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

Avaliação de uma implementação de MetaCrySL em MPS

Danilo Santos Sales

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

Orientador
Prof. Dr. Rodrigo Bonifácio de Almeida

Brasília
2022

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

Avaliação de uma implementação de MetaCrySL em MPS

Danilo Santos Sales

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

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

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

Brasília, 14 de maio de 2022

Dedicatória

Dedico este trabalho aos meus pais, e a minha vó Joanita, os quais o amor e dedicação possibilitaram essa conquista.

Agradecimentos

Agradeço primeiramente a Deus, por me dar a oportunidade de ter cursado esta graduação, aos meus pais pela sua dedicação e sacrificio, ao fornecerem uma melhor educação, qualidade de vida e principalmente, tempo. Agradeço ao meu orientador pela paciência ao longo do desenvolvimento deste trabalho, devido aos empecilhos e dificuldades que surgiram.

Resumo

O uso de APIs criptográficas exige um conhecimento especializado, que guia a escolha do algoritmo e de seus parâmetros. A má utilização desses algoritmos pode causar inúmeros graves problemas. Com objetivo de mitigar o mau uso, foi criada a ferramenta CogniCrypt, que a partir de arquivos de especificação - arquivos CrySL - faz uma análise do código, e são destacados os pontos em que ocorre um mau uso. Entretanto o reuso de arquivos de especificação CrySL se mostra um desafio devido a grande variabilidade presente, seja por diferença de versões entre sistemas operacionais ou devido a existência de diferentes variações de um mesmo algoritmo. A implementação do pipeline MetaCrySL em MPS, soluciona esse problema de reuso, a partir de uma ferramenta que permite ao especialista criar os arquivos de especificação para os algoritmos, expressando suas variações e especificidades, com suporte a edição rica - recursos de edição típico de IDE - e que ao final gera os arquivos de especificação CrySL.

Palavras-chave: CogniCrypt, DSL, MetaCrySL, MPS, trabalho de conclusão de curso

Abstract

The use of cryptographic APIs requires specialized knowledge, which guides the choice of the algorithm and its parameters. The misuse of these algorithms can cause numerous serious problems. In order to mitigate misuse, the CogniCrypt tool was created, which from specification files - CrySL files - makes an analysis of the code, and the points in which misuse occurs are highlighted. However, the reuse of CrySL specification files is a challenge due to the great variability present, either due to different versions between operating systems or due to the existence of different variations of the same algorithm. The implementation of the MetaCrySL pipeline in MPS solves this reuse problem, starting from a tool that allows the specialist to create the specification files for the algorithms, expressing their variations and specificities, with support for rich editing - typical IDE editing resources - and that in the end generates the CrySL specification files.

Keywords: CogniCrypt, DSL, MetaCrySL, MPS, thesis

Sumário

Lista de Figuras

Capítulo 1

Introdução

A utilização de APIs criptográficas exigem um conhecimento mais específico e uma maior atenção, pois para que a informação seja criptografada, é necessário conhecimento das premissas que o algoritmo requer; A criação incorreta de algum valor ou objeto, a ser usado como parâmetro, ou chamadas incorretas de um método podem adicionar vulnerabilidades.

Dentre as APIs criptográficas de diferentes linguagens, a *Java Criptography Architecture* (JCA) é a *Application Programming Interface* (API) mais popular entre os desenvolvedores da linguagem Java [1], a qual é parte da API de segurança nativa de linguagem. Embora a JCA proveja abstrações, estas focam-se mais em aspectos de design e interoperabilidade, continuando assim sob encargo do desenvolvedor o uso correto da API. Estudos recentes apontam que desenvolvedores frequentemente usam as APIs de criptografia de forma incorreta [2].

Com o intuito de auxiliar o desenvolvedor, e mitigar as vulnerabilidades geradas pelo uso incorreto de APIs criptógráficas, o ferramenta CogniCrypt - disponível para uso na IDE Eclipse - faz a análise estática do código, em que verifica o uso correto das APIs, e também oferece ferramentas de geração de código a partir de casos de uso pré-estabelecidos. Essas funcionalidades estão disponíveis não apenas à bilbioteca JCA, como também a outras APIs criptográficas disponíveis na plataforma Java.

Para efetuar a análise estática sobre o código, o plugin CogniCrypt utiliza arquivos de especificação, escritos com uma *Domain Specific Language* (DSL), que contém como determinado algoritmo da API pode ser utilizado. A DSL foi projetada de modo a facilitar a especificação da correta utilização da API, por parte dos especialistas em segurança. Entretanto ao elaborar os arquivos de descrição as API criptográficas variam bastante, tanto entre diferentes APIs, em que algumas podem ser mais flexíveis do que outra, como também apresentar variabilidade entre diferentes SOs, e entre outra possibilidades. O que gera a necessidade de múltiplos arquivos de especificação, onde o "núcleo"da especificação

acaba por se repetir entre os arquivos.

Para superar esse empecilho da variabilide e reuso na linguagem CrySL, foi proposto o MetaCrySL [3]. O MetaCrySL tinha como requisito inicial o suportar as fontes de variabilidade mais comuns nas APIs criptográficas e que ela fosse compilável para CrySL, de forma que não fosse necessário allterar o ecossitema CogniCrypt.

A primeira implementação da linguagem MetaCrySL ocorreu sob a linguagem Rascal-MPL, a qual apresentou dificuldades de incorporação no ecosistema CogniCrypt, sobretudo devido a diferença de linguagem - a base de código do CogniCrypt é escrita em Java. Diante dessa restrições foi feita uma nova implementação da linguagem sob o framework XText, o qual tem integração facilitada com CogniCrypt e com o Eclipse. O próprio framework XText é base para a implementação da linguagem CrySL.

1.1 Problema Explorado Nesta Monografia

Embora tenha-se atingido uma implementação estável, e interoperável dentro da plataforma em que o CogniCrypt é desenvolvido, a solução desenvolvida em XText não é de simples compreensão a um iniciante em modelagem de DSL, devido a necessidade de aprendizado da gramática, entender quais padrãos correspondem a um determinado *token* e também há restrições quanto à customização e suporte de diferentes tipos de notação para o usuário usuário especialista. Diante desses obstáculos, optou-se por investigar a implementação da linguagem MetaCrySL em outra ferramenta, o *workbench* MPS - um workbench para desenvolvimento de DSL da empresa Jetbrains - por ser tida como uma ferramente onde o desenvolvimento da DSL seria mais natural, a um programador iniciante na implementação de DSLs.

1.2 Objetivos

1.2.1 Objetivo Geral

O objetivo principal deste trabalho é implementar a linguagem MetaCrySL, na *workbench* MPS, analisar a expressividade da *workbench*, e como ela pode reduzir o tempo de aprendizado e especificação de uma DSL.

1.2.2 Objetivos Específicos

Para atingir o objetivo geral deste trabalho, foram mapeados os seguintes objetivos:

1. Compreender a tecnlogia MPS, para especificar uma linguagem e desenvolver meta-programas.

2. Implementar a linguagem MetaCrySL em MPS, reportando a experiência de uso da tecnologia.
3. Evidenciar a diferença entre o modelo de implementação de DSLs no MPS, em contraste com o Rascal e XText.
4. Avaliar a implementação de MetaCrySL em MPS, comparando com resultados obtidos nas outras tecnologias.

Capítulo 2

Revisão e trabalhos Relacionados

Previamente a exposição do que foi desenvolvido neste trabalho, é necessário o conhecimento de conceitos e definições, sobre as quais foi baseado, como também do trabalhos anteriores que auxiliaram no desenvolvimento e também serviram de comparação em outra seções deste trabalho.

2.1 Software Language Workbench

Conforme definido por Martin Fowler, em seu artigo [4], uma *language workbench* tem como características:

1. O usuário poder livremente definir linguagens, as quais são completamente integradas entre si.
2. A fonte primária de informação é uma representação persistente abstrata.
3. A *DSL* é definida a partir de três elementos principais: *schema*, *editor* e *generator*.
4. O usuário da linguagem manipula a *DSL* através de um editor projecional.
5. A *language workbench* pode armazenar informações incompletas ou contraditórias.

2.2 Linguagem CrySL

Com o intuito de auxiliar desenvolvedores na escrita de código criptográfico, devido as dificuldades inerentes [1], foi desenvolvida uma ferramenta de análise de código estática, o *plugin* CogniCrypt. Para especificar o que deve ser analisada e como é a correta utilização das APIS criptográficas, foi desenvolvida a DSL CrySL. [5]

2.3 MetaCrySL

A linguagem CrySL, trouxe suporte ao uso das APIs criptográficas. Entretanto, há o problema da variabilidade das APIs criptográficas, dentre alguns deles estão o sistema operacional e a versões diferentes do algoritmo escolhido. Para resolver esse problema foi desenvolvida a arquitetura de alto nível MetaCrySL [3], a qual consiste de:

1. Uma linguagem CrySL estendida, denominada SPEC
2. Uma linguagem de refinamentos, denominada REFINEMENT
3. Uma linguagem de configuração, denominada CONFIG

2.4 Trabalho correlatos

2.4.1 Solução em Rascal

Inicialmente a linguagem MetaCrySL, foi implementada em RASCAL-MPL, mas como dito em [6], a adoção de Rascal-MPL é dificil de ser adotada, por não se integrar com o ecossistema já existente do plugin CogniCrypt.

2.4.2 Solução em Xtext

A implementação em Xtext, foi a segunda implementação de MetaCrysl, feita posteriomente após analisados as dificuldade presente na solução em Rascal-MPL; Sendo esta já bem integrada, inclusive sendo escrita na mesma linguagem, e na mesma IDE, que o CogniCrypt, respectivamente, Xtend e Eclipse. [6]

Capítulo 3

Desenvolvimento de uma Linguagem em MPS

3.1 Edição Projecional

Antes de iniciar a demonstração de como especificar uma linguagem em MPS, é necessário compreender que a edição do arquivos não ocorrer de maneira habitual - a edição textual. A interação com os arquivos ocorre sobre uma abstração, a edição projecional; Na qual a exibição do arquivo não necessariamente corresponde ao formato em que o mesmo é armazenada, mas ocorre uma transformação onde o conteúdo é exibido da maneira especificada, assim é possivel especificar na workbench, uma experiência de interação melhor adaptada à necessidade, ao campo de atuação do usuário especialista, que poderia um editor de texto comum, ou um editor especializado, como por exemplo um que permitisse a escrita de equações químicas, com a mesma aparência vista no livros.

