Universidade Federal de Pernambuco
Centro de Informátiva
Graduação em Ciência da Computação

Síntese de programas utilizando a linguagem Alloy

João Pedro Marcolino Cordeiro

Trabalho de Graduação

Recife, Julho de 2017

Universidade Federal de Pernambuco
Centro de Informática
Graduação em Ciência da Computação

Síntese de programas utilizando a linguagem Alloy

Trabalho apresentado ao Programa de Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Aluno: João Pedro Marcolino Cordeiro
(jpmc@cin.ufpe.br)

Orientador: Alexandre Cabral Mota
(acm@cin.ufpe.br)

Recife, Julho de 2017

Agradecimentos

Agradeço aos meus pais, José Cordeiro da Silva e Zildete Marcolino Silva, por todo o amor, compreensão, estímulo e apoio em todos os momentos da minha vida. Agradeço a minha irmã, Joseanne Cordeiro, pelo amor, apoio e companheirismo inestimáveis. Agradeço aos amigos que estiveram do meu lado durante todos esses anos estudando, aprendendo, ensinando, desenvolvendo e apoiando uns aos outros. Ao meu orientador, Alexandre Mota, pela disponibilidade e ajuda no desenvolvimento desse trabalho. Por último, agradeço ao Centro de Informática e a todos os seus alunos, professores, pesquisadores e funcionários por todo o aprendizado, oportunidades, infraestrutura e excelência acadêmica em todos os anos da minha graduação.

Resumo

Desenvolvimento de sistemas complexos de software ou hardware é um processo caro, que requer conhecimento específico não só de programação como de especialistas do domínio. Além disso, o processo de planejamento, design e codificação frequentemente introduz erros ao sistema que não são percebidos imediatamente pelos desenvolvedores. Nos últimos anos, métodos baseados em formalismos matemáticos, conhecidos como métodos formais, para especificação, desenvolvimento e verificação de software vêm sendo estudados e desenvolvidos; buscando assim mudar a maneira tradicional de desenvolver sistemas computacionais.

Uma dessas abordagens é a síntese de programas, que tem por objetivo automatizar a codificação de programas executáveis a partir de especificações de comportamento. Várias ferramentas desenvolvidas recentemente como SKETCH (Solar-Lezama A. , 2003), Rosette (Torlak & Bodik, 2013) e Microsoft PROSE (Polozov & Gulwani, 2015) possuem funcionalidades de síntese, mas apresentam limitações que não permitem a generalização e facilidade de desenvolvimento.

Este trabalho apresenta uma generalização do processo de síntese de programas utilizando a linguagem Alloy. Também é apresentado um modelo de um sintetizador capaz de gerar programas na linguagem IMP e exemplos de programas sintetizados por ele.

Sumário

1 – Introdução 1

2 – Fundamentos 2

2.1 *Boolean satisfiability problem* 2

2.2 Métodos formais 2

2.3 Verificação de programas 3

2.3.1 Prova de teoremas 3

2.3.2 Checagem e busca de modelos 3

2.4 Síntese de programas 3

2.4.1 Especificações de propriedades 3

2.4.2 Classificação de abordagens de síntese 5

3 – Revisão Bibliográfica 6

3.1 SKETCH 6

3.2 Rosette 7

3.3 VS3 —Verification and Synthesis using SMT Solvers 8

3.4 Microsoft PROSE 9

3.5 Evolução gramatical 9

3.6 Regressão simbólica 9

3.7 Comparação entre as ferramentas 10

4 – Alloy 11

4.1 Sintaxe 11

4.1.1 Assinaturas e relações 11

4.1.2 Funções, predicados, fatos e asserções 12

4.1.3 Comandos e escopo 13

4.1.4 Modularização 14

4.2 Semântica e operadores 14

4.2.1 Conjuntos, átomos e relações 14

4.2.2 Constantes 15

4.2.3 Operadores sobre conjuntos 15

4.2.4 Operadores sobre relações 16

4.3 O analisador Alloy 16

4.4 Alloy* 17

5 – IMP 18
5.1 Sintaxe 18
5.2 Semântica 18
5.2.1 Estados de programa 19
5.2.2 Avaliações de expressões aritméticas 19
5.2.3 Avaliações de expressões booleanas 19
5.2.4 Execuções de comandos 20

6 – Síntese de programas em Alloy 21
6.1 Módulo de síntese 21
6.1.1 Estruturas genéricas para definição de linguagens 22
6.1.2 Predicados de síntese 23
6.2 Módulo de sintaxe 23
6.2.1 Registros 24
6.2.2 Expressões aritméticas 24
6.2.3 Expressões booleanas 25
6.2.4 Comandos 26
6.3 Módulo de semântica 27
6.3.1 Semântica sobre expressões aritméticas 28
6.3.2 Semântica sobre expressões booleanas 29
6.3.3 Semântica sobre execução de comandos 29

7 – Exemplos de síntese 31
7.1 Sintetizando constantes 31
7.2 Sintetizando expressões 33
7.3 Sintetizando programas 35

8 – Conclusão 38
8.1 Trabalhos futuros 38

Referências 40

1 – Introdução

Desenvolvimento de sistemas complexos de software ou hardware é um processo caro, que requer conhecimento específico não só de programação como de especialistas do domínio. Além disso, o processo de planejamento, design e codificação frequentemente introduz erros ao sistema que não são percebidos imediatamente pelos desenvolvedores. Nos últimos anos, métodos baseados em formalismos matemáticos, conhecidos como métodos formais, para especificação, desenvolvimento e verificação de software vêm sendo estudados e desenvolvidos; buscando assim mudar a maneira tradicional de desenvolver sistemas computacionais.

Uma dessas abordagens é a síntese de programas, que tem por objetivo automatizar a codificação de programas executáveis a partir de especificações de comportamento. Várias ferramentas desenvolvidas recentemente como SKETCH (Solar-Lezama A. , 2003), Rosette (Torlak & Bodik, 2013) e Microsoft PROSE (Polozov & Gulwani, 2015) possuem funcionalidades de síntese, mas apresentam limitações que não permitem a generalização e facilidade de desenvolvimento.

O objetivo desse trabalho é atacar essas limitações e apresentar uma generalização do processo de síntese de programas utilizando a linguagem Alloy. Além disso, também tem o objetivo de desenvolver um sintetizador capaz de gerar programas na linguagem IMP e apresentar exemplos de programas sintetizados por ele.

Este documento é divido em capítulos, estruturados da seguinte forma: O Capítulo 2 aborta conceitos fundamentais para compreensão do trabalho e dos capítulos posteriores como resolvedores SAT, métodos formais, verificação de programas através de checagem de modelos e prova de teoremas e conceitos síntese de programas. No Capítulo 3 é apresentada uma revisão bibliográfica de ferramentas e trabalhos pertencentes ao estado-da-arte em síntese de programas nos últimos anos. No Capítulo 4 a sintaxe e a semântica da linguagem Alloy, utilizada para desenvolver o sintetizador produzido por este trabalho, são especificadas. O Capítulo 5 define a sintaxe e semântica de uma linguagem imperativa simples conhecida como IMP, que é a linguagem dos programas gerados pelo sintetizador produzido. O Capítulo 6 detalha o modelo Alloy do sintetizador e introduz o processo para síntese de programas utilizando Alloy. O Capítulo 7 apresenta exemplos de programas IMP gerados utilizando o sintetizador e serve como um tutorial do processo de síntese usando o mesmo. Por fim, o Capítulo 8 apresenta as principais contribuições do trabalho e possíveis trabalhos futuros.

