordem incorreta.

Uma forma de contornar esse problema é através da utilização de uma linguagem de especificação de uso correto de API's criptográficas juntamente com o suporte de uma

ferramenta de análise estática. Essa abordagem foi adotada na construção do plugin CogniCrypt [4] para a IDE Eclipse, o qual consiste na implementação de uma *Domain Specific Language* chamada CrySL, juntamente com uma ferramenta de análise estática de código.

No caso do plugin CogniCrypt, a análise estática é executada toda vez que o desenvolvedor salva o código no seu editor, reportando alertas e erros no console do Eclipse a cada vez que uma API criptográfica é utilizada de maneira incorreta. Já o uso correto, por sua vez, é especificado via regras CrySL.

Os tipos de erros reportados pelo CogniCrypt são:

1. **Constraint Error:** Disparado quando a análise estática detecta uma String (ou inteiro) incorreto sendo enviado como parâmetro para uma chamada de método.
2. **TypeState Error:** Quando a sequência de chamadas de método em um objeto não está de acordo com a especificação CrySL.
3. **Incomplete Operation Error:** Quando uma chamada de método necessária em um objeto está faltando.
4. **Required Predicate Error:** Disparado quando a análise infere que a combinação do uso de alguns objetos é incorreta.

Todos esses erros são encontrados a partir de divergências das regras CrySL que acompanham uma instalação do CogniCrypt. A linguagem CrySL em si foi desenvolvida com o propósito primordial de permitir a escrita de especificações de uso correto de API's criptográficas por especialistas em criptografia e segurança computacional [3]. Com base no apoio desses especialistas as seguintes decisões de projeto foram tomadas em relação ao desenho da linguagem CrySL:

1. **Allow List:** O uso correto das API's criptográficas é definido de maneira explícita onde aplicável, e quaisquer desvios são considerados erros.
2. **Typestate e Fluxo de Dados:** A linguagem CrySL deve ser capaz de especificar quais métodos podem ser chamadas nos objetos da API e em qual ordem.
3. **Restrições:** Muitas API's criptográficas utilizam *strings* para parametrizar algoritmos de critpografia específicos. A linguagem CrySL deve ser capaz de especificar quais parâmetros são aplicáveis para determinadas chamadas de método.

Especificações CrySL são determinadas a partir das classes criptográficas individuais. A primeira linha de uma especificação CrySL, portanto, especifica a qual classe Java ela se refere. Essa cláusula é determinada a partir da palavra chave `SPEC`. Depois

desta, as cláusulas seguintes especificam os campos chamados de `OBJECTS`, `EVENTS`, `ORDER`, `CONSTRAINTS` e `ENSURES`. Juntas, essas seis cláusulas formam os componentes essenciais e obrigatórios para qualquer especificação CrySL.

A cláusula `OBJECTS` é responsável por definir quais objetos serão utilizados nas cláusulas seguintes da especificação. Já a cláusula `EVENTS` é responsável por definir (explicitamente ou através de *pattern-matching*) quais métodos podem ser usados na classe especificada. Eventos também podem ser definidos por meio de *agregações*, que são disjunções de múltiplos eventos.

A seção `ORDER` especifica um tipo de expressão regular e especificam as ordens dos eventos que devem acontecer no objeto especificado. Já a seção `CONSTRAINTS` define uma série de restrições impostas sobre os objetos definidos na cláusula `OBJECTS`.

A última cláusula mandatória numa especificação CrySL é `ENSURES`, sendo responsável por definir as interações entre múltiplas classes, especificando o que a classe em questão garante desde que tenha sido usada corretamente. Predicados especificados dentro de uma cláusula `ENSURES` podem ser *requeridos* por outras classes através da cláusula opcional `REQUIRES`.

O trecho de código a seguir (retirado da página oficial do plugin CogniCrypt) ilustra as cláusulas obrigatórias da linguagem CrySL (assim como a cláusula `REQUIRES` opcional).

```
SPEC   javax.crypto.Cipher
OBJECTS
    java.lang.String trans;
    byte[] plainText;
    java.security.Key key;
byte[] cipherText;
EVENTS
    Get: getInstance(trans);
    Init: init(encmode, key);
    doFinal: cipherText = doFinal(plainText);
ORDER
    Get, Init, (doFinal)+
CONSTRAINTS
    encmode in {1,2,3,4};
    alg(trans) in {"AES", ..., "RSA"};
    alg(trans) in {"AES"} => mode(trans) in {"CBC"};
REQUIRES
    generatedKey[key, part(0, "/", trans)];
ENSURES
```

```
    encrypted[cipherText, plainText];
```

Além de `REQUIRES`, CrySL também oferece suporte para a cláusula opcional `FORBIDDEN`. Essa seção especifica quais métodos jamais devem ser chamados por serem inseguros. A adição da cláusula `FORBIDDEN`, apesar de se afastar da decisão de projeto de utilizar uma abordagem de *allow listing*, contribui para tornar as especificações CrySL mais concisas e fáceis de se escrever. Por último, a cláusula opcional `NEGATES` permite a invalidação de um predicado existente.

2.1.1 A linguagem de metaprogramação Meta-CrySL

A linguagem Meta-CrySL surge com a proposta de facilitar o reuso de especificações CrySL, abordando os problemas relacionados à variabilidade das API's e normas distintas emitidas por diversas entidades especializadas em criptografia. De modo a resolver esses dois problemas, a arquitetura de alto nível de Meta-CrySL é baseada em três linguages principais: são elas a linguagem CrySL estendida, a linguagem de refinamentos e a linguagem de configurações.

A linguagem CrySL estendida é necessária para permitir que especialistas de segurança possam escrever regras CrySL de maneira mais genérica, permitindo o reuso dessas regras entre diversas especificações. Pontos de variação das regras são especificados através de meta-variáveis tais como no exemplo a seguir:

```
CONSTRAINTS
    algorithm in $set;
```

Aqui, `$set` é uma meta-variável que será substituída por uma string representando o algoritmo a ser utilizado no `CONSTRAINT` na regra final.

O outro tipo de variabilidade endereçada através da linguagem CrySL estendida é a variabilidade de tipos, as quais podem ser resolvidas por meio de parâmetros de tipo suportados pela linguagem:

```
OBJECTS
    <T> primitive;
```

No entanto, um dos aspectos mais poderosos de Meta-CrySL é a sua linguagem de refinamentos. Os refinamentos de Meta-CrySL permitem estabelecer transformações em especificações através da inserção de novos *constraints*, eventos e ordenamentos. Para realizar essas transformações, a linguagem de refinamentos espera receber uma especificação base (escrita na linguagem CrySL estendida) e uma lista de refinamentos que podem ser aplicados.

Por último, a linguagem de configuração é responsável por ditar o processo de *build* das regras Meta-CrySL. Os arquivos de configuração ditam uma especificação base e uma série de refinamentos que deverão ser aplicados em cima dessa especificação. Nesses arquivos são também indicados os caminhos onde podem ser encontrados os arquivos de refinamentos e a especificação base, assim como um diretório de saída onde os arquivos CrySL compilados deverão ser escritos.

A partir dessa linguagem de configuração, um especialista em segurança pode, para um mesmo conjunto de especificações e refinamentos, especificar diversos arquivos CrySL diferentes a depender dos objetivos desejados: podem ser geradas regras CrySL para diferentes versões de API's ou para atender os requisitos de diferentes órgãos de criptografia.

Como a linguagem MetaCrySL gera arquivos CrySL, é uma decisão de projeto da própria linguagem que as checagens de tipo sejam deixadas para a ferramenta *Crypto-Analysis* do CogniCrypt. Essa decisão permite que a implementação de Meta-CrySL seja mais simples, embora não garanta que os arquivos CrySL gerados tenham os tipos corretos.

2.2 Implementação de linguagens

O foco principal desse trabalho de graduação consiste na implementação de uma *Domain Specific Language* baseada em CrySL e com algumas extensões. Desse modo, boa parte do preparo para esse projeto consistiu no estudo de técnicas de implementação de linguagens. Linguagens de computador são complicadas de se implementar corretamente, de modo que um bom entendimento das melhores práticas e processos de desenvolvimento e manutenção de linguagens são necessários.