Um arquivo de defingção MetaCrySL, que define as regras de uso da classe KeyGenerator, possui a aparência presente na figura Figura 3.1 .

Com a edição projecional, não há a necessidade de construir uma gramática e um parser, que seriam utilizadas para construção da AST. Conforme há interação do programador com a *workbench* vai sendo construída a AST correspondente ao programa está sendo escrito, sendo esse processo totalmente transparente. A AST construida é então armazenada dentro de um arquivo XML, com a sintaxe especializada para a ferramenta Figura 3.3. Na figura Figura 3.2 é exibida a estrutura da AST gerada para o arquivo KeyGenerator.

O uso da *workbench* MPS será demonstrada através de implementação da linguagem MetaCrySL[3].

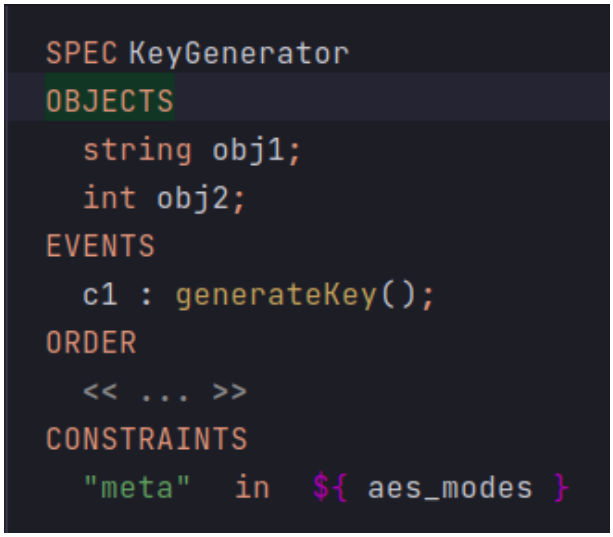

Figura 3.1: Aparência de um editor projecional modelado para MetaCrySL.

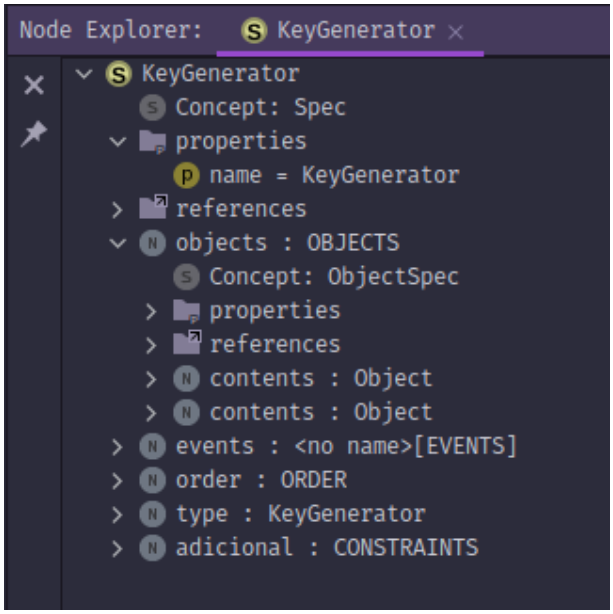

Figura 3.2: Aparência de um editor projecional modelado para CrySL.

3.2 Linguagem SPEC

Para definir da estrutura da linguagem SPEC, como dito anteriormente, não é especificada uma gramática a ser usada por um *parser*; Logo a estruturação da AST, tem de ser definidas usando outro conceitos; Conceitos estes presentes na *Structure Language* do MPS.

Um arquivo MetaCrySL em MPS, é que é chamado de *Model*, que é uma coleção de nós de diferentes "tipos", denominados *Concept*.

Figura 3.3: Arquivo XML que armazena os dados que especificam KeyGenerator.

3.2.1 Concept

Um *Concept* define quais propriedade um nó terá, quais "tipos"de nó poderão ser adicionados como nó filho - na seção children - e quais nós já existentes podem ser referenciados nele.

Para definir o arquivo base, onde será adicionada as informações pelo usuário, é definido um *Concept* Figura 3.4 em que sua instância será a raiz - *root*, pois isto que permite que seja criado um arquivo. A propriedade ABSTRACT define se aquele arquivo MetaCrySL é *abstract*, dentro os nós filhos:

1. O nó `type` define sobre qual classe a especificação será aplicada
2. O nó *ObjectSpec*, define um bloco que representa a definição do objeto que farão parte da especificação
3. O nó *EventSpec*, define um bloco que representa a definição dos métodos que farão parte da especificação
4. O nó *OrderSpec*, define um bloco que representa a definição da ordem em que os métodos poderão ser aplicados
5. O nó adicional, define um bloco onde poderão ser adicionado outros blocos de restrições - CONSTRAINS, FORBIDDEN,NEGATES, ENSURES e REQUIRES.

Com a definição dos nós filhos, é delineada parcialmente a estrutura da AST, pois os nós filhos do *Concept* SPEC, também poderão ter outros nós filhos.

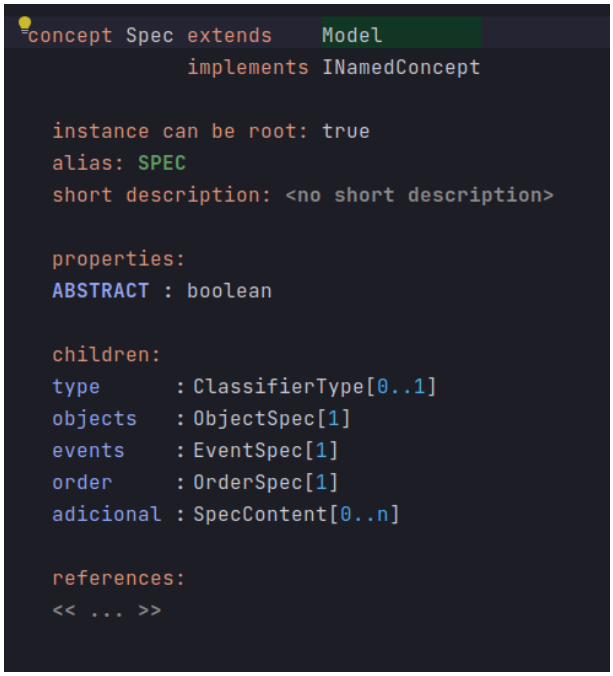

Figura 3.4: Aparência de um editor projecional modelado para CrySL.

O campo type foi escolhido como nó filho, pois definir ao este campo como um *Concept*, estão diponíveis mais recursos e comportamento especializados; Assim ao invés de inserir uma *string* para representar a classe a ser monitorada, pode se referenciar diretamente classes já existente no Java, como também fazer o uso de generics. Enquanto que os campo *property*, permitem apenas a inserção do valor primitivo.

Para permitir um variabilidade de nós, que podem ser criado em um campo, usado o recurso de polimorfismo, mais especificamente de interface, podendo isso ser observado mais especificamento no campo *adicional*, onde podem ser adicionado vários campos do tipo *SpecContent*, sendo que a mesma é uma inteface, logo todos os nós em que suas classes à implementem podem ser usadas nesse campo.

Onde os tipos type, ObjectSpec, EventSpec, OrderSpec são também um *Concept*, e optou-se por criá-los por isso propiciar a manutenbilidade bem como o encapsulamento e reuso.

3.2.2 ObjectSpec

O campo *objects* aceita um nó do tipo *ObjectSpec*. O ObjectSpec possui um alias e um único campo, onde são instanciados nós que implemetem a interface do tipo *IObjects-Content*, Figura 3.15. O *alias* quando definido, auxilia na inserção do nó a árvores, pois aparece com esse nome no *autocomplete*. Neste projeto apenas o *Concept* Object imple-

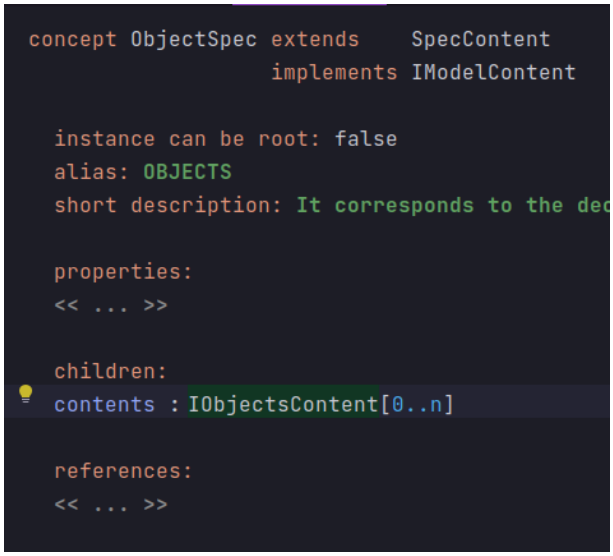

Figura 3.5: Declaração do Concept ObjectSpec.

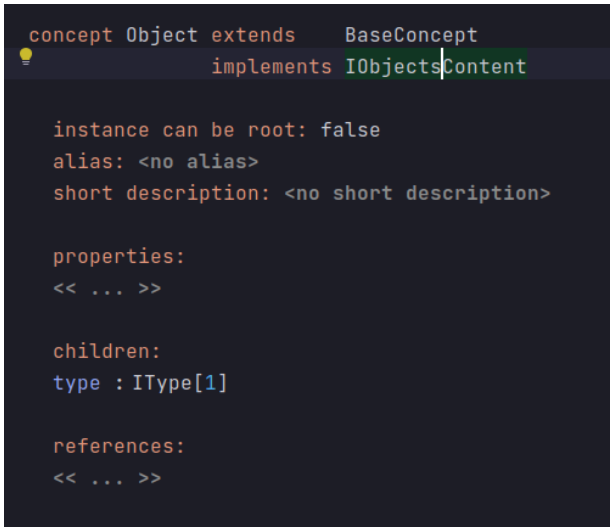

Figura 3.6: Declaração do Concept Object.

menta essa interface. Possui os campos *name*, que guarda identificador da variavel e o type que guarda o tipo da variável. O campo type é do tipo IType, por essa ser uma interface provida pelo MPS, que nos dá acesso a todos os tipos que implementem esse interface, como primitivos e as classes base do Java, que implemetam essa interface, facilitando o acesso a elas, para a definição do tipo do object.