2 – Fundamentos

Neste capítulo serão apresentados conceitos fundamentais para a compreensão do trabalho. Serão abordados métodos formais, resolvedores SAT, verificação e síntese de programas. Na seção sobre síntese de programas, serão descritas características comuns em abordagens de síntese e também maneiras de classifica-las.

2.1 *Boolean satisfiability problem*

O problema da satisfatibilidade booleana, mais conhecido pela sua abreviação SAT, é o problema de determinar se uma dada formula lógica é satisfatível, ou seja, pode ser avaliada como $TRUE$, para alguma valoração de suas variáveis. Caso tal valoração exista, essa fórmula é dada como satisfatível; caso contrário, ela é considerada insatisfatível. A fórmula $a \wedge \neg b$, por exemplo, é satisfatível para os valores $a = TRUE$ e $b = FALSE$. Já a fórmula $a \wedge \neg a$ não é satisfatível pois não existe uma valoração que a torne verdadeira.

Resolvedores SAT (do inglês SAT *solvers*) são programas e algoritmos desenvolvidos para resolver problemas SAT. Alguns exemplos são o MiniSat[1], Sat4j[2], e zChaff[3]. Muitos desses resolvedores são considerados aplicações *off-the-shelf*, ou seja, aplicações que estão disponíveis imediatamente para desenvolvedores ou usuários que queiram utilizá-las em suas tarefas ou como parte de outra aplicação.

2.2 Métodos formais

Métodos formais são técnicas e ferramentas de Engenharia de Software estudadas desde a década de 60 (McCarthy, 1963) para a especificação, design e verificação de sistemas de Software ou Hardware. Baseados em formalismos matemáticos, métodos formais comumente são usados para garantir certas propriedades sobre um sistema, como conformidade de um produto final com sua especificação inicial e a corretude dessa especificação. Métodos formais são aplicados frequentemente para verificação de programas e síntese de programas.

Existem diversas linguagens que são usadas para especificação e análise em métodos formais, como Z[4], Alloy[5], CSP[6], PRISM[7], entre outras. A linguagem Z é usada para descrever e modelar sistemas, especificando e refinando componentes desses sistemas e expressando restrições entre esses componentes (ISO/IEC 13568, 2002). A linguagem Alloy, baseada na linguagem Z, também é utilizada para descrever e modelar sistemas, mas provê uma ferramenta, o Alloy Analyzer, não disponível em Z, que permite checagem de modelos (Jackson, Software Abstractions: Logic, Language, and Analysis, 2006). Já CSP, ou *Communicating Sequential Processes*, é uma linguagem para descrever padrões de interação (Hoare, 1985). Ela é frequentemente utilizada para verificação e

[1] Disponível em: http://minisat.se/
[2] Disponível em: http://www.sat4j.org/
[3] Disponível em: http://www.princeton.edu/~chaff/zchaff.html
[4] Disponível em: https://www.iso.org/standard/21573.html
[5] Disponível em: http://alloy.mit.edu/
[6] Disponível em: http://www.usingcsp.com/
[7] Disponível em: http://www.prismmodelchecker.org/

especificação de concorrência em sistemas computacionais. Como um último exemplo, PRISM é uma ferramenta e linguagem usada para analisar e especificar sistemas computacionais através de checagem de modelos probabilísticos (Kwiatkowska, Norman, & Parker, 2011).

2.3 Verificação de programas

Verificação de programas é um processo usado para provar, através de métodos formais, que um algoritmo ou programa atende uma determinada especificação ou propriedade. Dentre as abordagens usadas para verificação, as mais comuns são checagem de modelos e prova de teoremas.

2.3.1 Prova de teoremas

A abordagem de prova de teoremas consiste em derivar, a partir do programa e da especificação da propriedade sendo verificada, um conjunto de teoremas matemáticos. A prova desse conjunto implica que o programa atende a propriedade. Um provador de teoremas interativo, como Isabelle[8] ou HOL[9], e até resolvedores SAT podem ser usados para provar esses teoremas (Dennis, Yessenov, & Jackson, 2008).

2.3.2 Checagem e busca de modelos

Outra abordagem para verificação é a checagem e busca de modelos. Dado um programa e a especificação da propriedade a ser verificada, o universo de possíveis estados do modelo que corresponde a sequência de execução do programa é explorado com o objetivo de identificar uma sequência que não respeite a especificação (Visser, Havelund, Brat, Park, & Lerda, 2003), (Deng, Lee, & Robby, 2006). A linguagem e a ferramenta Alloy, como mencionado anteriormente, podem ser usadas para especificar, checar e analisar esses modelos.

2.4 Síntese de programas

Ferramentas e métodos de síntese de programas têm como objetivo automatizar parte do (ou todo) processo de programação de software. Em termos mais claros: dado uma especificação sobre o comportamento de um programa, ferramentas de síntese procuram gerar um programa executável que atende a essa especificação (Basin, Deville, Flener, Hamfelt, & Nilsson, 2004). A maneira que essas especificações são dadas e a capacidade de síntese varia de acordo com a abordagem utilizada.

2.4.1 Especificações de propriedades

Cada técnica de síntese de programa define uma maneira de especificar o comportamento esperado para os programas sintetizados. As três mais comuns são através de asserções, contratos ou exemplos de entrada e saída. Tomemos como exemplo o programa “max2” da Figura 1, que calcula o máximo entre dois números através de uma função *max*. Esse programa recebe como entrada as variáveis *x* e *y* e armazena o valor da resposta desejada na variável *ans*.

[8] Disponível em: https://hol-theorem-prover.org/
[9] Disponível em: https://hol-theorem-prover.org/

```
int x;
int y;
int ans = 0;
int max(x,y) {
     if (x >= y) return x;
     else return y;
}
ans = max(x,y);
```

Figura 1: Pseudocódigo do programa max2

2.4.1.1 Asserções

A maneira mais comum de especificar um comportamento desejado para um programa é através de asserções inseridas diretamente no código fonte. A Figura 2 mostra um exemplo de uma asserção aplicável para o programa max2.

```
assert ans = (x if x >= y else y)
```

Figura 2: Especificação de programa através de asserção

2.4.1.2 Contratos

Outra opção é especificar claramente através de contratos qual é comportamento esperado para o programa. Contratos são predicados da lógica de primeira ordem que especificam propriedades sobre os estados iniciais e finais do programa. A Figura 3 mostra um exemplo de pré e pós condições sobre o programa max2.

```
pred pre_condition {
     int ans = 0
}
pred post_condition {
     x >= y → ans = x
     y > x → ans = y
}
```

Figura 3: Especificação de programa através de contrato

2.4.1.3 Exemplos de entrada e saída

Para usuários leigos, no entanto, escrever asserções ou propriedades lógicas sobre o comportamento de um programa pode ser mais complexo do que eles estão preparados. Algumas abordagens de síntese são capazes de gerar programas com base em um conjunto de pares de entrada e saídas desejadas que exemplifiquem o comportamento esperado do programa, que pode ser facilmente fornecido por um usuário. Na Figura 4, um possível conjunto de exemplos para o programa max2 é apresentado.

```
example_1 {
     initial_state {x = 1; y = 2; ans = 0}
     final_state {x = 1; y = 2; ans = 2}
}
example_2 {
     initial_state {x = 3; y = 1; ans = 0}
     final_state {x = 3; y = 1; ans = 3}
}
```

Figura 4: Especificação de programa através de exemplos

2.4.2 Classificação de abordagens de síntese

Existem diferentes maneiras para avaliar e classificar abordagens de síntese de programas. As mais recentes na literatura possuem características que podem ser usadas para diferenciá-las e compará-las. Essas características variam quanto a técnica utilizada, quanto ao tipo de especificação, quanto ao paradigma de linguagem de programação sintetizada e quando ao poder de síntese.