De maneira geral, é possível separar o projeto de uma linguagem em uma série de componentes menores que podem ser combinados em um *pipeline*. A Figura 2.1 mostra o esquema genérico dos componentes que fazem parte do projeto de uma linguagem de programação [1].

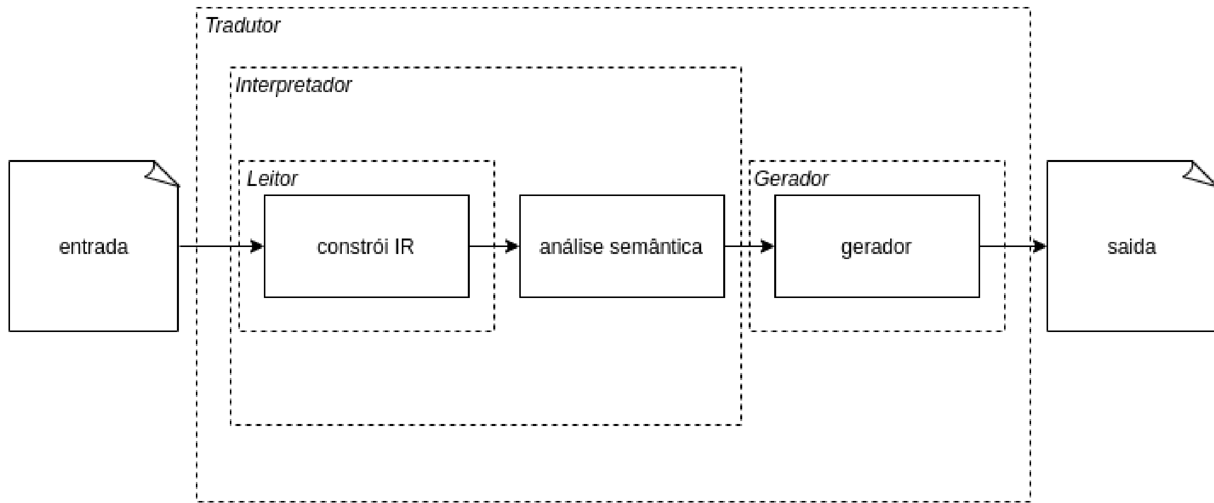

Figura 2.1: *Pipeline* de uma aplicação de linguagem de programação [1]

A depender do tipo de aplicação que está sendo desenvolvida, apenas algumas das partes desse *pipeline* serão utilizadas. Em geral há quatro tipos de projeto de linguagem de programação:

1. *Leitor*: O trabalho de um leitor é construir uma estrutura de dados (chamada de representação intermediária ou IR, do inglês *Intermediate Representation.* Os dados de entrada podem ser arquivos de texto ou arquivos binários.

2. *Gerador*: Um gerador percorre a IR e emite uma saída (*output*)

3. *Tradutor*: Um tradutor lê arquivos de texto ou binários e emite uma saída em outra linguagem, sendo tipicamente resultante da união de um leitor e um gerador.

4. *Interpretador*: Um interpretador lê, decodifica e executa as instruções presentes no arquivo de entrada.

O projeto de Meta-CrySL se encaixa como um tradutor. Isto é, são lidos arquivos de entrada na linguagem de programação Meta-CrySL e o *output* são arquivos na linguagem CrySL pura. No próximo capítulo, os componentes individuais que compõe o *pipeline* de Meta-CrySL serão especificados.

O primeiro componente no *pipeline* de projetos de linguagem e programação, o leitor, envolve a construção de algum tipo de analisador sintático (*parser*). O trabalho de um analisador sintático é checar se uma determinada sentença na entrada corresponde à sintaxe da linguagem. Esses analisadores sintáticos analisam uma cadeia de entrada, buscando identificar as várias estruturas que fazem parte daquela linguagem em particular. Esse tipo de análise pode ser feita através de uma estratégia *top-down*, onde se

começa analisando a partir do topo da árvore sintática, descendo até os nós terminais. Analisadores sintáticos desse tipo são conhecidos como analisadores sintáticos de descida recursiva [1].

Como a escrita de um analisador sintático a partir do zero é um trabalho repetitivo e suscetível a erros, é comum a utilização de uma linguagem de domínio específico que permita a especificação de uma gramática. Uma linguagem cono essa, que possibilita a descrição de outras linguagens, é acompanhada do uso de ferramentas que traduzem a gramática em um parser equivalente de maneira automática. Uma ferramenta como essas (e que é utiliza pelo próprio Xtext [5]), é o ANTLR [1]. O ANTLR é uma ferramenta comumente utilizada no projeto de linguagens de programação, e é capaz de gerar parsers através de uma gramática usando uma estratégia de mapeamento de gramáticas em parsers de descida recursiva.

Existem diversas outras abordagens que podem ser utilizadas na construção de um analisador sintático, e a decisão de qual estratégia utilizar se baseia, em primeiro lugar, nos requisitos da própria linguagem a ser desenvolvida. É natural que linguagens mais complicadas, com expressões mais complexas, exijam ferramentas de análise mais sofisticadas.

No ANTLR, por exemplo, a construção do parser ocorre por meio de um algoritmo de mapeamento da gramática em um analisador sintático descendente recursivo. A partir da análise sintática, a representação intermediária é construída. O tipo de representação intermediária a ser utilizada também depende muito do tipo de linguagem que está sendo desenvolvida: compiladores, por exemplo, em geral precisam de representações intermediárias mais próximas das instruções de baixo nível a serem executadas na máquina. A construção de uma representação intermediária, além de evitar a necessidade de repetir o processo de análise sintática a cada estágio do *pipeline*, facilita a extração dos dados a serem utilizados nas etapas posteriores.

A análise semântica da linguagem lida, por sua vez, consiste na implementação de ferramentas que consigam entender o que as cadeias de entrada querem dizer. É a partir dessa análise que a informação relevante é extraída da entrada. Para isso, é necessária a construção de algum tipo de *tabela de símbolos*. Uma tabela de símbolos é basicamente um tipo de dicionário cujo trabalho é mapear símbolos em suas respectivas definições. O tipo específico de tabela a ser implementado também depende muito da aplicação e características da linguagem sendo desenvolvida.

Feita a análise semântica, há dois caminhos que o *pipeline* da linguagem pode seguir: pode ser desenvolvido um interpretador, que executa as instruções armazenadas na representação intermediária, ou o programa pode ser traduzido numa outra linguagem. Nessa segunda opção, o último componente do *pipeline* passa a ser um gerador de código, o qual

é responsável por emitir código fonte na linguagem de saída especificada.

2.2.1 Engenharia de Linguagens

Como linguagens de programação são artefatos complexos e complicados de se implementar corretamente, foram sugeridas maneiras de aplicar técnicas de engenharia de software na implementação de ferramentas que manipulam linguagens. Uma dessas abordagens foi proposta por Ralf Lämmel em seu artigo, *Towards an engineering discipline for GRAMMARWARE* [8], o qual busca trazer os princípios e práticas da engenharia de software para o projeto de gramáticas e linguagens. O termo *grammarware* é utilizado, nesse contexto, para indicar gramáticas e todo o ferramental relacionado ao desenvolviemnto de linguagens, caracterizando, dessa forma, um conjunto de práticas que tem como objetivo aumentar a qualidade e a produtividade na construção desses artefatos.

Além disso, o termo *grammarware* não denota apenas gramáticas no uso mais comum da palavra, relacionado à linguagens de programação como C, Java ou Python: sistemas de software atuais dependem cada vez mais de gramáticas no formato de API's, protocolos e linguagens de domínio específico, que também podem ser considerados como diferentes tipos de gramáticas. Desse modo, a disciplina de *grammarware* é aplicável não apenas no contexto mais restrito de linguagens de programação, mas nesse contexto mais geral e abrangente.

Sob essa perspectiva, foram propostos os seguintes seis princípios para guiar o desenvolvimento de *grammarware*:

1. Generalidade: deve ser capaz de englobar gramáticas no uso mais geral da palavra, desde as mais simples (formatos de troca de dados, por exemplo) até os casos mais complexos.

2. Abstração: visa não restringir gramáticas a aplicações tecnológicas específicas, tendo como objetivo principal a construção de gramáticas puras. Desse modo, busca-se fazer uma separação entre projeto e implementação de gramáticas.

3. Customização: a partir da gramática *pura* idealizada no item anterior, deve ser possível customizar a gramática para a aplicação específica desejada.