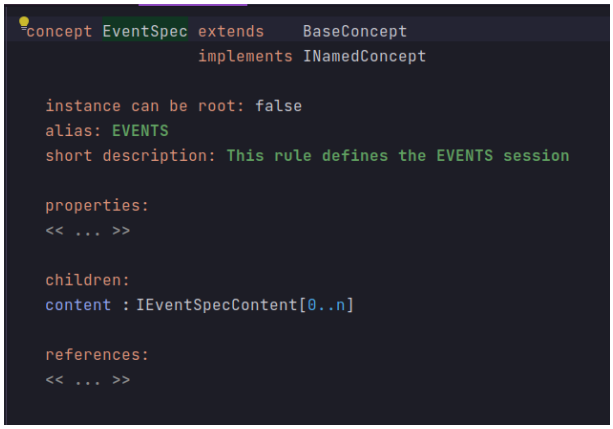

Figura 3.7: Declaração do Concept EventSpec.

3.2.3 EventsSpec

O campo events aceita um nó do tipo EventSpec, definido de maneira similar ao campo objects, possuindo um name, um alias e um campo content, sendo esse do tipo IEventSpecContent, Figura 3.7. Os *Concept* que implementam essa interface são EventMethod e EventAggregate.

Figura 3.8: Declaração de EventMethod.

Em EventMethod, Figura 3.8, foram definidos os campos alias e o campo method. O campo alias tem como valor o símbolo do dois pontos, únicamente definido para esse Concept, onde ao digitar esse símbolo no nó atual e ativar o *code completion* ele faz a substituição do texto pelo nó respectivo e o inseri na árvore. O campo method é do tipo MethodDef, onde efetivamente será feita a declaração da assinatura. Foi definido dessa

maneira, pois essa estrutura também é necessária na linguagem REFINEMENT. Então para possibilitar o reuso, essa parte foi declarada num outro Concept à parte.

```
concept EventAggregate extends    BaseConcept
                       implements IEventSpecContent

  instance can be root: false
  alias: :=
  short description: <no short description>

  properties:
  << ... >>

  children:
  aggregate : AggregateList[0..1]

  references:
  << ... >>
```

Figura 3.9: Declaração de EventAggregate.

Em EventAggregate, foi feita uma definição similar a EventMethod, declarando o campo alias, e o campo aggregate, do tipo AggregateList.

3.2.4 OrderSpec

```
concept OrderSpec extends    BaseConcept
                  implements IModelContent

  instance can be root: false
  alias: ORDER
  short description: Uses a regular expression

  properties:
  << ... >>

  children:
  content : IOrderSpecContent[0..n]

  references:
  << ... >>
```

Figura 3.10: Declaração de OrderSpec.

Similar ao que foi feito com ObjectSpec e EventSpec, no OrderSpec, Figura 3.10, também foi definido um campo alias, e um campo content em que podem ser adicionados

nós filhos que implementem a interface IOrderSpecContent. Implemetam essa interface os Concepts, ChoiceExp e SequenceExp.

3.2.5 ConstraintSpec

Os *Concept* EnsuresSpec, RequiresSpec, ForbiddenSpec, NegatesSpec, são estruturados de maneira semelhante ao ConstraintSpec, pois as expressões usadas nesses *Concept* são ao mesmas segundo a implementação em Xtext. Levando isso em conta, mas também que pode futuramente haver um diverncia quanto a isso, foi construida uma hieraquia em as expressões herdam de ConstraintExp e ConstraintExp herda de IConstraintContent, pois se houver mudança na hierarquia entre esse blocos, isso não afetaria a escolha de nós filgo para ConstraintSpec.

No *Concept* ConstraintSpec, Figura 3.15, foram definidos apenas os campos content, onde pode ser adicionado nós que implementem IConstraintContent, por consequência qualquer nó que implemente ConstraintExp pode ser adicionado como nó filho e o campo alias.

```
concept ConstraintSpec extends    SpecContent
                       implements <none>

  instance can be root: false
  alias: CONSTRAINTS
  short description:
      Defines constraints for objects defined under

  properties:
  << ... >>

  children:
  content : IConstraintContent[0..n]

  references:
  << ... >>
```

Figura 3.11: ConstraintSpec Concept.

Dentre os nós filhos que implementam a interface ContraintExp, há RelationalExp, Figura 3.15, BooleanExp, Figura 3.13, e ArithmeticExp , Figura 3.14, sendo estes *abstract concept*, não podem ser instanciados diretamente, mas funcionam como formas para os *Concept* que o implementarem. Foi escolhido essa abordagem pois conforme indica a BNF, há muita expressões de cada um desse tipos, em que o que muda na expressão é apenas o operador, que é defino nos nós filhos. Isso é ainda mais útil quando é implemetando o *Editor*, pois podemos reaproveitar o que foi definido para o *Concept* pai.

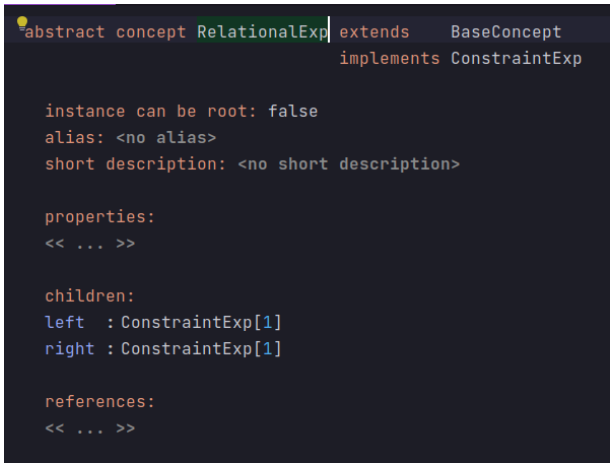

Figura 3.12: Concept RelationalExp.

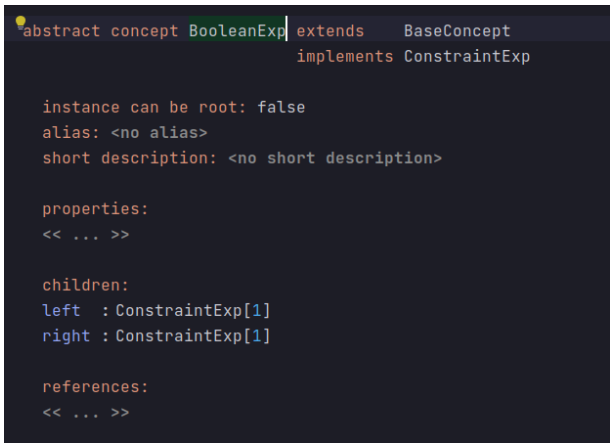

Figura 3.13: Concept BooleanExp.

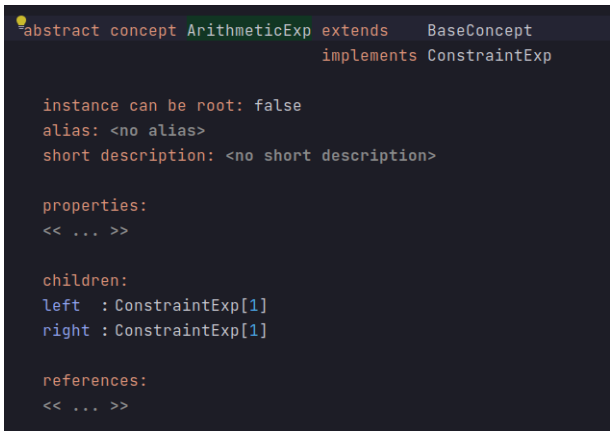

Figura 3.14: Concept ArithmeticExp.

As expressões que herdam de RelacionalExp são aqueles relacionados há comparações entre dois operados, sendo criados os *Concept* Equal, GreaterOrEqual, GreaterThan, LessOrEqual, LessThan e NotEqual.

As expressões que herdam de BooleanExp são aqueles que estabelem uma relação lógica entre os operandos, sendo criados os *Concept* ConjunctionExp, DisjunctionExp e ImpliesExp.

As expressões que herdam de ArithmeticExp são aquelas que estabelem uma relação aritmética entre os operandos, sendo criados os *Concept* MinusExpression e PlusExpression.

3.2.6 Editor

Definida a estrutura da raiz da AST, e quais valores é possivel atribuir ao seus campos, entretanto, ainda não está claro como o usuário irá interagir com o que foi definido anteriormente,. A interação do usuário com o *MPS*, é implementada com a estrutura *Editor*. Inicialmente quando não há nenhum *Editor* definido para o respectivo *Concept* o MPS gera um editor padrão com base nos campos definidos - o *reflective editor* Figura 3.15.

No *reflective editor* estão disponíveis apenas a edição padrão, através de autocomplete e sugestões providas pela *workbench*. Em MPS, um *Editor* "consiste de células, em que elas também contém outras celulas, sendo estas células, um texto, ou algum componente de interface"[7]
Para aproximar a projeção dos nós, a algo semelhante a um editor de texto típico, ao interagir com arquivo MetaCrySL, foi definido um editor para Spec, bem como para os outros *Concept*. Partindo daquilo que se percebe como comportamento tipico de um editor de texto, ao interagir com a *workbench* o usuário pode entrar com um texto que definem multiplas declarações e estas precisam estar organizadas na vertical. O MPS provê algumas células básicas, que auxiliam a implementação desse comportamento,*collection cell vertical* e a *collection cell horizontal*. A *collection cell vertical* indicada pelo simbolos [/ e /] agrupa e exibe as células na vertical, enquanto que a *collection cell horizontal* indicada pelos simbolo [> e <] agrupa e exibe células na horizontal. Para implementar a noção de linha editável, foi criado um *Concept* chamado EmptyLine, que não possui campos definidos, mas um comportamento bem especifico, de ser possível editar o seu conteúdo, adicionado um texto arbitrário e poder usar o conteúdo desse texto como argumento para ativar o menu de substituição, que substituiria o EmptyLine por um nó do tipo escolhido.

Agora para integrar todos esses processos, e ter como resultado a experiência almejada, nos *Concept* de nível mais alto, que agrupam outros concepts - Spec, ObjectSpec, EventSpec, OrderSpec - tiveram seu *Editor* definidos usando uma *collection cell vertical*, onde foi definido no Inspector, e que deve ocorrer a criação de um nó filho do tipo EmptyLine, ao

Figura 3.15: Aparência do reflective editor.

ser instanciado um nó do respectivo *Concept*, e os nós filhos ficam responsáves por como eles iriam exibir seu conteúdo, tipicamente usando uma *collection cell horizontal*. Com esses elementos, obtem-se uma experiência que lembra bastante a edição textual típica.