2.4.2.1 Quanto a técnica utilizada

Abordagens podem ser classificadas de acordo com a técnica utilizada em: síntese construtiva/dedutiva, síntese guiada por *schemas* e síntese indutiva (Basin, Deville, Flener, Hamfelt, & Nilsson, 2004).

2.4.2.2 Quanto ao tipo de especificação do programa a ser sintetizado

(Basin, Deville, Flener, Hamfelt, & Nilsson, 2004) também apresentam uma comparação entre as abordagens analisadas e as especificações dos programas que são gerados por elas. Especificações podem ser dadas através de contratos especificando as pré-condições do estado de um programa e as pós-condições do estado resultante da execução do programa; através de exemplos de entrada e saída desejadas para o programa (Polozov & Gulwani, 2015); ou simplesmente através de uma asserção em um algum ponto do programa (Solar-Lezama A. , 2003), (Torlak & Bodik, 2013).

2.4.2.3 Quanto ao paradigma de linguagem de programação sintetizada

Cada ferramenta de síntese pode ser diferenciada de acordo com o paradigma utilizado para representação dos programas sintetizados. Por exemplo, uma ferramenta de síntese pode ser especializada em sintetizar programas imperativos (Solar-Lezama A. , 2003), enquanto outras podem ser mais genéricas e possuir a capacidade de produzir programas pertencentes a múltiplos paradigmas (Torlak & Bodik, 2013).

2.4.2.4 Quanto a capacidade de síntese

Técnicas de síntese também variam de acordo com a capacidade e o escopo em que elas atuam. Existem ferramentas específicas para sintetizar constantes em um programa parcialmente completo (Solar-Lezama A. , 2003), outras são capazes de gerar expressões que podem ser reusadas em outros programas (Torlak & Bodik, 2013) e, por último, outras são capazes de gerar programas completos (Polozov & Gulwani, 2015).

3 – Revisão Bibliográfica

Neste capítulo serão apresentados trabalhos que constituem o estado da arte em síntese de programas. Os trabalhos analisados foram a ferramenta SKETCH, a linguagem Rosette, VS3, Microsoft PROSE, evolução gramatical e regressão simbólica. Por último, será discutida uma comparação esquemática dessas técnicas.

3.1 SKETCH

SKETCH (Solar-Lezama A. , 2003) é uma ferramenta de síntese de programas baseada em procura combinatória que toma como entrada uma implementação parcial (chamada de *sketch*) e uma especificação do comportamento desejado. O sintetizador completa a implementação parcial para se comportar de acordo com a especificação e o programa gerado tem a garantia de ser correto em relação a especificação.

Os *sketches* e as especificações são escritos em uma linguagem de programação criada especialmente para esse propósito. A linguagem SKETCH é uma linguagem imperativa relativamente simples e sua sintaxe se assemelha a da linguagem C. Um compilador e sintetizador para essa linguagem encontra-se disponível de maneira *open-source* na internet[10].

A Figura 5 apresenta um exemplo dessa linguagem. O símbolo ?? (chamado de *hole*, ou buraco) é utilizado para representar uma constante de valor inteiro desconhecida. O sintetizador substitui esses buracos por valores apropriados que satisfaçam a especificação do programa. Neste exemplo, a especificação é dada na forma de uma asserção. A palavra-chave *harness* é usada para indicar que o sintetizador deve achar um valor para o buraco que satisfaça a asserção para todos os possíveis valores de entrada x.

```
harness void doubleSketch(int x){
  int t = x * ??;
  assert t == x + x;
}
```

Figura 5 – Um exemplo de sketch na ferramenta SKETCH

O processo de síntese em SKETCH ocorre em três etapas. Primeiramente, o *sketch* sofre um pré-processamento para eliminar estruturas de alto nível, fazer simplificações e outras otimizações. Na segunda fase, valores aleatórios são atribuídos para os buracos e o *sketch* entra no loop de síntese e verificação. Nesse loop, durante a síntese, o *sketch* é convertido para uma fórmula booleana que é resolvida por um resolvedor SAT. Com isso, valores para os buracos são sintetizados para uma possível entrada do programa. Já durante a verificação, os valores atribuídos pela síntese são checados para todas as possíveis entradas. Caso esses valores não satisfaçam a verificação para algum valor de entrada, um contraexemplo é encontrado e usado automaticamente na próxima iteração de síntese. Quando um conjunto de valores satisfaz todos os inputs, o loop é terminado. Na terceira e última fase, o *sketch* original sofre um pós processamento e é retornado para o usuário com os buracos preenchidos.

As principais limitações dessa ferramenta são: SKETCH possui capacidade de sintetizar apenas constantes; além da especificação do comportamento, o desenvolvedor

[10] https://bitbucket.org/gatoatigrado/sketch-frontend/wiki/Home

também precisa fornecer uma implementação parcial para o programa que explicita as constantes desconhecidas que precisam ser encontradas; por último, a ferramenta só sintetiza programas imperativos pertencentes a uma linguagem específica, SKETCH.

3.2 Rosette

Rosette (Torlak & Bodik, 2013) é uma linguagem de programação que estende a linguagem Racket[11]. Racket é uma linguagem multi-paradigma pertencente à família de linguagens LISP com o objetivo de servir como uma plataforma para design, criação e implementação de outras linguagens de programação.

Rosette adiciona estruturas e funcionalidades à Racket, dentre elas síntese e verificação de programas. Especificamente para síntese, ela provê estruturas para definir "buracos" sintáticos. Um programa com buracos é chamado de *sketch*.

Valores em Rosette podem ser de dois tipos: concretos ou simbólicos. Um programa implementado apenas com valores concretos se comporta exatamente como um programa Racket. Já programas que contém valores simbólicos podem utilizar as funcionalidades de Rosette para verificação e síntese.

Nas Figuras 6 e 7 a seguir são mostrados exemplos de um *sketch* e de um predicado de síntese.

```
(require rosette/lib/synthax)

(define (poly x)
 (+ (* x x x x) (* 6 x x x) (* 11 x x) (* 6 x)))

(define (factored x)
 (* (+ x (??)) (+ x 1) (+ x (??)) (+ x (??))))

(define (same p f x)
 (assert (= (p x) (f x))))
```

Figura 6 - Um sketch em Rosette

```
> (define-symbolic i integer?)
> (define binding
    (synthesize #:forall (list i)
                #:guarantee (same poly factored i)))
> (print-forms binding)
'(define (factored x) (* (+ x 0) (+ x 1) (+ x 2) (+ x 3)))
```

Figura 7 - Um predicado de síntese em Rosette

O objetivo desse exemplo é sintetizar constantes para a função *factored* para que ela tenha o mesmo comportamento da função *poly* para todos os possíveis inputs.