4. Separação de conceitos: a ideia principal desse ponto é aplicar conceitos conhecidos de engenharia de software, tais como modularização, no contexto de *grammarware*, permitindo que cada elemento de um *grammarware* seja responsável por uma funcionalidade específica.

5. Evolução: como outros artefatos tecnológicos, há a constante necessidade de evolução por parte de *grammarware* também. Logo, deve ser estabelecida uma metodologia que permite a evolução de gramáticas.

6. Avaliação: deve ser possível estabelecer métricas quer permitam mensurar a qualidade de uma gramática. Ou seja, a disciplina de testes e testes automatizados deve ser aplicada também no desenvolvimento de *grammarware*

7. Automação: assim como em outras disciplinas de software, no contexto de *grammarware* a automação deve ser aplicada onde for possível.

O último ponto, referente a automação, é especialmente importante no contexto de *grammarware*. Transformações automáticas da gramática devem ser utilizadas para melhorar a rastreabilidade e escalabilidade de artefatos de engenharia. Essas transformações, em geral, podem estar relacionadas com a própria gramática (envolvendo elementos como sintaxe) ou como programas dependentes da gramática, além de transformações nas próprias estruturadas de dados utilizadas pela linguagem como um todo.

Nesse contexto, pode-se falar também de refatoração da gramática. A refatoração é um processo de transformação que, assim como no contexto de engenharia de software no sentido mais amplo, possibilita a evolução dos artefatos de engenharia sem, no entanto, alterar a sua funcionalidade. Uma disciplina de engenharia aplicada a gramáticas deve, dessa maneira, possibilitar a refatoração, assim como outras transformações que venham a acompanhar a evolução de gramáticas: novas regras de produção podem vir a ser necessárias, ou certas regras obsoletas podem ser removidas. Uma disciplina de engenharia aplicada a gramáticas deve ser capaz de possibilitar todas essas transformações.

Os princípios listados, no entanto, representam um estágio ideal para a implementação de *grammarware*, e sua total aplicação dentro de projetos reais ainda é desafiadora. Existem diversos desafios relacionados à complexidade de projetos de *grammarware* no mundo real que exigem mais pesquisa e avanços nessa área. Com a presença cada vez maior de artefatos de gramáticas e linguagens em sistemas computacionais modernos, a necessidade de uma engenharia de gramáticas torna-se cada vez mais aparente.

2.3 Solução em Rascal-MPL

Como indicado anteriormente, a primeira versão da linguagem Meta-CrySL foi implementada em Rascal-MPL. A linguagem Rascal-MPL é voltada especificamente para auxiliar no processo de desenvolvimento de linguagens de programação e DSL's, de modo que se tratava de uma boa alternativa para a implementação inicial, permitindo uma prova de conceito rápida.

No entanto, a adoção de Rascal-MPL dificulta a incorporação de Meta-CrySL dentro do ecossistema do plugin CogniCrypt. Como a própria linguagem CrySL é implementada através de Xtext, que além de ser um framework de construção de DSL's bastante completo, possui também uma integração imediata com a IDE Eclipse. Dessa maneira, foi identificado que, para facilitar a manutenção e a implantação de Meta-CrySL seria necessário realizar um porte da ferramenta para dentro do framework Xtext.

2.4 O framework Xtext

O Xtext é um framework para o desenvolvimento de linguagens de programação e DSL's [5]. Como tal, o Xtext fornece toda a infraestrutura de código necessária para escrever e manter uma linguagem: *parsers*, *typecheckers*, compiladores, até recursos avançados de integração com a IDE. Dessa maneira, é possível testar protótipos rapidamente e desenvolver de maneira iterativa, permitindo que o desenvolvedor possa focar completamente nas especificidades da linguagem que está desenvolvendo [5].

Além disso, o Xtext possui uma excelente integração com a IDE Eclipse, trazendo recursos como coloração de sintaxe e checagem de erros. A versão *Eclipse IDE for Java and DSL Developers* do Eclipse já vem com uma instalação de Xtext e Xtend.

O ponto principal de Xtext é sua linguagem de gramáticas. Esta por si só é uma DSL que permite a escrita de gramáticas através de uma linguagem simples e dinâmica (a própria linguagem de gramáticas é implementada em Xtext). O Xtext utiliza o ANTLR para a geração do analisador sintático a partir da gramática especificada.

Um exemplo de uma pequena regra escrita na linguagem de gramáticas de Xtext (e retirada da própria base de código de Meta-CrySL) pode ser vista logo abaixo:

```
MethodDef: methodName = ID '(' (args = FormalArgs)? ')' ;
```

Aqui, os nomes das regras são introduzidos antes do símbolo `:`, que nesse caso é `MethodDef`. É indicado aqui que elementos desse tipo são compostos por um atributo obrigatório chamado `methodName` e um atributo opcional chamado `args`. O operador `()?` denota que uma regra é opcional. Além disso, o atributo `args` está dentro dos literais `'('` e `')'`. Literais em Xtext são representados por aspas simples ou duplas.

Os atributos `methodName` e `args` são parseados por outras regras, nesse caso `ID` e `FormalArgs` respectivamente. A regra `ID` não é declarada explicitamente nessa gramática pois é herdada por todas as gramáticas de Xtext (Xtext fornece um conjunto de regras de uso comum para facilitar o desenvolvimento). Já a regra `FormalArgs` é definida mais adiante na gramática.

Dessa maneira, a gramática explicitada acima poderia realizar a análise sintática das declarações dos seguintes tipos:

```
method1()
method1(int, int)
```

A maior parte das regras definidas na linguagem de gramáticas de Xtext segue o mesmo padrão da regra apresentada acima como exemplo.

O Xtext utiliza o **MWE2 (Modeling Workflow Engine 2)** para executar a geração de artefatos. Um arquivo *.mwe2* é gerado no mesmo diretório onde fica o arquivo *.xtext* responsável pela definição das regras da gramática da linguagem. Ao executar esse arquivo, o Xtext deriva a especificação no formato ANTLR a partir da gramática, gerando os arquivos Java necessários no diretório *src-gen* do projeto correspondente.

2.5 A Linguagem de Programação Xtend

Xtend é uma linguagem de uso geral interoperável com Java. Além disso, Xtend também é uma linguagem mais compacta, oferecendo recursos como: inferência de tipos e *extension methods*. O fato de que Xtend é interoperável também permite que sejam utilizadas todas as bibliotecas Java normalmente.

A similaridade de Xtend com Java torna simples o uso da linguagem, sem necessitar de uma longa curva de aprendizado. Dessa maneira, desenvolvedores Java podem rapidamente se tornar produtivos no uso da linguagem e escreverem código mais enxuto e fácil de entender.

A linguagem Xtend foi desenvolvida utilizando o framework Xtext. Novos projetos em Xtext já assumem o uso de Xtend por padrão. Dessa maneira, o uso do Xtend para a escrita do gerador de código para a linguagem Meta-CrySL foi natural, pois Xtend possui uma forte integração com Xtext [5]. Como o próprio Xtend possui algumas funcionalidades interessantes para projetos de linguagens de programação (como, por exemplo, *template expressions*, que são bastante úteis para a construção de geradores de código), percebeu-se que a adoção da linguagem permitiria uma iteração e entregas rápidas no desenvolvimento de Meta-CrySL. Desse modo, Xtend foi utilizado no projeto para implementar os componentes do *pipeline* que lidam com a manipulação da representação intermediária e geração de código.

Capítulo 3

MetaCrySL em Xtext

Nesse capítulo será apresentada a solução proposta para a implementação de Meta-CrySL utilizando o *framework* Xtext. Essa apresentação ocorrerá em três partes: em primeiro lugar, será abordada a estratégia de implementação das três gramáticas necessárias para o funcionamento de Meta-CrySL: a linguagem CrySL estendida, a linguagem de refinamentos e a linguagem de configuração. Em conjunto, será descrito o processo adotado para geração automática de testes unitários do analisador sintático. Atenção especial será dada à estratégia de aplicação dos refinamentos a partir dos arquivos de configuração, módulo que aqui é chamado de *Meta-CrySL Preprocessor*. Por último, será também apresentado o módulo *PrettyPrinter*, o qual é responsável pela geração de código CrySL.