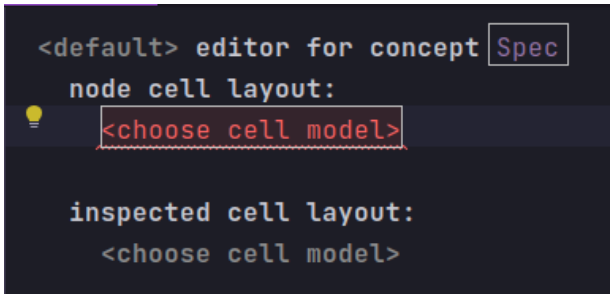

Figura 3.16: Aparência de um Editor sem células definidas.

Ao final da célula corrente, ao presionar a tecla Enter, é criado um nó do tipo EmptyLine, que representa uma nova linha vazia editável. Um EmptyLine é um Concept sem

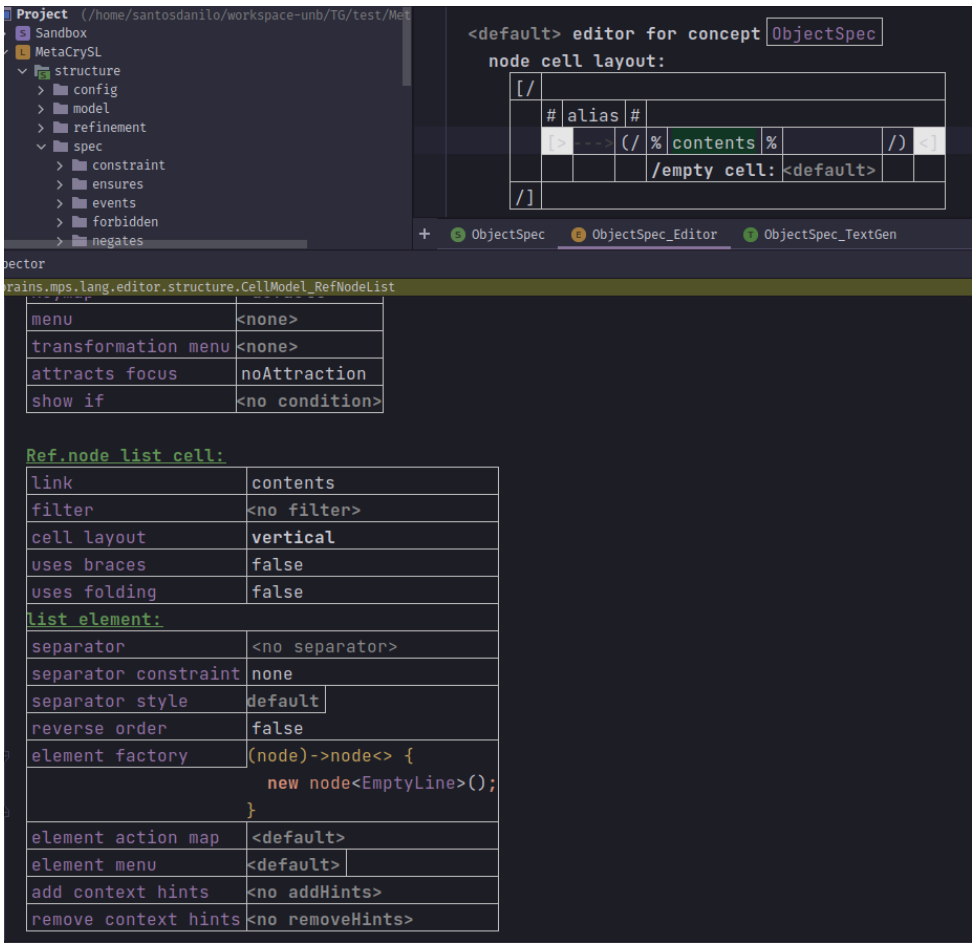

Figura 3.17: Janela do inspector.

nenhum campo, Figura 3.18, mas com um comportamento diferente; Pode-se adicionar qualquer texto dentro dele, o que permite digitar parcialmente alguma *keyword* ou sím-

bolo, que se aproxime de um alias, e substituir o esse nó, pelo do tipo selecionado no autocomplete. Esse comportamento foi definido no Inspector. Figura 3.19

Figura 3.18: ConceptEmptyLine.

Na seção acima, foi definida a implementação do comportanto de edição textual, que é comum há muitos *Concept*. Os trechos a seguir descrevem, partes mais especificas dos *Edtior* de cada *Concept*.

Em Spec, optou-se por deixar a primeira linha do arquivo MetaCrySL ser editável, onde seu elementos foram dispostos usando *collection cell horizontal*.

Dentro dessa collection cell foram incluidas outras células; Mais há esquerda, está a célula que refleta o campo abstract. Aqui utilizou-se a célula *flag* da bliblioteca *gram-marcells* para que ao invés inserir um valor booleano, pudessemos ao digitar a *keyword* ABSTRACT, e ocorre-se a substituição do texto pelo valor verdadeiro e esse valor fosse atribuido ao respectivo campo. Na célula central, fica o texto que evidência que é um arquivo SPEC, sendo usado o campo *alias* definido no *Concept*. Na célula mais a direita, é feita a exibição de uma outra célula, o *Editor* para o campo type, que lida com as particularidade de edição daquele nó. Figura 3.20

Para os *Concept* ObjectSpec, EventSpec, OrderSpec, ConstraintSpec, EnsureSpec, ForbiddenSpec, NegatesSpec e RequiresSpec, o *Editor* foi definido igualemente. Dentro de uma *collection cell vertical*, há a primeira célula, que exibe um valor constante, o campo *alias*, e outra célula para exibição dos nós filhos do campo contents usando uma *child node cell list(horizontal)* dentro de uma *cell collection horizontal*.

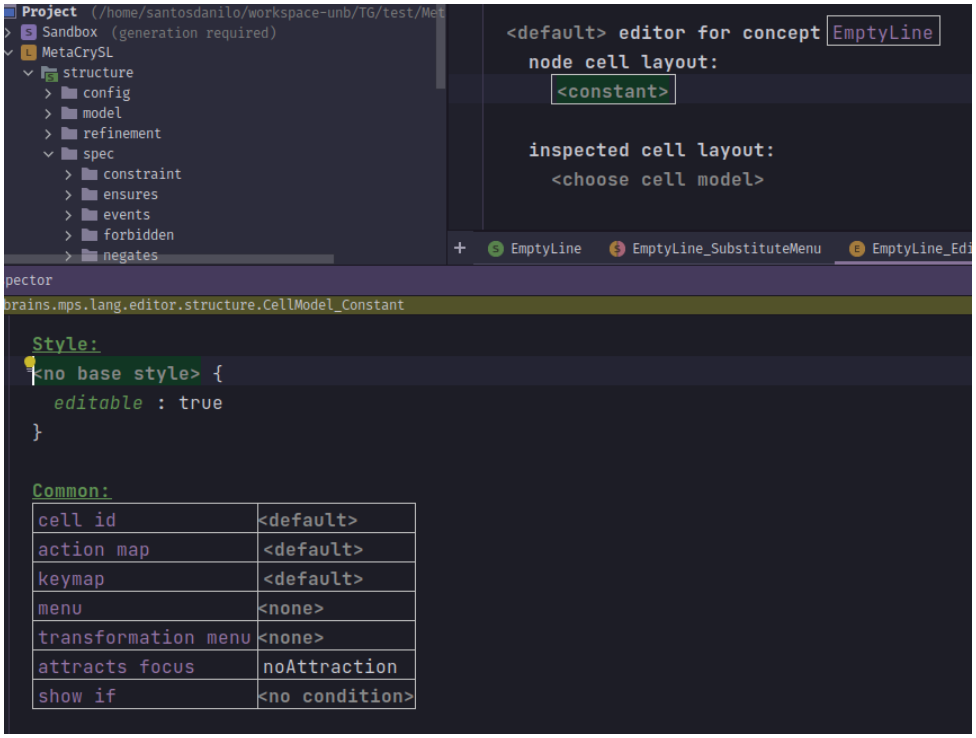

Figura 3.19: ConceptEmptyLine.

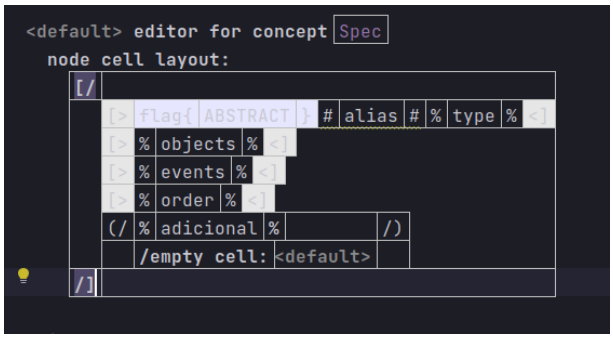

Figura 3.20: Editor de Spec.

3.2.7 Editor de Concept filho de ObjectSpec

Para o *Concept* Object, o *Editor* definido consiste de uma *cell horizontal collection*, constituida da primeira célula, que exibe o campo *type* - que possui um editor próprio -, uma célula constante que contém um espaço, uma célula constate que exibe o nome, e a celula constante que exibe o caracter ponto e vírgula, demarcando visualmente, o fim da declaração;

3.2.8 Editor de Concept filho de EventSpec

Para EventMethod, o *Editor* Figura 3.21 definido consiste de uma *cell horizontal collection*, com células filha, em que a primeira é uma célula *property declaration*, reponsável pela propriedade label, em seguida há uma *constant cell* que exibe o alias, e ao final há as células method que delega a manipulação para o *Editor* de MethodDef e uma *constant cell* que exibe o caracter ponto e virgula demarcando visualmente, o fim da declaração;

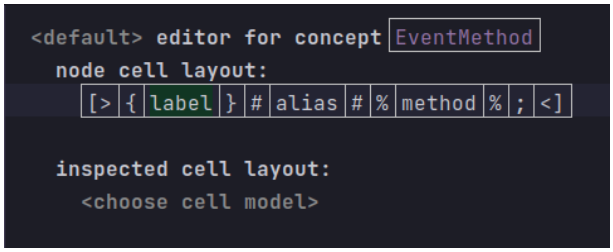

Figura 3.21: EventMethod Editor.

Para MethodDef, o *Editor* Figura 3.21 definido consiste de uma *cell horizontal collection*, onde a primeira célula, é um *wrap* da biblioteca *grammarcells*, que ao entrarmos com um texto, que bate com o padrão especificado, ele cria um nó do respectivo *Concept* e atribuiu o texto a propriedade do *Concept* criado. A célula seguinte é um *brackets* da biblioteca *grammarcells*, que além de adicionar simbolo especificado, ela instancia aumaticamente o *Concept* da célula da propriedade especificada, não sendo necessário o usuário manualmente criá-lo.