[11] http://racket-lang.org/

Na Figura 6 é apresentado o *sketch* escrito na linguagem Rosette. O símbolo ?? é utilizado para definir buracos sintáticos que serão preenchidos pelas constantes numéricas durante a síntese do programa. *Asserts* são usados para definir especificações sobre o programa.

A Figura 7 mostra o predicado de síntese usado sobre o *sketch* definido na figura anterior. A palavra-chave *define-symbolic* é utilizada para definir tipos de valores simbólicos, que atuam como um representante do conjunto de valores concretos do mesmo tipo. Já "*synthesize #:forall <input> #:guarantee <expression>*" é a sentença de síntese. Ela especifica que uma *<expression>* (expressão) que contém um *assert* deve ter seus buracos sintáticos preenchidos de maneira que a especificação seja atendida para todos os possíveis valores do *input* (entrada). O resultado, se existir, é uma associação de valores concretos aos valores simbólicos restantes na expressão, criados a partir da avaliação dos buracos sintáticos. No caso do exemplo, o *input "list i"* é uma lista de valores simbólicos do tipo inteiro que representa as entradas para o programa que serão automaticamente usadas durante o processo de síntese. Já a expressão que contém o *assert* é *"same poly factored i"*, que realiza a comparação entre as funções *poly* e *factored* nas entradas *i*. O resultado de síntese é apresentado na última linha da Figura 7, com os buracos sintáticos substituídos pelos valores zero, dois e três.

Além do símbolo ??, Rosette também provê estruturas para especificar buracos sintáticos mais complexos. É possível especificar, por exemplo, um buraco sintático que deve ser substituído por uma expressão gerada a partir de uma gramática definida pelo desenvolvedor.

As principais vantagens que Rosette possui são: os buracos de programas podem ser substituídos por expressões retiradas de uma gramática definida pelo desenvolvedor. Além disso, também é possível gerar programas que pertencem a qualquer linguagem, desde que ela seja definida utilizando as funcionalidades de criação de Racket, e gerar expressões dessa linguagem utilizando Rosette. No entanto, o desenvolvedor tem de fornecer uma implementação parcial para o programa que será completada pelo processo de síntese.

3.3 VS3 —Verification and Synthesis using SMT Solvers

VS3 é uma ferramenta usada para verificar propriedades complexas de programas e inferir invariantes para esses programas. O usuário provê um conjunto de predicados e padrões de invariantes e o VS3 encontra, automaticamente, instâncias concretas de invariantes baseadas nos padrões derivadas a partir do conjunto de predicados fornecidos (Srivastava, Gulwani, & Foster, 2009).

A aplicação é especificamente criada para derivar e sintetizar invariantes de programa e requer especificações complexas sobre o universo de possíveis invariantes e predicados necessários para o processo de síntese.

3.4 Microsoft PROSE

Microsoft *Program Synthesis using Examples* SDK[12], também conhecido como Microsoft PROSE SDK, é um *framework* que contém um conjunto de ferramentas e tecnologias usadas para programação através de exemplos. Dado uma linguagem específica para o domínio (*domain-specific language*, ou DSL) e exemplos de entradas e saídas esperadas para o programa, PROSE sintetiza um conjunto de programas derivados a partir da DSL que são consistentes com os exemplos dados (Polozov & Gulwani, 2015).

Uma DSL para PROSE é composta de três elementos: a sintaxe, a semântica e um conjunto de funções "testemunho", do inglês *witness functions*. A sintaxe especifica uma gramática que será usada para estruturar os programas sintetizados. A semântica implementa o comportamento das funções e termos terminais da gramática. Por último, as funções "testemunho" especificam uma "semântica inversa" para os operadores das funções da gramática e são usadas para auxiliar e acelerar o processo de síntese. Dados essa DSL e o conjunto de exemplos, PROSE sintetiza um conjunto de programas derivados da DSL que são consistentes com os exemplos dados. Além disso, uma função de *rank* pode ser especificada pelo usuário para ranquear automaticamente os programas retornados.

A principal vantagem de PROSE é que ele é capaz de sintetizar programas completos baseado em exemplos de entrada e saída. Assim, um usuário leigo pode fornecer esses exemplos e é capaz de utilizar a ferramenta para gerar programas sem necessariamente conhecer como a linguagem do programa gerado funciona. Por outro lado, construir a DSL necessária para o processo de síntese, especificamente as funções testemunho, requer conhecimento da semântica da linguagem a ser sintetizada.

3.5 Evolução gramatical

Evolução gramatical, do inglês *gramatical evolution*, é uma técnica de inteligência artificial baseada em algoritmos genéticos, especificamente em programação genética. Ela tem como objetivo gerar um programa executável que atenda uma função objetivo. Operadores genéticos são aplicados a uma string de inteiros, que é mapeada para um programa através de uma gramática (O'Neill & Ryan, 2012).

As principais vantagens de abordagens que utilizam evolução gramatical são: múltiplas linguagens podem ser usadas, desde que o mapeamento entre o gene e gramática da linguagem desejada seja definido, junto com a semântica da linguagem; e programas completos conseguem ser sintetizados sem necessidade de uma implementação parcial. A principal limitação é que a especificação do programa deve ser dada através de uma função objetivo, o que pode não ser simples para um usuário.

3.6 Regressão simbólica

Regressão simbólica é um tipo de análise regressiva que pesquisa um espaço de expressões matemáticas para encontrar um modelo que melhor se encaixa em um *dataset* que contém os exemplos de entrada e saída esperadas para o modelo (Vladislavleva,

[12] Disponível em: https://microsoft.github.io/prose/

Smits, & Hertog, 2008). A síntese ocorre a partir da combinação, através de programação genética, de elementos básicos como operadores matemáticos, constantes e variáveis.

A principal vantagem dessa técnica é que ela não depende de um modelo parcial fornecido ao processo de síntese, que é capaz de se basear apenas nos elementos básicos disponíveis para o processo e no *dataset* fornecido pelo usuário.

3.7 Comparação entre as ferramentas

Os quadros 1 e 2 mostram uma comparação esquemática das técnicas apresentadas nesse capítulo. Dentre elas, apenas PROSE, evolução gramatical e regressão simbólica são capazes de gerar programas completos, as outras técnicas geram apenas expressões, constantes ou invariantes para programas. Além disso, apenas PROSE e Evolução Gramatical recebem exemplos de entrada e saída diretamente durante o processo de síntese. Por último, vale notar que a maioria das técnicas analisadas são genéricas o suficiente para suportar síntese de programas em várias linguagens.

O sintetizador desenvolvido em Alloy como parte deste trabalho utiliza síntese indutiva, com especificações baseadas em exemplos ou asserções, é genérico o suficiente para suportar múltiplas linguagens e é capaz de sintetizar constantes, expressões e programas.