A Figura 3.1 apresenta o *pipeline* de linguagem específico para a implementação Meta-CrySL. Aqui são representados os três módulos principais de Meta-CrySL.

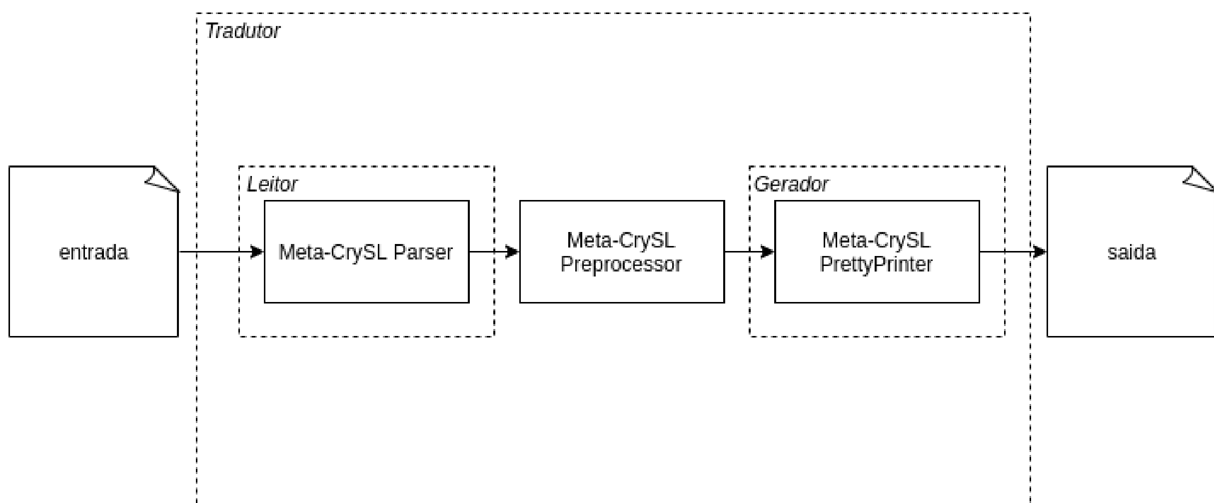

Figura 3.1: *Pipeline* para a linguagem Meta-CrySL

3.1 *MetaCrySL Parser*

Nessa seção serão abordados os detalhes a respeito da implementação das três gramáticas que compõe Meta-CrySL: as gramáticas para a linguagem CrySL estendida, a linguagem de refinamentos e a linguagem de configuração. Ao analisador sintático (*parser*) gerado pela combinação dessas três gramáticas é dado o nome de *Meta-CrySL Parser*.

No capítulo anterior foi apresentada a ideia de utilização de ferramentas de construção de analisadores sintáticos a partir de descrições de gramáticas. Ferramentas desse tipo, como o próprio ANTLR, em geral executam algoritmos de mapeamento de gramáticos em analisador sintáticos descendentes recursivos devido à sua natureza *top-down*. De fato, o Xtext utiliza o ANTLR internamente para geração automática de um analisador sintático.

A utilização de uma ferramenta de geração automática de uma analisador sintático a partir da descrição de uma gramática faz uso dos princípios de projeto de *grammarware* apresentados no capítulo anterior, notadamente o princípio referente à automação de transformações de gramáticas. O uso de uma ferramenta que automatiza o processo de geração do analisador sintático permite a refatoração e evolução fácil da gramática, sem que exista a necessidade de reescrever uma grande quantidade de código manualmente.

3.1.1 A gramática CrySL estendida

As especificações base em Meta-CrySL são escritas numa linguagem muito parecida CrySL, com a exceção de alguns construtos novos adicionados que permitem solucionar os problemas de variabilidade de API's criptográficas. Dessa maneira, a primeira etapa no projeto de implementação de Meta-CrySL em Xtext consistiu no processo de tradução das regras de gramática de CrySL para dentro do ambiente Xtext.

As três gramáticas que compõe Meta-CrySL estão definidas dentro do mesmo arquivo no projeto Xtext, chamado *MetaCrySL.xtext*. A primeira regra da gramática especifica as regras iniciais para cada uma das três gramáticas:

```
Model
  : {MetaCrySL} spec = Spec
  | {Refinement} refinement = Refinement
  | {Configuration} configuration = Configuration
 ;
```

Em seguida, a regra `Spec` é definida através do seguinte trecho de código, o qual é responsável por indicar as diferentes cláusulas que compõe uma especificação CrySL completa. Pode-se observar, pela regra a seguir, que a estrutura base para uma especificação CrySL é bastante simples:

```
Spec
 : 'ABSTRACT'? 'SPEC' classType = BaseSpecType
     objectSpec    = ObjectSpec
     (forbiddenSpec = ForbiddenSpec)?
     eventSpec     = EventSpec
     orderSpec     = OrderSpec
     (constraintSpec = ConstraintSpec)?
     (requireSpec = RequireSpec)?
     (ensureSpec = EnsureSpec)?
     (negateSpec = NegateSpec)?
;
```

Pode-se perceber que alguns dos campos são marcados como opcionais através do operador `()?` (mais especificamente, os campos `forbiddenSpec`, `constraintSpec`, `requireSpec`, `ensureSpec` e `negateSpec`).

A estrutura adotada para essa gramática permite que novas cláusulas para a regra `Spec` sejam adicionadas (ou que cláusulas não utilizadas sejam retiradas) de maneira bastante simples. Dessa maneira, a gramática é facilmente extensível. Existe um nível de interdependência entre algumas das regras mais similares ou que compartilham entre si algumas regras de uso geral, mas de maneira geral pode-se dizer que o padrão adotado nesse projeto permite uma manutenção e refatoração simples da gramática.

Atenção especial deve ser dada às regras que necessitavam do uso de *left-factoring* para evitar recursão à esquerda. Em gramáticas livres do contexto, dizemos que um não-terminal é recursivo à esquerda caso o símbolo mais à esquerda em uma de suas substituições seja ele próprio.

O parser gerado pelo **ANTLR** não é capaz de suportar recursão à esquerda, de modo que a técnica de *left-factoring* deve ser utilizada. Em Meta-CrySL, recursões à esquerda ocorrem em expressões de eventos, dentro da cláusula `EVENTS`. O conjunto de regras completas para eventos (com o uso de *left-factoring* pode ser observado a seguir):

```
EventExp
 :  ChoiceExp ;

ChoiceExp returns EventExp
 : SequenceExp => ({ChoiceExp.left = current} '|' right = SequenceExp)*;

SequenceExp returns EventExp
 : BasicEventExp => ({SequenceExp.left = current} ',' right = BasicEventExp)*;
```

```
BasicEventExp
 : {Optional}    exp = PrimaryExp '?'
 | {ZeroOrMore}  exp = PrimaryExp '*'
 | {OneOrMore}   exp = PrimaryExp '+'
 | {PrimaryExp}  exp = PrimaryExp
 ;

PrimaryExp returns EventExp
 : label = ID
 | '(' ChoiceExp ')'
 ;
```

Mais especificamente, as duas regras a seguir:

```
ChoiceExp returns EventExp
 : SequenceExp => ({ChoiceExp.left = current} '|' right = SequenceExp)*;

SequenceExp returns EventExp
 : BasicEventExp => ({SequenceExp.left = current} ',' right = BasicEventExp)*;
```

3.1.2 A gramática de refinamentos

A gramática de refinamentos é onde se encontra o maior poder de Meta-CrySL. Através dela, é possível customizar uma especificação de entrada para uma determinada variação das regras de uso das API's criptográficas.

As transformações de refinamento suportadas por Meta-CrySL são:

- **Definir conjunto de literais** : Vincula uma meta-variável a um conjunto de literais.
- **Renomear tipo qualificado**: Renomeia o tipo qualificado da especificação para o tipo indicado no refinamento
- **Adicionar evento**: Adiciona um novo evento à especificação base
- **Adicionar restrição**: Adiciona uma restrição (*constraint*) à especificação base
- **Adicionar requisito**: Adiciona um requisito (*require*) à especificação base
- **Adicionar *ensure*** : Adiciona um novo *ensure* à especificação base

A gramática que especifica os diferentes tipos de transformações a partir da aplicação de refinamentos é bastante simples e está definida no trecho de código a seguir:

```
RefinementOpr
  : {Rename} "rename" "spec" type = QualifiedName ";"
  | {AddEvent} "add" "event" event = Event ";"
  | {AddConstraint} "add" "constraint" constraint = ConstraintExp ";"
  | {AddRequire} "add" "require" require = RequirePredicate ";"
  | {AddEnsure} "add" "ensure" ensure = EnsurePredicate ";"
  | {DefineLiteralSet} "define" var = ID "=" set = LiteralSet ";"
  ;
```

A gramática de refinamentos é facilmente extensível, de modo que novos tipos de operação que venham a ser necessários podem facilmente ser incluídos na regra acima.