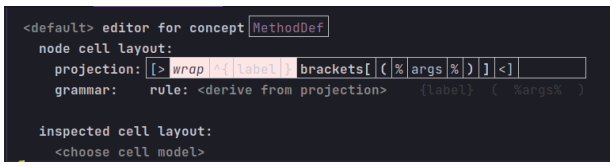

Figura 3.22: Editor para MethodDef.

Para FormalArgList, foi utlizado uma célula *ref node*, e como pode haver mais que um nó no campos args, foi definido um caracter separador e atribuida a regra *style Comma*, adquirindo a aparência de argumentos de uma função. O comportamento de criar um nó, ao digitar o caracter separador, é provido pelo próprio MPS, mas o comportamento de criar um novo argumento já do tipo correto, é obtido pela definição do *Editor* de QualifiedFormalArg Figura 3.21 e WildcardArgument onde ambos utilizam a célula *wrap*, que compara o texto com o padrão especificado, caso seja underscore é instanciado um WildcardArgument, senão é instanciado um QualifiedFormalArg.

3.2.9 Editor de Concept filho de OrderSpec

Para SequenceExp e ChoiceExp, o *Editor* foi definido da mesma maneira, em que é usada uma célula *collection (horizontal)*, com duas células *ref node* nas extremidas e uma célula *constant* da *grammarcells*, estruturando assim a expressão binária. A célula *constant* foi utilizada apenas com intuito de facilitar a refatoração, para uma generalização do *Editor* para expressões binário, pois essa célula permite a buscar o simbolo que representa a expressão, de maneira dinâmica.

Para outros *Concept* OneOrMore, Optional, ZeroOrMore, que derivam de BasicEventExp herdam o seu *Editor*, que consiste de célula *collection (horizontal)*, onde a célula a esquerda, um *node ref* é responsável pelo campo exp e a célula a direita, uma *grammar.constant* é responsável por exibir o símbolo definido nos *Concept* que herdam de BasicEventExp, no campo alias. Em PrimaryExp, há apenas uma célula *node ref* que delega a interação para o *Editor* de EventRef.

3.3 Linguagem REFINEMENT

3.3.1 Structure

```
concept Refinement extends    Model
                   implements INamedConcept

  instance can be root: true
  alias: REFINEMENT
  short description: <no short description>

  properties:
  << ... >>

  children:
  type    : Type[1]
  content : IRefinementContent[0..n]

  references:
  << ... >>
```

Figura 3.23: Concept Refinement.

A linguagem REFINEMENT,Figura 3.23, é estruturada a partir sob o *Concept* Refinement, com os campos name, herdado a partir de INameConcept, type e content. Assim como em SPEC, o campo type, referencia classes do Java, e no campo content são referenciados os nós filhos que implementem a interface IRefinementContent.

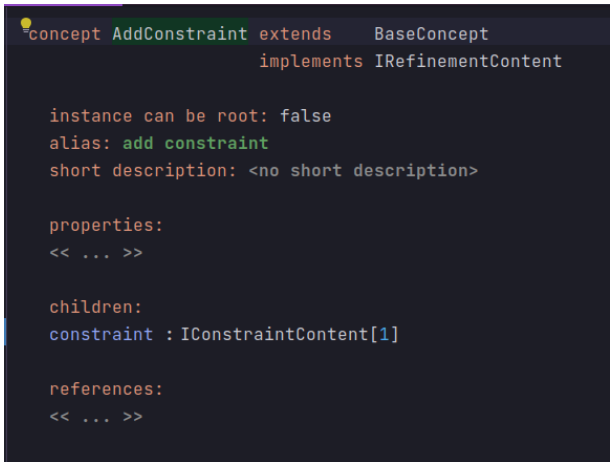

Figura 3.24: ConceptEmptyLine.

Figura 3.25: ConceptEmptyLine.

Os concepts que implementam essa interface são, AddContraint, AddEvent, AddRequire, AddEnsure, DefineLiteralSet e Rename.

O *Concept* AddContraint Figura 3.24, tem como único campo constraint, que pode ter como nó filho qualquer um dos *Concept* que implementem a interface IConstraintContent definidos para Spec.

O *Concept* AddEvent Figura 3.26, tem como único campo event, que pode ter como nó filho qualquer um dos *Concept* que implementem a interface IEventContent definidos para Spec.

O *Concept* AddRequire Figura 3.27, tem como único campo require, que pode ter como nó filho qualquer um dos *Concept* que implementem a interface IRequiresContent definidos para Spec.

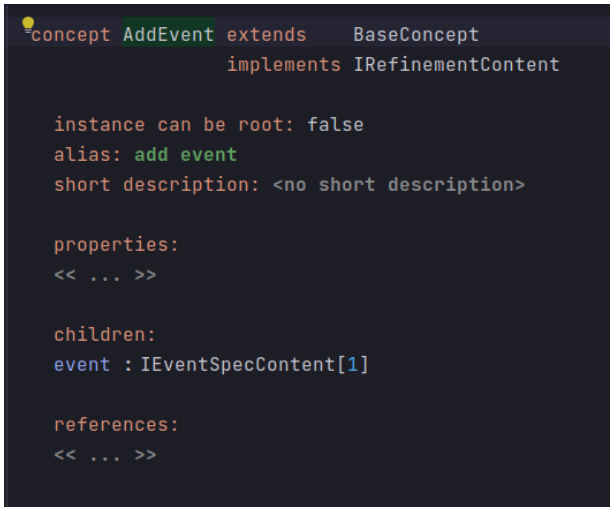

Figura 3.26: ConceptEmptyLine.

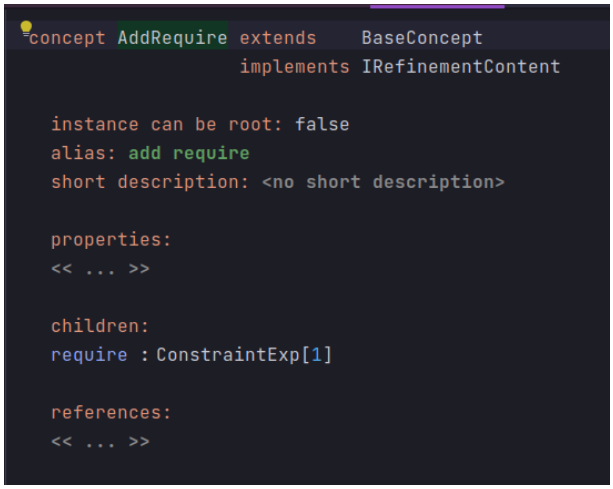

Figura 3.27: ConceptEmptyLine.

O *Concept* AddEnsure Figura 3.25 tem como único campo event, que pode ter como nó filho qualquer um dos *Concept* que implementem a interface IEnsureContent definidos para Spec.

O *Concept* Rename Figura **??** tem os campos type, do tipo ClassifierType - uma especialização de Type, o qual foi optado pois este referência apenas classes do Java, assim excluindo tipo primitivos, como possíveis tipo, já que uma SPEC é direcionado apena para classes.

Para o *Concept* DefineLiteralSet Figura 3.29, foram declarados os campos, label, do tipo ID e o campo set, do tipo ILiteralSet, podendo ser atribuidos os *Concept* de tipo literais - IntValue, StringValue.

Figura 3.28: Concept Rename.

Figura 3.29: Concept DefineLiteralSet.

3.3.2 Editor

Para os *Concept* AddContraint Figura 3.30, AddEvent Figura 3.32, AddRequire Figura 3.33, AddEnsure Figura 3.31 foi uma declaração simples, repetida em todos eles, pois todos consistem da exibição de um texto constante e uma expressão definda a partir de *Concept* de Spec. Foi externamente definida uma célula *collection (horizontal)*, e internamente, tendo como primeira célula o *alias*, que auxilia no *code completion*, a célula central que exibe o *Editor* definido para o *Concept* do respectivo campo e uma célula ao final, de texto fixo que ilustra um caractér delimitador.

Para o *Concept* Rename Figura 3.34 foram utilizada duas células *constant*, para exibir o alias, e o texto spec, e uma terceira célula foi usada para exibir o campo type, onde a exibição desse campo é delegada ao *Editor* definido para ClassifierType.

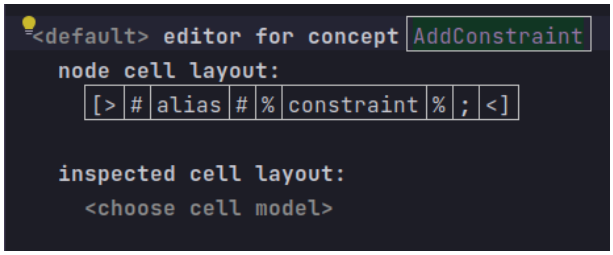

Figura 3.30: Editor para AddConstraint.

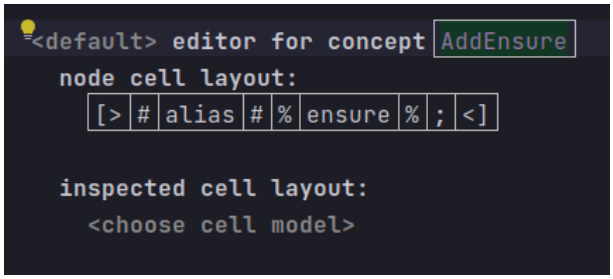

Figura 3.31: Editor para AddEnsure.

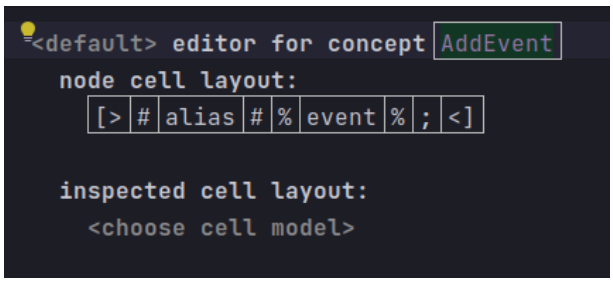

Figura 3.32: Editor para AddEvent.