Quadro 1: Comparação esquemática entre técnicas de síntese quanto a técnica e especificação

	Técnica utilizada	**Especificação**
SKETCH	Guiada por *schemas*	Asserções
Rosette	Guiada por *schemas*	Asserções
VS³	Dedutiva	Padrões de invariantes
PROSE	Indutiva	Exemplos
Evolução gramatical	Programação genética	Função objetivo
Regressão simbólica	Programação genética	Exemplos

Quadro 2: Comparação esquemática entre técnicas de síntese de programas quanto ao paradigma e a capacidade de síntese

	Paradigma	**Capacidade de síntese**
SKETCH	Imperativo	Constantes
Rosette	Multi-linguagem	Expressões
VS³	Invariantes para programas	Expressões
PROSE	Multi-linguagem	Programas
Evolução gramatical	Multi-linguagem	Programas
Regressão simbólica	Multi-linguagem	Programas

4 – Alloy

Neste capítulo será apresentada a linguagem Alloy, uma linguagem declarativa para modelagem e análise de sistemas de software publicada no periódico TOSEM por Daniel Jackson em 2002 (Jackson, 2002) e depois com mais detalhes em seu livro "*Software abstractions: logic, language, and analysis*" (Jackson, 2006). Este capítulo apresenta uma breve descrição dessa linguagem baseada no livro publicado em 2006.

Alloy é uma linguagem baseada em lógica de primeira ordem e álgebra relacional. Sua sintaxe permite expressividade suficiente para modelar e capturar detalhes de sistemas reais, mas também é simples o suficiente para suportar análises automáticas.

Ela provê três tipos de análise sobre modelos: simulações, verificações e *debugging*. O analisador Alloy consegue *simular* um sistema através da procura exaustiva por uma instância que atenda as especificações do modelo em análise. Ele também consegue *verificar* alguma determinada propriedade no sistema através da procura por um contraexemplo que não atenda a propriedade sendo verificada. Finalmente, o analisador pode ajudar na análise de erros, ou *debugging*, de uma especificação indicando quais restrições do modelo são conflitantes.

A versatilidade da linguagem, junto com a análise automática, motivou seu uso em várias aplicações, tanto práticas quando exploratórias. Exemplos incluem design e modelagem de sistemas, verificação de programas, geração automática de casos de teste, extração de especificações, geração de contraexemplos e configuração declarativa.

A lógica de Alloy é baseada num único conceito: relação. Funções são tratadas como relações binárias, conjuntos são relações unárias e escalares são relações unárias com um único membro. Uma especificação Alloy é uma coleção de restrições de primeira ordem sobre variáveis relacionais. As tuplas que são representadas por essas variáveis pertencem a um universo de elementos não interpretados, conhecidos como átomos.

Nas seções a seguir serão apresentados o subconjunto da sintaxe e da semântica de Alloy relevantes para esse trabalho.

4.1 Sintaxe

As principais estruturas sintáticas de Alloy podem ser divididas em quatro grupos: declarações de assinaturas e relações, que definem o universo de possíveis instâncias do modelo; sentenças de especificação de restrições e propriedades sobre o modelo; comandos para a execução de análises, verificações e definição do escopo dessas execuções; e finalmente sentenças para modularização e reuso de modelos.

4.1.1 Assinaturas e relações

Assinaturas são os elementos básicos de um modelo Alloy. Elas definem os conjuntos que compõem o universo a ser explorado pelo analisador.

4.1.1.1 sig

Uma *sig*, ou declaração de assinatura, é a palavra-chave usada para representar um conjunto de átomos. Um átomo, como esperado, é um elemento indivisível do modelo. Uma assinatura também pode conter campos, cada um deles representando uma relação entre assinaturas. O conjunto de assinaturas e suas relações define o conjunto universo

daquele modelo. A Figura 8 mostra exemplos de três assinaturas em Alloy para um modelo de um livro de endereços.

```
sig Nome {}
sig End {}
sig Livro { endereco: Nome -> End }
```

Figura 8: Alloy – Declaração de assinaturas

Esse modelo possui três assinaturas: **Nome,** representando o conjunto de nomes de pessoas e empresas; **End**, representando o conjunto de possíveis endereços**,** e **Livro**, representando o conjunto de livros de endereços. A assinatura **Livro** possui um campo *endereco* que define uma relação ternária entre **Livros**, **Nomes** e **Endereços**, representando o mapeamento entre esses.

4.1.1.2 extends

A palavra extends é usada para declarar que uma assinatura é um subconjunto de outra assinatura. Continuando com o exemplo do livro de endereços, nomes de pessoas físicas podem ser diferenciados de nomes de pessoas jurídicas através de uma assinatura da seguinte maneira:

```
sig NomeFisico, NomeJuridico extends Nome{}
```

Figura 9: Alloy - Declaração de subconjuntos de assinaturas

A sentença acima define duas novas assinaturas, **NomeFisico** e **NomeJuridico**, que são subconjuntos da assinatura **Nome**.

4.1.1.3 abstract

```
abstract sig Nome {}
sig NomeFisico, NomeJuridico extends Nome{}
```

Figura 10: Alloy - Declaração de assinaturas abstratas

Uma assinatura abstrata não contém elementos exceto por aqueles que estão contidos em seus subconjuntos. Se quisermos restringir o conjunto de nomes para que todas as instâncias pertençam ou a **NomeFisico** ou **NomeJuridico**, poderíamos reescrever a assinatura **Nome**, como na Figura 10, declarando-a como abstrata.

4.1.2 Funções, predicados, fatos e asserções

Um modelo em Alloy é definido tanto pelas suas assinaturas quanto pelo conjunto de restrições e propriedades impostas sobre esse modelo. As palavras-chave apresentadas nessa seção são usadas para definir essas propriedades.

4.1.2.1 fact

Palavra-chave usada para definir propriedades sobre o modelo que devem sempre ser respeitadas. Na Figura 11 temos um modelo sobre estações de rádio. O fato **NoOverlapping** define que duas estações de rádio disjuntas não podem compartilhar uma mesma banda de frequência. Além disso, por ser um fato, essa propriedade deve ser respeitada em todas as instâncias do modelo.

```
sig RadioStation {band: set Freq}
fact NoOverlapping {
    no disj s, s': RadioStation | some s.band & s'.band
}
```

Figura 11: Alloy - declaração de fatos de um modelo

4.1.2.2 fun

Usada para definir funções sobre conjuntos ou elementos do modelo. Uma função possui zero ou mais argumentos e simplesmente atribui um nome para uma expressão. Quando usada, expressões devem ser dadas para cada argumento. O significado da função é a expressão que ela representa, com cada ocorrência dos argumentos substituída pelas expressões dadas. A Figura 12, por exemplo, define uma função *plus* que recebe dois argumentos do tipo **`Int`** e representa a soma de ambos através de uma expressão também do tipo **`Int`**.

```
fun plus [n1, n2: Int] : Int { n1 fun/add n2 }
```

Figura 12: Definição de funções em Alloy

4.1.2.3 pred

Usada para definir propriedades sobre o modelo. Diferente de *facts*, restrições definidas em predicados só precisam ser respeitadas quando explicitado que o predicado deve ser verdadeiro. Um predicado possui zero ou mais argumentos e simplesmente atribui um nome para uma propriedade sobre o modelo. Quando usado, uma expressão deve ser dada para cada argumento. O significado do predicado é a propriedade que ele representa, com cada ocorrência dos argumentos substituída pelas expressão dadas. No exemplo apresentado na Figura 13, um predicado *pos*, que indica se um número inteiro é positivo, é definido.

```
pred pos  [n: Int] { n > 0 }
```

Figura 13: Definição de predicado em Alloy

4.1.2.4 assert

Usada para definir asserções para o modelo. Asserções poder ser invocadas para analisar uma propriedade ou restrição que deve ser respeitada de acordo com os fatos definidos no modelo. O analisador Alloy é capaz de checar asserções e, caso elas não sejam respeitadas, apresentar um contraexemplo que insatisfaz a asserção. A Figura 14 mostra uma asserção para um modelo cuja cardinalidade do conjunto de **`Livros`** deve ser maior que zero.

```
assert naoVazio { #Livro > 0 }
```

Figura 14: Alloy - definição de asserção

4.1.3 Comandos e escopo

Com o modelo definido, o analisador Alloy pode ser usado para executar verificações e análises através de checagem de modelos usando asserções e predicados. As palavras-chave apresentadas nessa seção são usadas para especificar esses comandos.