3.1.3 A gramática de configuração

De todas as três gramáticas que compõe Meta-CrySL, a gramática de configurações é a mais simples. De maneira geral, arquivos de configuração só precisam especificar um diretório de entrada, um diretório de saída e uma lista de módulos que devem ser carregados. Essa configuração é importante para compor os diferentes tipos de regras CrySL que podem ser geradas através das combinações entre especificações e refinamentos.

Um arquivo de configuração segue o seguinte modelo:

```
CONFIG ANDROID {
  inputDir  = "./path/to/inputDir" ;
  outputDir = "/path/to/outputDir" ;

  load "baseSpec.mcsl";
  load "baseRef.ref"
}
```

Como é possível ver, na configuração o usuário pode descrever uma série de módulos que devem ser carregados. Esses módulos podem ser ou especificações base na linguagem CrySL estendida (denominados pela extensão `.mcsl`, ou arquivos de refinamento (na extensão `.ref`). A gramática de configuração não passa de 10 linhas de código:

```
Configuration
  : "CONFIG" config = ID "{"
      'inputDir'  '=' inputDir  = STRING ";"
```

```
      'outputDir'  '=' outputDir  = STRING ";"
      modules += LoadModule+
  "}";

LoadModule
 :  "load" module = STRING ";" ;
```

3.1.4 Geração automatizada de testes

Finalizada a implementação da gramática, decidiu-se pela implementação de uma série de testes unitários responsáveis por verificar a corretude do analisador sintático. Essa decisão se baseia principalmente no princípio de avaliação de gramáticas em projetos de *grammarware* [8], o qual visa estabelecer métricas de completude e corretude em gramáticas.

Inicialmente os testes unitários foram escritos manualmente. Essa abordagem foi substituída por uma estratégia de geração automatizada de testes a partir de especificações MetaCrySL pré-existentes, decisão que também resulta da aplicação do princípio da automação em projetos de *grammarware*. A possibilidade de implementação de um gerador automático de testes se tornou possível devido à grande similaridade entre os vários testes necessários para o analisador sintático. Na maioria dos casos, esses testes deveriam apenas indicar que não ocorreram erros durante o processo de análise sintática do código fonte, de modo que o corpo das funções de testes seria basicamente o mesmo, alterando-se apenas o arquivo de configuração que seria carregado.

Dessa maneira, foi reunido um grande conjunto de regras CrySL usadas pelo CogniCrypt, as quais foram utilizadas como entrada para um gerador de testes automático. A esse gerador foi dado o nome **MCSLTestGeneration**. De modo a integrar o gerador de testes dentro do Meta-CrySL, optou-se que o gerador de testes fosse implementado como um plugin Maven, executando durante o próprio processo de ciclo de vida de uma *build*.

Para funcionar, o plugin necessita de dois parâmetros apenas: o caminho para um diretório que contenha os arquivos CrySL para os quais se deseja gerar testes unitários, e um caminho para um diretório de saída onde os testes serão gerados. Para geração automatizada de código foi utilizado o *Apache Freemarker* [9], uma *template engine* que facilita a geração de arquivos de texto baseados em templates.

A seguinte arquitetura de código foi utilizada: o plugin Maven recebe como parâmetro de entrada o caminho para um diretório que contém os arquivos CrySL de entrada. Para cada um desses arquivos, a classe `TestOjbect` é instanciada, recebendo como parâmetros o nome do arquivo CrySL e o seu caminho no sistema. A classe `TestObject` representa,

na memória, cada arquivo CrySL individual. O plugin cria então uma lista de objetos da classe `TestObject`, a qual é enviada para o Freemarker.

O Freemarker, ao receber a lista de objetos `TestObject`, utiliza como insumo um template escrito em um arquivo *.ftl*, o qual especifica a formatação do arquivo que deve ser gerado. A vantagem de se utilizar uma engine de templates tal como o Freemarker é que é simples e fácil trabalhar com geração de código.

O seguinte trecho de código ilustra o template utilizado pelo plugin **MCSLTestGeneration** para geração do arquivo de testes:

```
<#list rules as rule>
@Test
def void ${rule.name}() {
   val file = super.readFileIntoString(
   BR_UNB_CIC_METACRYSL_TESTS_FILES + "${rule.fileName}")

    val result = super.parseHelper.parse(file)
    Assert.assertNotNull(result)
    val errors = result.eResource.errors
    Assert.assertTrue(
    '''Unexpected errors: «errors.join(", ")»''', errors.isEmpty)
}
</#list>
```

Como é possível observar pelo código acima, o Freemarker recebe como parâmetro uma lista de objetos, aqui chamados de *rules*. É possível iterar através dessa lista através do operador `<# >`, e acessar os atributos desses objetos normalmente. O código gerado pelo Freemarker está na linguagem Xtend e utiliza as funções da biblioteca `JUnit`. Todos os testes possuem a mesma estrutura acima: o arquivo CrySL de entrada é lido numa *string* a partir do sistema de arquivos (utilizando-se o caminho especificado dentro do atributo `rule.fileName`), a qual é utilizada como entrada para o parser gerado pelo XText. Em seguida, verifica-se se o parser não retorna nenhum tipo de erro. Um teste unitário é gerado para cada arquivo CrySL de entrada.

A estrutura geral do plugin de geração de testes pode ser vista na imagem abaixo:

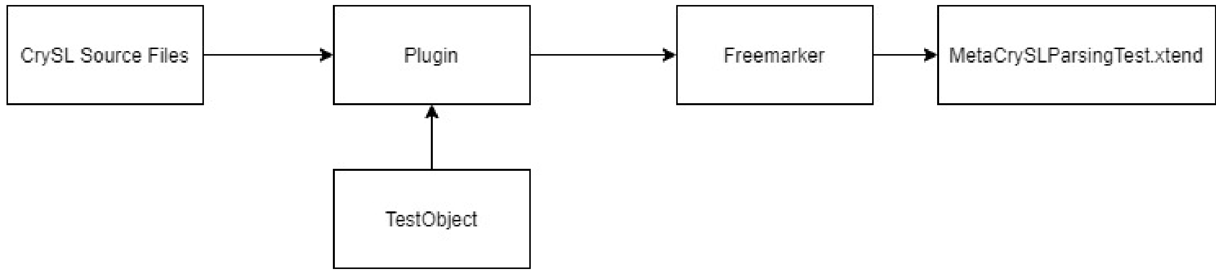

Figura 3.2: Arquitetura do gerador de testes unitários

A abordagem apresentada na Figura 3.2 permite que a suíte de testes para o Meta-CrySL seja constantemente ampliada de maneira simples, visto que há uma grande quantidade de especificações CrySL testadas e em uso diário, as quais podem ser acessadas facilmente no repositório do CogniCrypt. Utilizando-se essas especificações prontas, podemos verificar a corretude do parser sendo implementado, e podemos tornar esse processo de geração de testes automático, pois como se trata de um plugin Maven, a geração de testes irá ocorrer a cada ciclo de vida do projeto.

3.2 *MetaCrySL Preprocessor*

O pré-processador de Meta-CrySL reside no arquivo `MetaCrySLGenerator.xtend`, onde está implementada toda a lógica de pré-processamento (a qual consiste na aplicação dos refinamentos nas especificações).

O processo de geração de código ocorre em duas etapas: numa primeira fase, os arquivos de configuração são lidos e, em seguida, os arquivos de especificação e de refinamentos são carregados. A lógica de aplicação dos refinamentos é executada e como saída, são criados na memória objetos da classe `Spec`, os quais contém o código parseado de uma especificação CrySL já com os refinamentos aplicados. A segunda etapa se baseia em utilizar esse objeto `Spec` como entrada para um *prettyPrinter*, o qual efetivamente escreve o código CrySL gerado no sistema de arquivos. Nessa seção será apresentada a primeira parte, concernente a lógica de aplicação dos refinamentos a partir de arquivos de configuração.