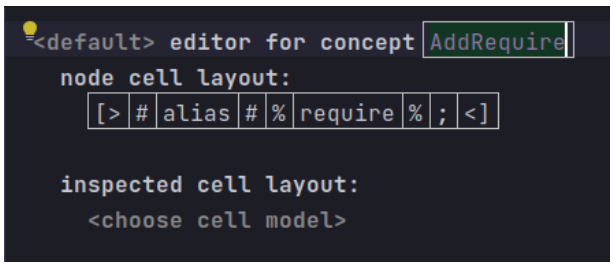

Figura 3.33: Editor para AddRequire.

Para o *Concept* DefineLiteralSet Figura 3.34 foi utilizada a célula *collection (horizontal)*, com quatro células internas, sendo a primera, uma célula constante, exibindo o alias, a segunda uma célula *node ref* que provê o acesso ao campo label, a terceira célula, uma

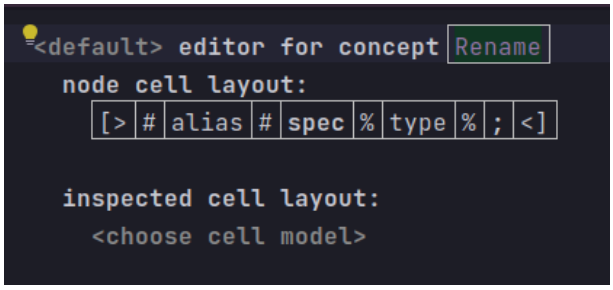

Figura 3.34: Editor para Rename.

constante para o símbolo de atribuição e quarta célula um *node ref* que delega o acesso ao *Editor* definido para o *Concept* que implemente ILiteralSet.

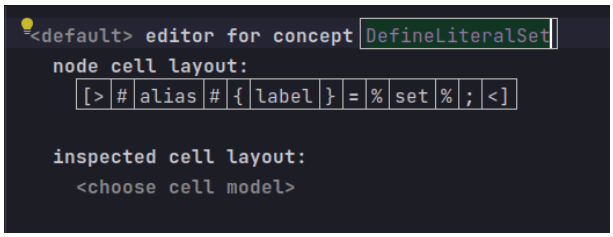

Figura 3.35: Editor para DefineLiteralSet.

3.4 Linguagem CONFIG

3.4.1 Structure

A linguagem CONFIG,Figura 3.36, é estruturada a partir sob o *Concept* Config, com os campos, alias, name, outputDir, para definir caminho de saida, inputSpec que permite atribuir multiplos arquivos SPEC, sendo requerido ao menos um arquivo, e inputRef que permite atribuir um ou nenhum arquivo REFINEMENT.

3.4.2 Editor

No *Editor* do *Concept* Config,Figura 3.37, foi utilizado externamento uma célula *collection (vertical)* para estrutura visualmente o arquivo de maneira vertical, onde a primeira linha consiste de uma célula constante que exibe o alias, uma célula *node ref* responsável pelo acesso e exibição do campo name e uma célula constante com o caracter chave delimitando visualmente o inicio do bloco. Na linha seguinte foi usando uma célula *indent* para indentar a célula seguinte que é um célula *collection (vertical)* para empilhar verticalmente as seções correspondentes ao arquivos SPEC e REFINEMENT. Para representar

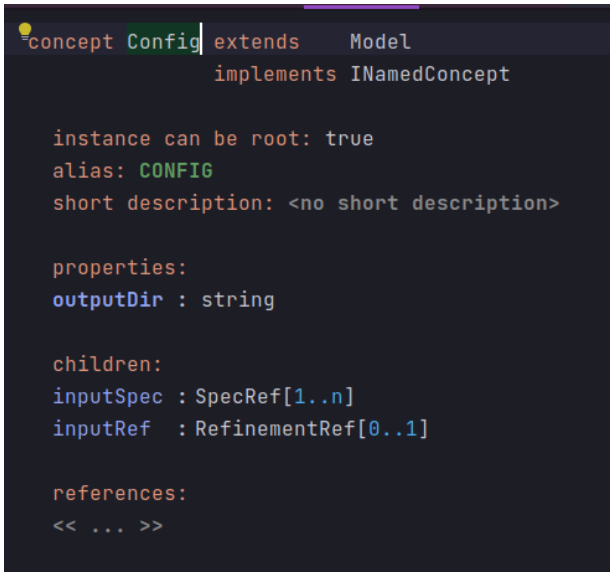

Figura 3.36: Concept Config.

a seção do arquivos SPEC, foi utilizado uma célula *constant* com o texto inputSpec, na primeira linha e na segunda linha foi usada uma célula para indentação e uma *collection (vertical)* para listar os nó que representam os arquivos SPEC. Para a seção dos arquivos REFINEMENT foi usado uma célula *constant* com texto inputRef, e na linha seguinte foi utilizada um célula para indentação e uma célula *node ref* responsável pelo campo inputRef, sendo essas células envolviadas por uma *collection (horizontal)*.

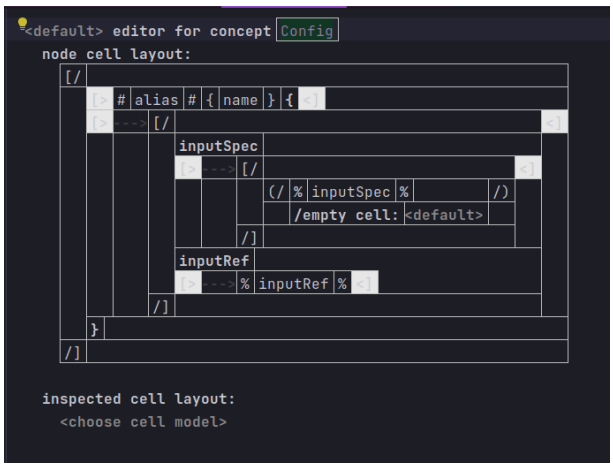

Figura 3.37: Editor Config.

3.5 Stylesheet

Ao definir as células é implementada como ocorre a inserção de dados e organização deles dentro do arquivo, e obtém-se a esperada interação semelhante a presente em um editor de texto. Mas há ainda um elementos visual, não implementando que ao pintar classes especificas de caracteres, ajuda na leitura e reconhecimento dos caracteres e sua função na gramática, recurso este que é o *text-highlight*. Como mostrado anteriormente o MPS permite a manipulação de alguma propriedades definidas para uma célula, geralmente sob a seção *Common*. Mas há outra seção, a *Style* que no permite utilizar estilos pré-definidos, e/ou definir localmente estilo para a célula.

Para aplicação sobre as celúlas foi contruída a *Stylesheet* MetaCryslHighlight, presente nas figuras Figura 3.38, Figura 3.39, Figura 3.40. Onde foram definidas valores para propriedade visuais, como cor, espaçamento, quebra de linha e associado a um identificador, que atua semelhante a uma class do *CSS* - Cascade Stylesheet.

São diversos as propriedades que podem ser modificadas, e multiplos *style* definidos, por isso nesta seção iremos apenas apresentar como um style é aplicado, e explicar o efeito que algumas propriedade causam visualmente sobre as células.

1. *apply* permite adicionar sob uma declaração *style* as propriedade presentes em outra declação *style*.
2. *punctuation-left* define se haverá um espaçamento a direto do símbolo
3. *punctuation-right* define se haverá um espaçamento a esquerda do símbolo
4. *text-foreground-color* define a cor do texto

Para aplicar o uma style sobre uma celúla, deve ser aberto o Inspector, onde a seção *Style* estará vazia como na Figura 3.41. Ao digitar o nome de uma declaração *style* no campo em que aparece o *placholder* no base style, a função *auto completion* é ativada, onde pode-se finalizar a seleção da declaração esperada, como em Figura 3.42.

3.6 Pipeline

Após ter definido as três linguagens necessária para o ambiente MetaCrySL, o próixmo passo consistiu de implementar o *pipeline* que utiliza o arquivo CONFIG, como entrada, juntamento com arquivos de REFINEMENT e SPEC, que são carregados a partir de um arquivo CONFIG.

A arquitetura de MetaCrySL para Raskal, Figura 3.43, propõe "um pipeline de múltiplos estágios para o processamento de linguagem, onde um módulo carregar o arquivo

```
stylesheet MetaCryslHighlight {
  style brackets {
    punctuation-left : true
    punctuation-right : true
  }

  style KeyWord {
    apply : KeyWord
  }

  style StringLiteral {
    apply : StringLiteral
  }

  style NumericLiteral {
    apply : NumericLiteral
  }

  style AnyBracket {
    apply : AnyBracket
  }

  style Parenthesis {
    apply : AnyBracket
    apply : PARENTH
    matching-label : parenthesis
  }

  style LeftParen {
    apply : Parenthesis
    punctuation-right : true
  }
```

Figura 3.38: Folha de Estilo MetaCryslHighlight.

CONFIG, que especifica um conjuneot de especificações CrySL extendida e um conjunto de REFINEMENTS que devem ser usadas durante o processamento das regras CrySL. Após isso o módulo Loader faz o parse desses conjuntos de regras CrySL e REFINEMENT, gerando uma representação abstrata. O modulo de pré-processamento manipula as instancias aplicando as operações de REFINEMENT e ao final gera um representação abstrata de regras CrySL. Ao final o módulo Pretty Printer gera como saída arquivos de especificação CrySL"[3]

Essa descrição em alto nível para Raskal é bem ilustrativa, e generaliza bem para

Figura 3.39: Folha de Estilo MetaCryslHighlight.

diferentes *language workbench*, entretanto em MPS, essa estruturas são um pouco mais integradas. Não há especificamente o módulo de loader, pois os arquivos envolvidos já estão no contexto de processamento, sendo na verdade carregados quando a IDE é iniciada. O pré-processador é equivalente ao Generator, que é um arquivo com um conjunto de regras de redução, as quais são aplicadas até que não seja possivel aplicar mais nenhuma, onde a cada aplicação são gerados modelos intermediários. Ao final o módulo Petty Printer é análogo ao TextGen, que tem arquivos texto como saída. Esse arquivos texto são os nosso artefatos úteis, arquivos de especificação CrySL.

```
style Semicolon {
  apply : Semicolon
}

style Operator {
  apply : Operator
}

style Separator {
  punctuation-left : false
  punctuation-right : false
}

style MetaVariableLeft {
  apply : Brace
  text-foreground-color : # b300b3
}

style MetaVariableRight {
  apply : Brace
  text-foreground-color : # b300b3
}

style Quotation {
  punctuation-left : true
}

}
```

Figura 3.40: Folha de Estilo MetaCryslHighlight.