4.1.3.1 check

Usada para executar asserções. O analisador Alloy procura alguma instância do modelo em questão que não atenda a propriedade definida pela asserção sendo analisada. Caso encontrada, a instância serve como um contraexemplo para a asserção e é apresentada para o usuário. Caso não encontre nenhuma instância, o analisador emite uma mensagem dizendo que nenhum contraexemplo foi encontrado. Na Figura 15, a asserção definida na Figura 14 é invocada com o escopo padrão do analisador.

```
check naoVazio
```

Figura 15: Alloy - declaração de check de asserções

4.1.3.2 run

Usada para executar predicados. O Alloy Analyser procura um conjunto de instâncias para os conjuntos definidos no modelo que atendam as propriedades definidas no modelo e no predicado sendo executado. Se tal conjunto for encontrado, ele é apresentado ao usuário. Caso contrário, o analisador emite uma mensagem informando ao usuário que nenhuma instância foi encontrada e que o modelo pode ser inconsistente. Não há garantia sobre inconsistência porque o analisador explora apenas um subconjunto limitado (*bounded)* do universo de possíveis exemplos definido pelo modelo Alloy

4.1.4 Modularização

A linguagem Alloy permite organização de modelos em módulos, que podem ser reusados em outros modelos. As palavras-chave apresentadas nessa seção são usadas para especificar nomes de módulos e indicar reuso.

4.1.4.1 module

A palavra *module* é usada para definir nomes de módulos. Cada módulo em Alloy possui um *module* header formado pelo caminho até o arquivo dentro do sistema de arquivos e do nome do arquivo.

4.1.4.2 open

Utilizada para reuso de conjuntos, predicados, funções, etc que foram definidos em outro arquivo, semelhante a palavra *import* de outras linguagens como C.

4.2 Semântica e operadores

4.2.1 Conjuntos, átomos e relações

Todas as estruturas de Alloy são construídas a partir de átomos e relações. Átomos são as entidades básicas de um modelo, enquanto relações expressam relacionamento entre átomos.

Como um exemplo, a Figura 16 mostra um modelo de uma lista de endereços.

```
sig Nome {}
sig End {}
sig Livro {}
```

Figura 16: Alloy - Um modelo simples de um livro de endereços

Uma das possíveis instâncias desse modelo é formada pelos seguintes conjuntos:

$$Nome = \{(N0), (N1), (N2)\}$$
$$End = \{(E0), (E1), (E2)\}$$
$$Livro = \{(L0), (L1), (L2)\}$$

Cada um dos conjuntos **`Nome`**, **`End`** e **`Livro`** é tratado como uma relação unária em Alloy, ou seja, os elementos desses conjuntos são tuplas com um único elemento. Cada um desses elementos, como N0, N1 e L2, é um átomo do modelo.

Vale notar, Alloy trata todas as assinaturas e relações entre assinaturas como relações independentes. Por exemplo, podemos modificar o exemplo do livro de endereços para capturar o mapeamento entre nomes e endereços em um livro da seguinte maneira:

```
sig Nome {}
sig End {}
sig Livro { endereco: Nome -> End }
```

Figura 17: Alloy - Um modelo de um livro de endereços

Uma das possíveis instâncias desse modelo é formada pelos seguintes conjuntos:

$$Nome = \{(N0), (N1), (N2)\}$$
$$End = \{(E0), (E1), (E2)\}$$
$$Livro = \{(L0), (L1), (L2)\}$$
$$endereço = \{(L0, N0, E1), (L0, N1, E0), (L1, N1, E0)\}$$

O campo *endereço* da assinatura Livro é interpretado pelo Alloy como uma relação ternária da forma $endereço: Livro \rightarrow Nome \rightarrow End$. Essa interpretação, junto com os operadores relacionais de Alloy, acaba fornecendo um poderoso potencial de manipulação de conjuntos.

4.2.2 Constantes

Alloy possui três constantes:

- **none**, o conjunto vazio ;
- **univ**, o conjunto universo, contendo todos os átomos e relações do modelo;
- **iden**, a função identidade, mapeando assinaturas para elas mesmas.

4.2.3 Operadores sobre conjuntos

Os operados definidos em Alloy que atuam sobre conjuntos são:

- +, união de conjuntos
- &, intercessão de conjuntos
- -, diferença de conjuntos
- **in**, subconjunto, equivalente ao operador matemático $\subset$
- =, igualdade entre conjuntos

4.2.4 Operadores sobre relações

Por último, Alloy provê operadores que operam especificamente sobre relações:

- → produto cartesiano
- . composição de relações, semelhante ao operador matemático ∘
- [] composição de relações inversa
- ~ relação transposta
- ^ fecho transitivo
- * fecho transitivo-reflexivo
- <: restrição de domínio
- :> restrição de imagem
- ++ substituição