O processo de geração de código se inicia a partir da chamada ao método `generateCode`, o qual recebe como parâmetro o caminho para um arquivo de configuração. A classe `MetaCrySLGenerator` também implementa o método `genericMetaCrySLParser`, o qual é um método genérico que realiza o *parse* de arquivos de configuração, especificações em CrySL e refinamentos. Para tal, O método `genericMetaCrySLParser` recebe tanto o caminho para o arquivo a ser parseado como também um objeto da classe `ModelType`,

o qual especifica o tipo de arquivo (entre configuração, refinamento e especificação) de modo que o parser correto possa ser escolhido.

O diagrama abaixo ilustra a arquitetura geral do esquema de geração de código na classe `MetaCrySLGenerator`:

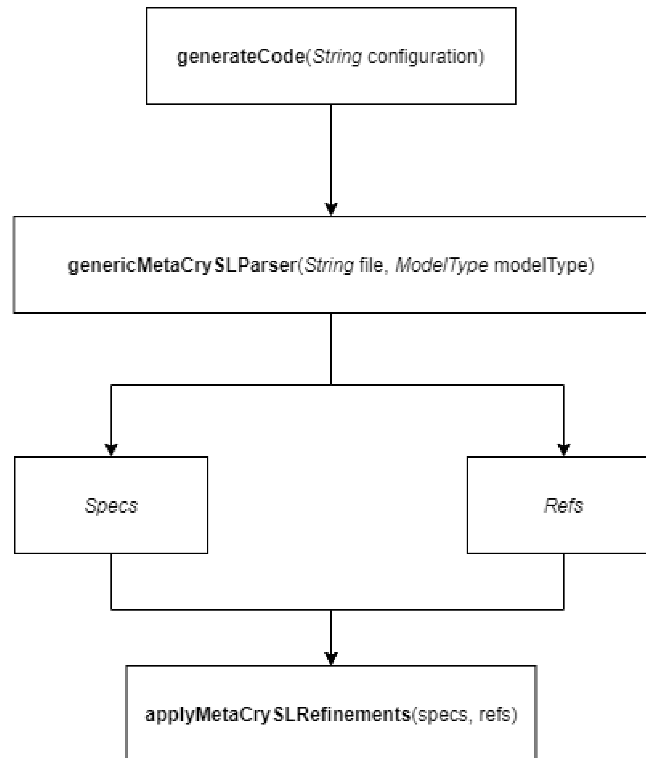

Figura 3.3: Arquitetura do gerador de código

Com a análise estático do arquivo de configuração realizada, os caminhos para os módulos são varridos em busca de arquivos de especificação ou de refinamentos. Caso o módulo carregado no arquivo de configuração seja um diretório, o diretório é aberto e todos os arquivos dentro dele são lidos. A extensão do arquivo é utilizada para checar se o arquivo que está sendo lido se trata de um arquivo de especificação ou de refinamento.

Para cada arquivo lido, o método `genericMetaCrySLParser` é chamado novamente, e o modelo retornado é guardado em um `Hashmap`. Para arquivos de especificação, o modelo é guardado num `Hashmap` chamado *specs*, o qual tem como chave o nome qualificado da classe Java a qual aquela especificação se aplica. Já arquivos de refinamentos são armazenados dentro do `Hashmap` *refs*, o qual tem como chave uma string indicando uma única especificação e como valor o modelo de um refinamento.

Antes dos refinamentos serem aplicados, é necessário que seja realizada uma fusão entre os diversos tipos de refinamentos que podem ser aplicados numa única especificação. O

método `mergeRefinements` da classe `MetaCrySLGenerator` serve especificamente para isso, como mostra o trecho de código a seguir:

```
def Refinement mergeRefinements(String name, Refinement ref1, Refinement ref2) {
    val visitor = new MergeRefinementVisitor(name)
    val refs = CollectionLiterals.newArrayList(ref1, ref2)

    for (ref : refs) {
        for (r : ref.refinements) {
            visitor.doSwitch(r)
        }
    }
    return visitor.refinement
}
```

O método `mergeRefinements` recebe dois refinamentos diferentes e gera uma instância da classe `MergeRefinementVisitor`, a qual constrói um novo objeto de refinamento que é composto pelas operações de refinamentos especificadas em cada um dos dois refinamentos recebidos. Essa classe, a qual herda da classe `MetaCrySLSwitch` que é gerada automaticamente pelo Xtext, utiliza o padrão *Visitor* para realizar essa fusão de refinamentos. O padrão *Visitor* é utilizado amplamente dentro do pré-processador da linguagem, Dentro dessa classe, são implementados os métodos `caseRename`, `caseDefineLiteralSet`, `caseAddConstraint`, `caseAddEnsure` e `caseAddEvent`, ou seja, um método para cada um dos tipos de operações possíveis em refinamentos. O padrão *Visitor* permite navegar facilmente entre esses refinamentos.

Com os dois *hashmaps* populados e um único refinamento final construído a partir do conjunto maior de refinamentos, o método `applyMetaCrySLRefinements` é chamado, recebendo cada um dos *hashmaps* como parâmetro. O método `applyMetaCrySLRefinements` é o responsável por associar cada especificação com um conjunto de refinamentos que devem ser nela aplicados. A classe `ApplyRefinementVisitor` também é instanciada nesse método.

A classe `ApplyRefinementVisitor`, a qual herda da classe `MetaCrySLSwitch` e que é gerada automaticamente pelo Xtext, também utiliza o padrão *Visitor* para realizar a aplicação dos refinamentos em cada modelo de especificação. No caso dessa classe específica, o construtor da classe recebe o modelo da especificação enquanto o método `doSwitch`, implementado pela classe-mãe, é chamado com uma única operação de refinamento. Uma nova especificação é montada com base nas operações de refinamento lidas, e um novo modelo de especificação é retornado pelo *Visitor*.

O seguinte trecho de código ilustra a lógica de aplicação dos refinamentos dentro do método `applyMetaCrySLRefinements`:

```
  val compiledSpecs = new ArrayList<Spec>

  if (refs.containsKey(k)) {
    val ref = refs.get(k)
    val applyVisitor = new ApplyRefinementVisitor(spec)
    for (r : ref.refinements) {
      applyVisitor.doSwitch(r)
    }
    compiledSpecs.add(applyVisitor.spec)
}
```

Como pode ser visto no código acima, a lista `compiledSpecs` reúne todos os modelos de especificação gerados a partir da lógica descrita acima. Cada um desses modelos já representa uma especificação CrySL completa, com os refinamentos aplicados e as meta-variáveis substituídas por valores terminais.

No pré-processador, o padrão *Visitor* é utilizado tanto para realizar a fusão do conjunto de refinamentos que se aplica a uma determinada especificação em um único refinamento, como também para aplicar esse refinamento final dentro da especificação. A etapa de aplicação dos refinamentos conclui o trabalho de manipulação da representação intermediária da linguagem, de modo que o próximo passo é gerar, a partir dessa representação intermediária, os arquivos CrySL finalizados. Para mais informações a respeito, é possível checar o código-fonte completo no seguinte link: *https://github.com/PAMunb/MetaCrySL*.

3.3 *MetaCrySL PrettyPrinter*

Com os modelos de especificação CrySL refinados, o passo restante é a geração de código propriamente dita através do módulo *MetaCrySL PrettyPrinter*. Os arquivos de configuração de Meta-CrySL especificam um diretório de saída onde o usuário espera que os arquivos gerados sejam escritos. No processo de parser do arquivo de configuração, o caminho para esse diretório é guardado.

O processo de *prety-printing* inicia-se com uma chamada ao método `compile` da classe `MetaCrySLGenerator`. Trata-se de um método simples, cuja única função é receber a lista de modelos de especificação gerados na etapa anterior, iterar por eles e, para cada um, gerar uma instância da classe `CodeWriter` e executar seu método `generate()`:

```
def void compile(List<Spec> specs) {
  for(spec: specs) {
    val code_writer = new CodeWriter(spec, this.outputDir)
      code_writer.generate()
    }
}
```

A classe `CodeWriter` é responsável por organizar o processo de *pretty-printing*. Como pode ser visto pelo trecho de código acima, essa classe recebe um modelo de especificação e o caminho para um diretório de saída.