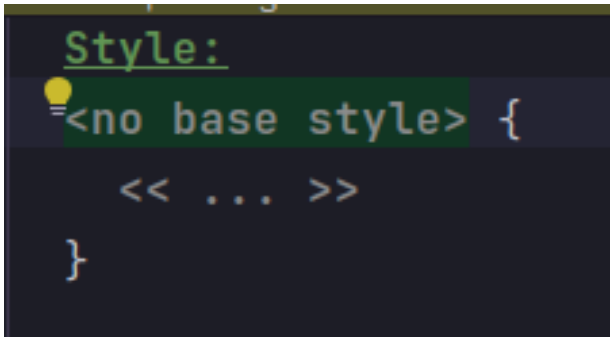

Figura 3.41: Seção Style vazia.

Para realizar a transformação do arquivos SPEC, o *Generator*, Figura 3.44 e Figura 3.45, tem implementadas duas regras de redução; A *pre-processing scripts* que realiza alguma tarefa antes de qualquer outra regra, e *reducion rules* que realiza a transformação de um *Concept* é algum outro resultado.

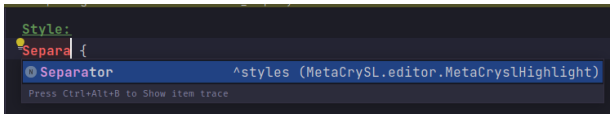

Figura 3.42: Seção Style ao selecionar ativar code completion.

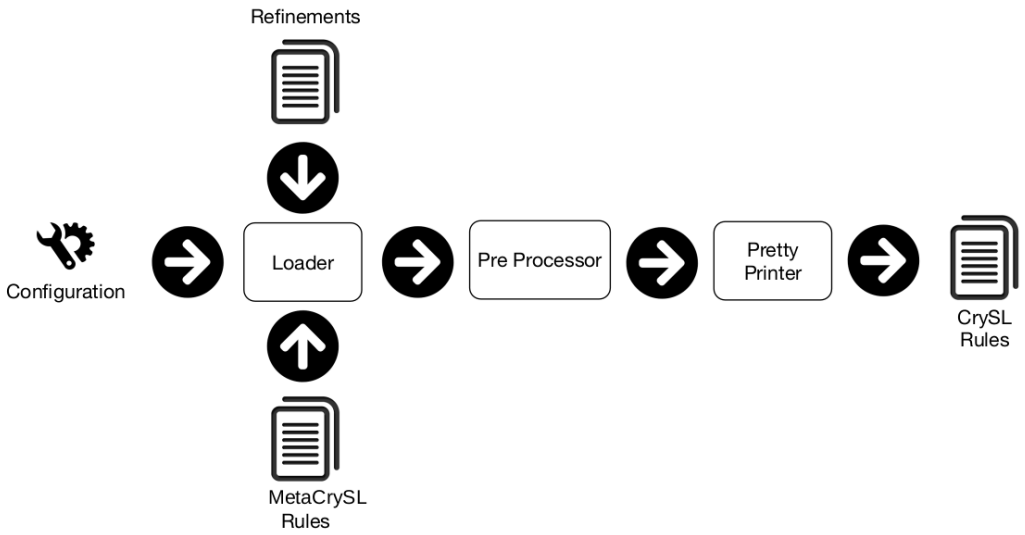

Figura 3.43: Arquitetura em alto nível de MetaCrySL para Raskal.

A *reducion rules* foi utilizado sobre o RefinementRef removendo-o do processo de transformação, usando *abandon input*.

A regra *pre-processing scripts* implementa as modificações especificadas no REFINEMENT, que serão aplicadas sobre os arquivos SPEC. O arquivo RefinementApply especifica como as transformações devem ser feitas.

Em RefinementApply, Figura 3.46, é executado um algoritmo simples, em que é obtido o nó referente ao REFINEMENT e uma lista de SPECs. Então é executado um primeiro nível de iteração sobres a lista de arquivos SPEC, onde são obtidos e em um segundo nivel, que itera sobre as declarações dentro de REFINEMENT, são aplicadas transformação sobre o arquivos SPEC com o uso de funções auxiliares especializas para cada declaração de REFINEMENT em específico. Essas funções estão agrupadas sob a class Util, como métodos estáticos, que recebem um SPEC e seu correspondente REFINEMENT.

As funções que realizam as transformações sobre arquivos SPEC são especificas de acordo o tipo de REFINEMENT. A função AddEvent, Figura 3.47 anexa o Event especificado a lista de nós IEventContent.

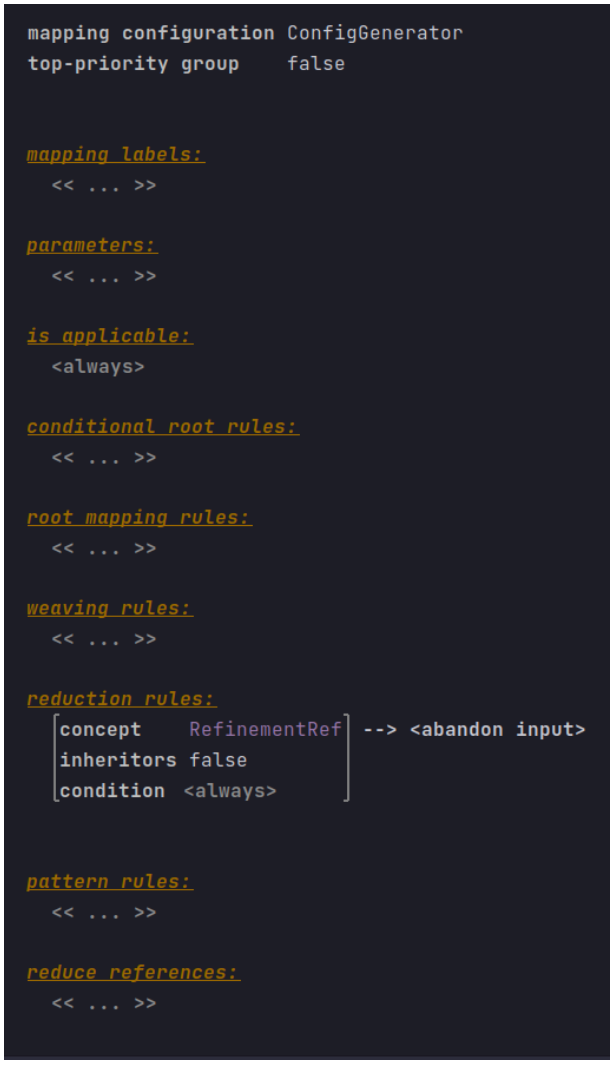

Figura 3.44: Config Generator.

As funções AddRequire, AddConstante, Figura 3.48, e AddEnsure, Figura 3.47, modificam o arquivo da seguinte maneira; Caso não existe nenhuma regra do respectivo bloco, ele é criado e é adicionado a regra à ele; Caso já existe é apenas anexada ao bloco existente.

A função Rename,Figura 3.50, realiza a substituição do nós type, pelo especificação em REFINEMENT.

A função DefineLiteralSet,Figura 3.50, procura os nós do tipo MetaVariable e realiza a substituição do nós pelo valor especificado em REFINEMENT.

Com essas operação, conseguimos implementar todas as transformação necessárias, e como etapa final, é executado o TextGen, o qual gera os arquivos textos.

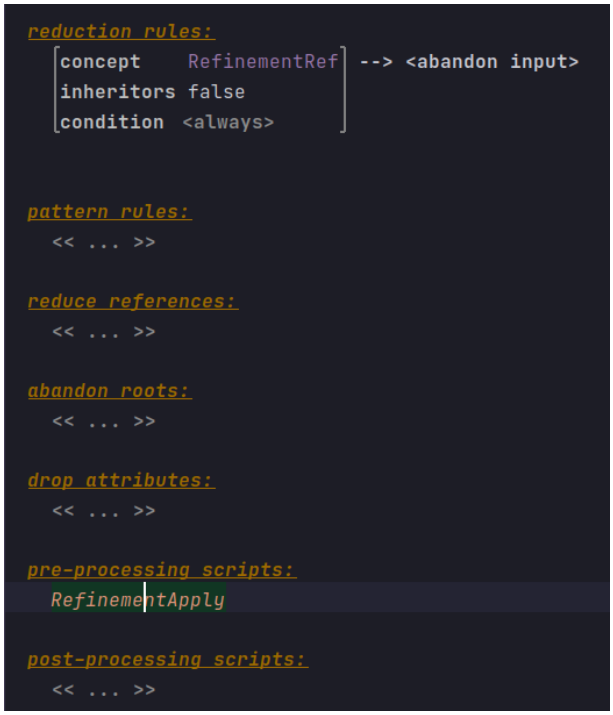

Figura 3.45: Config Generator.

3.7 Geração de Artefatos

Após terem sidos realizadas todas as transformações intermediárias do ConfigGenerator, há o último passo que é a conversão do modelo em arquivo texto CrySL. Para fazer a conversão de modelo em texto, o MPS tem o recurso chamado *TextGen*.

No *TextGen*, podemos definir, na seção de metadata, como obter o nome do arquivos, o caminho de saida, e a extensão do arquivo. Na seção de transformação, que pode-se localizada visualmente dentro da função, são definidas a operações. Podem ser utilizadas estruturas de controle típicas do java, e implementar qualquer lógica necessária, mas a operação que escreve no arquivo de saída é apenas a operação *append*. Com ela definimos o texto de saída, podendo concaternar texto constante, como valor de nó da árvore, delegar aquele trecho da saída ao *TextGen* definido para o *Concept* do nós e também usar o caractér quebra de linha.