4.3 O analisador Alloy

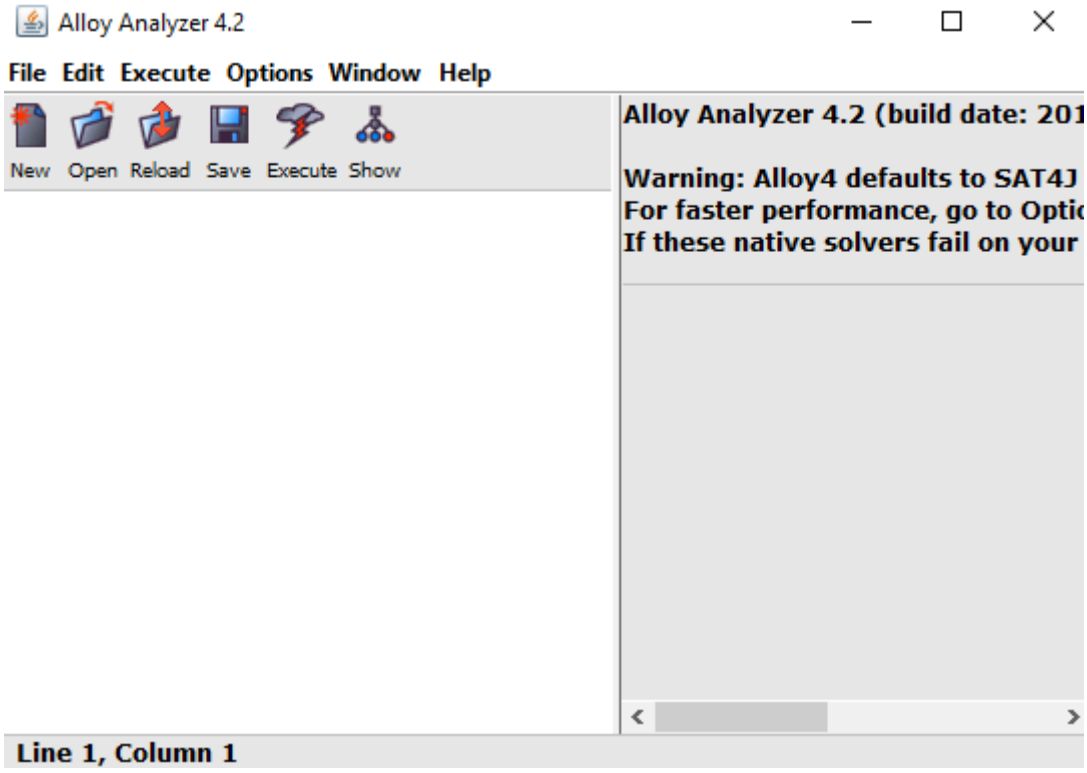

Figura 18: O analisador Alloy

O Alloy Analyser é uma aplicação Java crida para desenvolver modelos em Alloy e analisar propriedades desses modelos. Uma captura de tela da janela principal está disponível na Figura 18. Ele é composto de um painel de edição que ocupa a maior área da janela, um painel na lateral direita que mostra detalhes sobre execuções de comandos e erros de compilação, menus no topo da janela para configuração e funcionalidades detalhadas e uma barra de tarefas com as funções básicas do programa.

Para todas as análises feitas pelo analisador, o modelo escrito em Alloy é transformado em uma sentença da lógica relacional. Um resolvedor de problemas de satisfatibilidade SAT é então usado para resolver a sentença. O resolvedor SAT dentro do Alloy Analyser é *pluggable*, ou seja, pode ser substituido sem dificuldade ou alteração

do processo através de um dos menus de configuração. Isso permite que o Alloy se beneficie de avanços na área de resolução de problemas SAT. Os resolvedores disponíveis por padrão na versão 4.2 do analisador Alloy são: MiniSat, ZChaff e SAT4J.

Além de executar análises, o analisador Alloy também gera diagramas para exemplificar instâncias do modelo em análise. A Figura 48 mostra um desses diagramas.

4.4 Alloy*

Alloy* é uma segunda linguagem desenvolvida por (Milicevic, Near, Kang, & Jackson, 2015) que estende a sintaxe de Alloy com o objetivo de suportar quantificadores de alta-ordem, do inglês *higher-order quantifiers*. Para tornar esses quantificadores analisáveis, Alloy* combina duas técnicas: síntese indutiva guiada por contraexemplos, do inglês *CounterExample-Guided Inductive Synthesis (*CEGIS) (Solar-Lezama, Tancau, Bodik, Seshia, & Saraswat, 2006); e o Kodkod[13], um resolvedor de problemas da lógica relacional.

Um exemplo de síntese de programas usando Alloy*, apresentado na Figura 19, está disponível no website do Alloy*[14]. Esse exemplo, no entanto, não é diretamente aplicável em Alloy puro devido aos quantificadores de alta-ordem. Um sintetizador semelhante a esse exemplo, mas que é aplicável para Alloy puro, foi desenvolvido por este trabalho e é apresentado no Capítulo 6.

```
abstract sig Bool {}
one sig BoolTrue extends Bool {}
one sig BoolFalse extends Bool {}

abstract sig Node {}
abstract sig BoolNode extends Node {}
abstract sig IntNode  extends Node {}
abstract sig Var      extends IntNode {}

abstract sig IntLit   extends IntNode { val: one Int }
abstract sig IntComp  extends BoolNode {
  left, right: IntNode
}
abstract sig BoolComp extends BoolNode {
  left, right: BoolNode
}

sig EQ, GTE, LTE, GT, LT extends IntComp {}
sig And, Or extends BoolComp {}
sig Not     extends BoolNode { arg: BoolNode }
sig ITE     extends IntNode {
  condition: BoolNode, then, elsen: IntNode
}
```

```
pred synth[root: IntNode] {
  all env: Var -> one Int |
    some eval: Node -> one (Int+Bool) when env in eval && semant
      spec[root, eval]
}
```

```
pred semantics[eval: Node -> (Int + Bool)] {
  all n: ITE {
    eval[n] in Int
    eval[n.condition] in Bool && eval[n.then] in Int && eval[n.elsen] in Int
    eval[n.condition] = BoolTrue implies
      eval[n.then] = eval[n] else eval[n.elsen] = eval[n]
  }

  all n: BoolNode | eval[n] in Bool
  all n: IntComp  | eval[n.left] in Int and eval[n.right] in Int
  all n: BoolComp | eval[n.left] in Bool and eval[n.right] in Bool
  all v: Var      | one eval[v] and eval[v] in Int
  all i: IntLit   | one eval[i] and eval[i] in Int and eval[i] = i.val

  all n: Not  | eval[n.arg] in Bool
  all n: EQ   | eval[n.left] = eval[n.right] implies
                  eval[n] = BoolTrue else eval[n] = BoolFalse
  all n: GTE  | eval[n.left] >= eval[n.right] implies
                  eval[n] = BoolTrue else eval[n] = BoolFalse
  all n: LTE  | eval[n.left] <= eval[n.right] implies
                  eval[n] = BoolTrue else eval[n] = BoolFalse
  all n: GT   | eval[n.left] > eval[n.right] implies
                  eval[n] = BoolTrue else eval[n] = BoolFalse

  all n: LT   | eval[n.left] < eval[n.right] implies
                  eval[n] = BoolTrue else eval[n] = BoolFalse

  all n: And  | (eval[n.left] = BoolTrue and eval[n.right] = BoolTrue) implies
                  eval[n] = BoolTrue else eval[n] = BoolFalse
  all n: Or   | (eval[n.left] = BoolTrue or eval[n.right] = BoolTrue) implies
                  eval[n] = BoolTrue else eval[n] = BoolFalse
  all n: Not  | eval[n.arg] = BoolFalse implies
                  eval[n] = BoolTrue else eval[n] = BoolFalse
}
```

*Figura 19 - Um exemplo de síntese de programas em Alloy**
Fonte: http://alloy.mit.edu/alloy/hola/, acesso em Julho de 2017

[13] Disponível em: http://emina.github.io/kodkod/
[14] Disponível em: http://alloy.mit.edu/alloy/hola/

5 – IMP

A linguagem IMP é uma linguagem imperativa simples semelhante a C apresentada por Glynn Winskel em 1993. Esse capítulo apresenta a sintaxe e a semântica dessa linguagem como descrita em (Winskel, 1993). O sintetizador criado por esse trabalho, introduzido no Capítulo 6, gera programas da linguagem IMP.

5.1 Sintaxe

Os conjuntos sintáticos associados a IMP são:

- Os números **N**, que contém os números inteiros positivos, negativos e zero;
- Os valores booleanos **T** = {**true, false**};
- Os registros **Loc**, semelhantes a variáveis de programa;
- As expressões aritméticas **AExp;**
- As expressões booleanas **BExp**
- E os comandos **Com**.

No contexto desse trabalho, os átomos do conjunto **N** correspondem aos elementos do conjunto de números decimais inteiros $\mathbb{Z}$, enquanto os átomos do conjunto **Loc** são as cadeias de caracteres não vazias, ou cadeias de caracteres não vazias seguidas de números.