O corpo do método `generate()` pode ser visto no trecho de código a seguir:

```
def void generate() {
  val pw = new PrintWriter(outputDir+ this.typeName + ".crysl", "UTF-8")
  pw.println(writeHeader())
  pw.println(writeObjects())
  pw.println(writeForbidden())
  pw.println(writeEvents())
  pw.println(writeOrder())
  pw.println(writeConstraints())
  pw.println(writeRequire())
  pw.println(writeEnsures())

  pw.close()
 }
```

O código da classe `CodeWriter` é bastante simples. O método `generate`, por exemplo, apenas instancia a classe Java `PrintWriter`, gerando um arquivo dentro do caminho de saída especificado, e depois escreve nesse arquivo executando os métodos responsáveis por gerar cada uma das cláusulas que compõe uma especificação CrySL. Tomando o método `writeEvents`, por exemplo:

```
def String writeEvents() {
  val events = new ArrayList<String>
  val visitor = new CodeWriterVisitor()
  events.add('\nEVENTS\n')

  for(event: spec.eventSpec.events) {
    val e = visitor.prettyPrintEvent(event)
```

```
    if(e !== null) {
      events.add('\t' + e + ';')
    }
  }

  return String.join('\n', events)
}
```

O método `writeEvents` itera dentro da lista de eventos especificada no atributo `spec`. Relembrando a definição da gramática, a gramática base possui um atributo `eventSpec`, o qual é definido pela seguinte regra:

```
EventSpec : {EventSpec} 'EVENTS' events += Event+ ;
```

Em Xtext, o operador `+=` indica a construção de uma lista dentro do atributo `events`. O método `writeEvents` instancia a classe `CodeWriterVisitor` e itera através dessa lista, passando cada evento para o método `prettyPrintEvent` da classe `CodeWriterVisitor`. Esse método retorna uma string, a qual é escrita no arquivo final.

O código para os outros métodos da classe `CodeWriter` é basicamente o mesmo, mudando apenas o método de `CodeWriterVisitor` que é chamado e qual atributo do `spec` é iterado.

3.4 Avaliação dos Resultados

Nessa seção será feita uma avaliação da implementação de Meta-CrySL apresentada. O objetivo principal desse estudo é verificar se os artefatos gerados pela nova versão de Meta-CrySL correspondem aos artefatos gerados pela implementação original. Para atingir esse objetivo, as seguintes questõs de pesquisa foram definidas:

1. A nova implementação consegue reconhecer arquivos de entrada Meta-CrySL que a versão original conseguia reconhecer? Aqui busca-se por problemas no analisador sintático da nova implementação.

2. A nova implementação aplica os refinamentos corretamente? A ideia dessa pergunta é verificar a corretude da nova implementação.

Para responder essas duas perguntas, as seguintes métricas foram estabelecidas:

1. Quantidade de arquivos gerados sobre o total de arquivos de especificação de entrada (como o Meta-CrySL só gera um arquivo de saída caso tenha conseguido passar pelo analisador sintático, essa métrica permite identificar qual a porcentagem de arquivos de entrada resultaram em erros no analisador sintático)

2. Quantidade de arquivos que não foram gerados corretamente (aqui serão detectados problemas no pré-processador ou no *PrettyPrinter* da nova implementação)

Para conduzir os experimentos, foi utilizada a base de código da implementação anterior em Rascal-MPL, a qual já possuía uma suíte de testes com uma ampla variedade de especificações Meta-CrySL, refinamentos e arquivos de configuração para a plataforma Android. Mais especificamente, foi escolhida a suíte de testes contendo especificações Meta-CrySL conforme as normas do padrão BSI. Dessa forma, seria possível comparar as saídas geradas pela implementação atual com as saídas geradas pela implementação Rascal-MPL de modo a responder as perguntas propostas acima.

Para a suíte de testes Android-BSI, foram construídos três arquivos de configuração. Cada um desses arquivos carrega o mesmo conjunto de especificações como base, mas carrega conjuntos diferentes de especificações a depender da versão do Android para os quais as regras de uso das API's criptográficas foram escritas.

Configuração
Android0108
Android0116
Android25plus

Tabela 3.1: Arquivos de configuração utilizados nos testes

Para esse conjunto específico de configurações, as regras de refinamento são carregadas de maneira cumulativa. Isso é, a configuração *Android0108* carrega apenas refinamentos que se aplicam para as versões Android entre 01 e 08, enquanto que a configuração *Android016* carrega todas aquelas carregadas por *Android0108* juntamente com todas aquelas entre 08 e 16, e assim por diante. Dessa maneira, é possível gerar, para a mesma base de especificações, regras específicas para versões diferentes das API's criptográficas.

Todos os arquivos de entrada para testes foram coletados a partir do repositório da implementação de Meta-CrySL em Rascal-MPL e organizados dentro da pasta *test-resources/android-bsi*, dentro da pasta de testes do projeto. Foram escritos testes para realizar a geração dos arquivos no seguinte formato:

```
@Test
def void test0108() {
    val config = URI.createURI("./test-resources/
        android-bsi/config/Android0108.config").path
    val generator = new MetaCrySLGenerator
    val specs = generator.generateCode(config)
    generator.compile(specs)
```

```
}
```

O teste acima carrega o arquivo de configuração *Android0108* e instancia *MetaCrySL-Generator*, invocando o método *generateCode* com o arquivo de configuração como parâmetro. O método *generateCode* retorna uma lista de modelos de especificação, os quais podem ser enviados para o método *compile* da classe.

O arquivo de configuração *Android018* é bastante simples: é estabelecido, nas linhas iniciais, o caminho para o diretório de entrada (que deve conter os arquivos Meta-CrySL e de refinamento), e também executados comandos *load*, os quais permitem carregar arquivos específicos ou mesmo diretórios inteiros contendo arquivos:

```
CONFIG Android0108 {
  inputDir  = "./test-resources/android-bsi/" ;
  outputDir = "./test-resources/android-bsi/target/mcsl/0108/" ;

  load "base/";
  load "0108/";
  load "01plus/";
}
```

No caso específico do arquivo *Android018*, estão sendo carregados o diretório *base/*, o qual contém todas as especificações base Meta-CrySL, assim como os diretórios *0108/* e *01plus/*, os quais contém arquivos de refinamentos. A partir dessa configuração, o programa deve ser capaz de gerar os arquivos CrySL refinados dentro do diretório especificado por *outputDir*.

Testes similares foram escritos também para os arquivos de configuração *Android0116* e *Android25plus*. Os resultados obtidos a partir da execução desses testes foram compilados nas tabelas A.1, A.2, A.3, as quais estão disponíveis no Apêndice A.

A partir dos dados coletados, foi possível perceber que, dos 32 arquivos de entrada utilizados para testes, 100% destes foram gerados pela nova implementação de Meta-CrySL a partir das três configurações utilizadas para testes (ou seja, um total de 96 arquivos foram gerados pelos testes). Isso quer dizer que essas especificações Meta-CrySL passaram com sucesso pelo novo analisador sintático gerado pelo Xtext a partir da gramática implementada.

Dos arquivos gerados para a configuração *Android0108*, 90% não apresentaram nenhum erro na aplicação dos refinamentos, ou seja, o conteúdo dos arquivos se assemelha aos arquivos CrySL gerados pela implementação original de Meta-CrySL em Rascal-MPL. Para a configuração *Android0116*, aproximadamente 85% dos arquivos não apresentaram nenhum tipo de erro na aplicação dos refinamentos. Essa porcentagem permaneceu a

mesma para a configuração *Android25plus*. De fato, os erros foram acumulados com os refinamentos carregados a partir de versões anteriores.

Capítulo 4

Conclusão

O projeto desenvolvido ao longo desse trabalho de graduação resultou numa implementação da linguagem Meta-CrySL a partir do *Xtext*. Dentre os aprendizados envolvidos, foi necessário um grande estudo dentro da área de implementação de linguagens de modo geral, baseado fortemente nos conceitos abordados no livro *Language Implementation Patterns*, de Terence Parr [1], além dos princípios de *grammarware* apresentados no artigo *Towards an Engineering Discipline for GRAMMARWARE* [8]. Fora isso, foi também necessário um entendimento aprofundado da própria linguagem CrySL e da arquitetura de alto-nível de Meta-CrySL.