O *TextGen* definido para Spec,Figura 3.50, possui todas essas operações e estruturas descritas. É verificado se o nó tiver o campo ABSTRACT como verdadeiro, ele tem como saida o texto que expressa isso. Na operação *append* seguinte, há gerado o texto que diz que aquele arquivo é um SPEC, que é concatenador a uma chamada de método defindo para o campo type, que obtem o nome da classe Java atribuida. Há ainda o bloco do forEach, que intera sobre o nós relacionados a ConstrainsSpec, EnsureSpec e outros outro

```
mapping script RefinementApply

script kind    : pre-process input model
modifies model : true

(genContext, model)->void {
  nlist<SpecRef> specificationInputs = genContext.inputModel.nodes(SpecRef);
  node<RefinementRef> refinementInput = genContext.inputModel.nodes(RefinementRef).first;

  for (node<SpecRef> specRef : specificationInputs) {
    node<Spec> specDef = specRef.ref;
    for (node<> rule : refinementInput.ref.content) {
      ifInstanceOf (rule is AddEvent addEvent) {
        Util.AddEvent(specDef, addEvent);
      }
      ifInstanceOf (rule is AddRequire addRequire) {
        Util.AddRequire(specDef, addRequire);
      }
      ifInstanceOf (rule is AddEnsure addEnsure) {
        Util.AddEnsure(specDef, addEnsure);
      }
      ifInstanceOf (rule is AddConstraint addConstraint) {
        Util.AddConstraint(specDef, addConstraint);
      }
      ifInstanceOf (rule is Rename rename) {
        Util.Rename(specDef, rename);
      }

    }
  }
}
```

Figura 3.46: Refinement apply.

que são nós no campo adicional, onde é verificado se aquele bloco possui alguma definição dentro dele, se houver a geração de texto é delegada ao *TextGen* do respectivo *Concept*.

```
public class Util {
  public static void AddEvent(node<Spec> spec, node<AddEvent> addEvent) {
    node<IEventSpecContent> copy = addEvent.event.copy;
    spec.events.content.addLast(copy);
  }

  public static void AddEnsure(node<Spec> spec, node<AddEnsure> addEnsure) {
    node<ConstraintExp> copy = addEnsure.ensure.copy;
    node<EnsuresSpec> list = (node<EnsuresSpec>) spec.adicional.findFirst({~it => it.isInstanceOf(EnsuresSpec); });
    if (list != null) {
      node<EnsuresSpec> field = list;
      field.content.addLast(copy);
    } else {
      node<EnsuresSpec> nodeToAdd = new node<EnsuresSpec>();
      nodeToAdd.content.add(copy);
      spec.adicional.addLast(nodeToAdd);
    }
  }
```

Figura 3.47: Refinement apply.

```
  public static void AddRequire(node<Spec> spec, node<AddRequire> addRequire) {
    node<ConstraintExp> copy = addRequire.require.copy;
    node<RequiresSpec> list = (node<RequiresSpec>) spec.adicional.findFirst({~it => it.isInstanceOf(RequiresSpec); });
    if (list != null) {
      node<RequiresSpec> field = list;
      field.content.addLast(copy);
    } else {
      node<RequiresSpec> nodeToAdd = new node<RequiresSpec>();
      nodeToAdd.content.add(copy);
      spec.adicional.addLast(nodeToAdd);
    }
  }

  public static void AddConstraint(node<Spec> spec, node<AddConstraint> addConstraint) {
    node<IConstraintContent> copy = addConstraint.constraint.copy;
    node<ConstraintSpec> list = (node<ConstraintSpec>) spec.adicional.
        findFirst({~it => it.isInstanceOf(ConstraintSpec); });
    if (list != null) {
      node<ConstraintSpec> field = list;
      field.content.addLast(copy);
    } else {
      node<ConstraintSpec> nodeToAdd = new node<ConstraintSpec>();
      nodeToAdd.content.add(copy);
      spec.adicional.addLast(nodeToAdd);
    }
  }
```

Figura 3.48: Refinement apply.

```
  public static void Rename(node<Spec> spec, node<Rename> rename) {
    node<ClassifierType> copy = rename.type.copy;
    spec.type = copy;
  }

  public static void DefineLiteralSet(node<Spec> spec, node<DefineLiteralSet> define) {
    node<DefineLiteralSet> copy = define.copy;
    node<ConstraintSpec> block = (node<ConstraintSpec>) spec.adicional.
        findFirst({~it => it.isInstanceOf(ConstraintSpec); });
    sequence<node<>> seq = block.content.where({~it => it.isInstanceOf(MetaVariable); });

  }

  public static void DefineQualifiedType(node<Spec> spec, node<DefineQualifiedType> define) {
    node<DefineQualifiedType> copy = define.copy;
    sequence<node<>> seq = spec.objects.contents.where({~it => it.isInstanceOf(Object); }).
        where({~it => ((node<Object>) it).type.isInstanceOf(WildCardType); });

  }

}
```

Figura 3.49: Refinement apply.

```
  public static void Rename(node<Spec> spec, node<Rename> rename) {
    node<ClassifierType> copy = rename.type.copy;
    spec.type = copy;
  }

  public static void DefineLiteralSet(node<Spec> spec, node<DefineLiteralSet> define) {
    node<DefineLiteralSet> copy = define.copy;
    node<ConstraintSpec> block = (node<ConstraintSpec>) spec.adicional.
        findFirst({~it => it.isInstanceOf(ConstraintSpec); });
    sequence<node<>> seq = block.content.where({~it => it.isInstanceOf(MetaVariable); });

  }

  public static void DefineQualifiedType(node<Spec> spec, node<DefineQualifiedType> define) {
    node<DefineQualifiedType> copy = define.copy;
    sequence<node<>> seq = spec.objects.contents.where({~it => it.isInstanceOf(Object); }).
        where({~it => ((node<Object>) it).type.isInstanceOf(WildCardType); });

  }

}
```

Figura 3.50: Refinement apply.

```
text gen component for concept Spec {
file name : <Node.name>
file path : <model/qualified/name>
extension : (node)->string {
  return "crysl";
}
encoding : utf-8
text layout : <no layout>
context objects : << ... >>

  (node)->void {
    if (node.ABSTRACT) {
      append {ABSTRACT };
    }
    append {SPEC } ${node.type.getDetailedPresentation()} \n;
    append \n;
    append ${node.objects} \n;
    append ${node.events} \n;
    append ${node.order} \n;
    foreach block in node.adicional {
      if (!(block.children.isEmpty())) {
        append ${block} \n;
      }
    }
  }
}
```

Figura 3.51: TextGen para Spec.

Capítulo 4

Avaliação de Resultados

Obtidas a implementação da linguagem, tendo definida sua estrutura, como será gerado um artefato útil a partir do que foi escrito com aquela linguagem, e definido como o usuário vai interagir com essa linguagem, conseguindo explicitar os conceitos desejados e tendo uma experiência próxima da habitual com editores de texto, cabe avaliar os resultados obtidos nesse processo. Essa avaliação ocorrerá a partir de pontos de vistas diferentes - do ponto de visto do especialista e de um construtor de DSL.

4.1 Como especificador de linguagem

A maneira à qual o MPS, é arquitetado sobre uma IDE rica em suporte ao processo de desenvolvimento torna o processo de especificação mais simples e familiar. O conceito presentes na linguagem base permitem um reaproveitamente de conhecimentos prévios de outro paradigmas. A especificação de um *Concept*, Figura 3.4, é similar a definir uma linguagem OO, onde se definie uma classe com campos, tipos para esses campos e a cardinalidade de elementos que esse campo suporta. O suporte a composição de linguagens, ocorre através dos campos, e a extensão de uma similar a herança, a implementação de um classe abstrata ou interface. [8]

Entretanto há dificuldades que podem dificuldar a adoção do MPS. A mais notável, é a dificulade ou mesmo impossibilidade de utilizar arquivos previamente definidos em alguma outra *workbench* com edição textual, já que a entrada de informações sobre a AST, é projecional. Ainda assim é seria possível, em alguns casos, superar essa limitação, primeiramente implementando os recurso de copy-paste presentes no MPS, ou escrevendo um programa que fizesse a transformação do arquivo-fonte para o mesmo formato de representação interna do MPS.

A existência e separação de elementos, que definem coisas especificas e particulares, possbilita ainda uma melhor manutenção, e encapsulamento dos diversos comportamentos

que o MPS possui, bem como a compreensão do mesmos, e a associação do que usar, para implementar um determinado comportamento.

4.2 Como usuário especialista

A primeira vista pode parecer semelhante a outros editores, mas o suporte a múltiplas notações - textual, tabular, gráfica e símbólica [9] - possibilita uma edição mais rica dos arquivos, permitindo expressar em outrao formatos conceitos que previamente poderiam apenas ser descritos textualmente, facilitando a compreensão, bem como a criação da especificação. E a integração com a IDE, que disponibiliza recursos como sugestões, ou alterações sobre uma porção do arquivo, valorizam ainda mais o ambiente de edição

Capítulo 5

Trabalhos Futuros

Apesar dos resultado obtidos, com a geração dos arquivos de especificação, dentro de um ambiente ricamente interativa, há pontos que não foram trabalhados por do tempo e da estratégia assumida. Dentre as adições que podem ser feitas a esse trabalho, estão a inclusão de uma notação visual, que expresse a relação entre os events dentro da seção order, e incrementar o sistema de tipos, que é uma funcionalidade presente no MPS, pois permitira verificar outras tipos de erro, como por exemplo a realização de operações entre tipo diferentes de operados.

Referências

[1] Nadi, Sarah, Stefan Krüger, Mira Mezini e Eric Bodden: *"jumping through hoops": Why do java developers struggle with cryptography apis?* Em *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, páginas 935–946, 2016. 1, 4

[2] Amann, Sven, Hoan Anh Nguyen, Sarah Nadi, Tien N. Nguyen e Mira Mezini: *A systematic evaluation of static api-misuse detectors.* IEEE Transactions on Software Engineering, 45(12):1170–1188, 2019. 1

[3] Bonifácio, Rodrigo, Stefan Krüger, Krishna Narasimhan, Eric Bodden e Mira Mezini: *Dealing with variability in API misuse specification.* CoRR, abs/2105.04950, 2021. `https://arxiv.org/abs/2105.04950`. 2, 5, 6, 29

[4] Fowler, Martin: *Language workbenches: The killer-app for domain specific languages?* junho 2005. 4

[5] Krüger, Stefan, Johannes Späth, Karim Ali, Eric Bodden e Mira Mezini: *Crysl: An extensible approach to validating the correct usage of cryptographic apis.* IEEE Transactions on Software Engineering, páginas 1–1, 2019. 4

[6] Silva, Vinicius Costa e: *Implementação da linguagem de metaprogramação meta-crysl utilizando o framework xtext.* 2020. 5

[7] Jetbrains: *Editor*, 2021. `https://www.jetbrains.com/help/mps/editor.html`, acesso em 2021-11-01. 15

[8] Völter, Markus e Konstantin Solomatov: *Language modularization and composition with projectional language workbenches illustrated with mps.* janeiro 2010. 39

[9] Erdweg, Sebastian, Tijs van der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth e Jimi van der Woning: *Evaluating and comparing language workbenches: Existing results and benchmarks for the future.* Computer Languages, Systems Structures, 44:24–47, 2015, ISSN 1477-8424. `https://www.sciencedirect.com/science/article/pii/S1477842415000573`, Special issue on the 6th and 7th International Conference on Software Language Engineering (SLE 2013 and SLE 2014). 40