A nova implementação traz toda a gramática de Meta-CrySL para dentro do ambiente *Xtext*, trazendo consigo uma suíte de testes automatizados construída através do plugin de geração de código, *MCSLTestGeneration*, mencionado no capítulo anterior. Ao longo de todo esse trabalho tomou-se muito cuidado em seguir os princípios para *grammarware* [8] com o foco na manutenibilidade e reuso. Dessa maneira, a gramática desenvolvida para Meta-CrySL aqui é facilmente extensível.

A nova implementação foi testada com base numa suíte de testes construída para geração de especificações CrySL para Android-BSI. As configurações para essas especificações foram escritas a partir da suíte de testes implementada para a versão de Meta-CrySL em Rascal-MPL. A partir desses testes, foi possível perceber que o analisador sintático construído a partir da nova gramática funciona corretamente. Um total de 100% dos arquivos lidos teve sua análise sintática concluída com sucesso. Dos 96 arquivos gerados em testes, aproximadamente 86% não apresentou nenhum tipo de divergência com relação às mesmas especificações geradas pela versão anterior. A partir de uma investigação mais detalhada dos arquivos que apresentaram divergências, percebeu-se que a maior parte dos problemas encontrava-se na aplicação dos refinamentos dentro da cláusula `CONSTRAINTS` ou `REQUIRES`.

A partir dos erros observados na geração de arquivos, pode-se concluir que um caminho para melhorias futuras dentro desse trabalho estaria relacionado com testes mais abrangentes da corretude de aplicação dos refinamentos. Atualmente a suíte de testes de Meta-CrySL está mais voltada para testes do analisador sintático (também chamados de testes de *parsing*) ou testes de integração completos, os quais passam por todo o *pipeline*, assegurando que nenhum tipo de erro grave ocorreu. De maneira a melhorar a qualidade do trabalho e a confiabilidade da nova implementação, uma sugestão de caminho futuro seria um processo mais rigoroso de checagem da aplicação dos refinamentos, que se trata de um ponto-chave de Meta-CrySL. Juntamente com isso, um trabalho de verificação de corretude do próprio resultado da aplicação dos refinamentos seria adequado. A arquitetura atual de Meta-CrySL não se preocupa se, por exemplo, refinamentos contraditórios são incluídos numa mesma especificação CrySL compilada. Um passo interessante futuro seria um processo de análise individual dos refinamentos, reportando possíveis contradições para o usuário.

Referências

[1] Parr, Terence: *Language Implementation Patterns Create Your Own Domain-specific and General Programming Languages.* Oreilly and Associates Inc, 2018. ix, 3, 9, 10, 11, 32

[2] Nadi, S., S. Krüger, M. Mezini e E. Bodden: *"jumping through hoops": Why do java developers struggle with cryptography apis?* páginas 935–946, 2016. 1, 5

[3] Krüger, S., J. Späth, K. Ali, E. Bodden e M. Mezini: *Crysl: An extensible approach to validating the correct usage of cryptographic apis.* IEEE Transactions on Software Engineering, páginas 1–1, 2019. 1, 6

[4] Krüger, S., S. Nadi, M. Reif, K. Ali, M. Mezini, E. Bodden, F. Göpfert, F. Günther, C. Weinert, D. Demmler e R. Kamath: *Cognicrypt: Supporting developers in using cryptography.* páginas 931–936, 2017. 1, 6

[5] Bettini, Lorenzo: *Implementing domain-specific languages with Xtext and Xtend: learn how to implement a DSL with Xtext and Xtend using easy-to-understand examples and best practices.* Packt Publishing, 2016. 2, 3, 11, 14, 15

[6] PAMunb: *Metacrysl.* `https://github.com/PAMunb/MetaCrySL`, 2020. 4

[7] Acar, Y., M. Backes, S. Fahl, S. Garfinkel, D. Kim, M. L. Mazurek e C. Stransky: *Comparing the usability of cryptographic apis.* páginas 154–171, 2017. 5

[8] Klint, Paul, Ralf Lämmel e Chris Verhoef: *Towards an engineering discipline for grammarware.* ACM Trans. Softw. Eng. Methodol., 14, 2003. 12, 21, 32

[9] *What is apache freemarker™?*, Aug 2020. `https://freemarker.apache.org/`. 21

Apêndice A

Dados Coletados

Especificação	Arquivo gerado	Arquivo não apresenta erros
AlgorithmParameters	✓	✗
CertPathTrustManagerParameters	✓	✓
Cipher	✓	✗
CipherInputStream	✓	✓
CipherOutputStream	✓	✓
DHGenParameterSpec	✓	✓
DSAGenParameterSpec	✓	✓
DigestInputStream	✓	✓
DigestOutputStream	✓	✓
GCMParameterSpec	✓	✓
HMACParameterSpec	✓	✓
IvParameterSpec	✓	✓
Key	✓	✓
KeyGenerator	✓	✓
KeyManagerFactory	✓	✓
KeyPair	✓	✓
KeyPairGenerator	✓	✓
KeyStore	✓	✓
KeyStoreBuilderParameters	✓	✓
Mac	✓	✓
MessageDigest	✓	✓
PBEKeySpec	✓	✓
PBEParameterSpec	✓	✓
PKIXBuilderParameters	✓	✓
PKIXParameters	✓	✓
RSAKeyGenParameterSpec	✓	✓
SSLContext	✓	✓
SecretKey	✓	✗
SecretKeyFactory	✓	✓
SecretKeySpec	✓	✓
SecureRandom	✓	✓
Signature	✓	✓

Tabela A.1: Resultados de testes para *Android0108*

Especificação	Arquivo gerado	Arquivo não apresenta erros
AlgorithmParameters	✓	✗
CertPathTrustManagerParameters	✓	✓
Cipher	✓	✗
CipherInputStream	✓	✓
CipherOutputStream	✓	✓
DHGenParameterSpec	✓	✓
DSAGenParameterSpec	✓	✓
DigestInputStream	✓	✓
DigestOutputStream	✓	✓
GCMParameterSpec	✓	✓
HMACParameterSpec	✓	✓
IvParameterSpec	✓	✓
Key	✓	✓
KeyGenerator	✓	✓
KeyManagerFactory	✓	✓
KeyPair	✓	✓
KeyPairGenerator	✓	✗
KeyStore	✓	✓
KeyStoreBuilderParameters	✓	✓
Mac	✓	✓
MessageDigest	✓	✓
PBEKeySpec	✓	✗
PBEParameterSpec	✓	✓
PKIXBuilderParameters	✓	✓
PKIXParameters	✓	✓
RSAKeyGenParameterSpec	✓	✓
SSLContext	✓	✓
SecretKey	✓	✗
SecretKeyFactory	✓	✓
SecretKeySpec	✓	✓
SecureRandom	✓	✓
Signature	✓	✓

Tabela A.2: Resultados de testes para *Android0116*

Especificação	Arquivo gerado	Arquivo não apresenta erros
AlgorithmParameters	✓	✗
CertPathTrustManagerParameters	✓	✓
Cipher	✓	✗
CipherInputStream	✓	✓
CipherOutputStream	✓	✓
DHGenParameterSpec	✓	✓
DSAGenParameterSpec	✓	✓
DigestInputStream	✓	✓
DigestOutputStream	✓	✓
GCMParameterSpec	✓	✓
HMACParameterSpec	✓	✓
IvParameterSpec	✓	✓
Key	✓	✓
KeyGenerator	✓	✓
KeyManagerFactory	✓	✓
KeyPair	✓	✓
KeyPairGenerator	✓	✗
KeyStore	✓	✓
KeyStoreBuilderParameters	✓	✓
Mac	✓	✓
MessageDigest	✓	✓
PBEKeySpec	✓	✗
PBEParameterSpec	✓	✓
PKIXBuilderParameters	✓	✓
PKIXParameters	✓	✓
RSAKeyGenParameterSpec	✓	✓
SSLContext	✓	✓
SecretKey	✓	✗
SecretKeyFactory	✓	✓
SecretKeySpec	✓	✓
SecureRandom	✓	✓
Signature	✓	✓

Tabela A.3: Resultados de testes para *Android25plus*