O restante da sintaxe abstrata é definido através de uma gramática numa notação semelhante ao formalismo de Backus-Naur, do inglês *Backus-Naur Form* (BNF), apresentada abaixo:

Para o conjunto **AExp:**

$$a ::= \ n \mid X \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 \times a_1$$

Para o conjunto **BExp**:

$$b ::= \ \boldsymbol{true} \mid \boldsymbol{false} \mid a_0 = a_1 \mid a_0 \leq a_1 \mid \neg b \mid b_0 \wedge b_1 \mid b_0 \vee b_1$$

Para o conjunto **Com:**

$$c ::= \ \boldsymbol{skip} \mid X := \ a \mid c_0; c_1 \mid \boldsymbol{if}\ b\ \boldsymbol{then}\ c_0\ \boldsymbol{else}\ c_1 \mid \boldsymbol{while}\ b\ \boldsymbol{do}\ c$$

Onde:

- *a, b* e *c* são símbolos não-terminais e representam os conjuntos **AExp**, **BExp** e **Com**, respectivamente.
- *n* é um símbolo terminal que representa um número **N**,
- *X* é um símbolo terminal que representa um registro **Loc**,
- a_i representam expressões aritméticas do conjunto **AExp**,
- b_i representam expressões booleanas do conjunto **BExp**,
- c_i representam comandos do conjunto **Com.**

5.2 Semântica

A semântica de IMP é composta da definição de um estado de programa, definições de como expressões aritméticas e booleanas devem ser avaliadas; e definições

de como comandos influenciam os estados de um programa. Essa semântica pode ser definida tanto de forma operacional como denotacional. Nesta seção será apresentada a semântica denotacional de IMP, que, por ser expressa através de funções, corresponde diretamente a estruturas da linguagem Alloy utilizada para criar o sintetizador apresentado no Capítulo 6.

5.2.1 Estados de programa

Em IMP, um estado é definido como uma função $\sigma : \boldsymbol{Loc} \rightarrow \boldsymbol{N}$, que leva de registros para valores numéricos. Dessa maneira, $\sigma(X)$ corresponde ao conteúdo, ou valor, do registro *X* no estado σ. O conjunto de possíveis estados Σ de um programa é o conjunto dos possíveis estados σ.

As funções semânticas apresentadas a seguir são:

$$A : AExp \rightarrow (\Sigma \rightarrow N)$$
$$B : BExp \rightarrow (\Sigma \rightarrow T)$$
$$C : Com \rightarrow (\Sigma \rightarrow \Sigma)$$

Elas são apresentadas através de funções parciais da forma $A[\![a]\!]$, onde a é uma expressão gerada a partir da sintaxe abstrata apresentada anteriormente.

5.2.2 Avaliações de expressões aritméticas

A avaliação de expressões aritméticas é realizada através de uma função $A : AExp \rightarrow (\Sigma \rightarrow N)$, definida de maneira parcial nesta seção.

A avaliação de números correspondem sempre ao mesmo valor, para todos os estados possíveis de Σ:

$$A[\![n]\!] = \{(\sigma, n) \mid \sigma \in \Sigma\}$$

A avaliação de registro correspondem sempre ao valor correspondente aquele registro em cada possível estado de Σ:

$$A[\![X]\!] = \{(\sigma, \sigma(X)) \mid \sigma \in \Sigma\}$$

A avaliação de somas, subtrações e multiplicações corresponde ao resultado da soma, subtração e multiplicação, respectivamente, das avaliações de seus operandos:

$$A[\![a_0 + a_1]\!] = \{(\sigma, n_0 + n_1) \mid (\sigma, n_0) \in A[\![a_0]\!] \,\&\, (\sigma, n_1) \in A[\![a_1]\!]\}$$
$$A[\![a_0 - a_1]\!] = \{(\sigma, n_0 - n_1) \mid (\sigma, n_0) \in A[\![a_0]\!] \,\&\, (\sigma, n_1) \in A[\![a_1]\!]\}$$
$$A[\![a_0 \times a_1]\!] = \{(\sigma, n_0 \times n_1) \mid (\sigma, n_0) \in A[\![a_0]\!] \,\&\, (\sigma, n_1) \in A[\![a_1]\!]\}$$

5.2.3 Avaliações de expressões booleanas

A avaliação de expressões booleanas é realizada através de uma função $B : BExp \rightarrow (\Sigma \rightarrow T)$, definida de maneira parcial nesta seção.

A avaliação de literais booleanos corresponde aos seus valores em todos os possíveis estados de Σ:

$$B[\![\boldsymbol{true}]\!] = \{(\sigma, \boldsymbol{true}) \mid \sigma \in \Sigma\}$$

$$B[\![\boldsymbol{false}]\!] = \{(\sigma, \boldsymbol{false}) \mid \sigma \in \Sigma\}$$

A avaliação de comparações aritméticas corresponde a comparação das avaliações aritméticas de seus operandos nos respectivos estados de Σ:

$$B[\![a_0 = a_1]\!] = \{(\sigma, \boldsymbol{true}) \mid \sigma \in \Sigma \,\&\, A[\![a_0]\!]\sigma = A[\![a_1]\!]\sigma\}\, U \\ \{(\sigma, \boldsymbol{false}) \mid \sigma \in \Sigma \,\&\, A[\![a_0]\!]\sigma \neq A[\![a_1]\!]\sigma\}$$

$$B[\![a_0 \leq a_1]\!] = \{(\sigma, \boldsymbol{true}) \mid \sigma \in \Sigma \,\&\, A[\![a_0]\!]\sigma \leq A[\![a_1]\!]\sigma\}\, U \\ \{(\sigma, \boldsymbol{false}) \mid \sigma \in \Sigma \,\&\, A[\![a_0]\!]\sigma \nleq A[\![a_1]\!]\sigma\}$$

A avaliação de operações booleanas corresponde ao resultado da operação aplicada sobre a avaliação booleana de seus operandos nos respectivos estados de Σ:

$$B[\![\neg b]\!] = \{(\sigma, \neg_T t) \mid \sigma \in \Sigma \,\&\, (\sigma, t) \in B[\![b]\!]\}$$

$$B[\![b_0 \wedge b_1]\!] = \{(\sigma, t_0 \wedge_T t_1) \mid \sigma \in \Sigma \,\&\, (\sigma, t_0) \in B[\![b_0]\!] \,\&\, (\sigma, t_1) \in B[\![b_1]\!]\}$$

$$B[\![b_0 \vee b_1]\!] = \{(\sigma, t_0 \vee_T t_1) \mid \sigma \in \Sigma \,\&\, (\sigma, t_0) \in B[\![b_0]\!] \,\&\, (\sigma, t_1) \in B[\![b_1]\!]\}$$

5.2.4 Execuções de comandos

A execução de comandos e como eles afetam os estados de um programa é definido através da função $C : Com \rightarrow (\Sigma \rightarrow \Sigma)$, apresentada abaixo:

A execução do comando ***skip*** não influencia em qualquer estado do programa:

$$C[\![\boldsymbol{skip}]\!] = \{(\sigma, \sigma) \mid \sigma \in \Sigma\}$$

A execução de atribuição corresponde ao estado resultante da substituição do valor associado ao registro especificado:

$$C[\![X := a]\!] = \{(\sigma, \sigma[m/X]) \mid \sigma \in \Sigma \,\&\, \mathrm{n} = A[\![a]\!]\sigma\}$$

A execução de composições corresponde a composição funcional dos comandos:

$$C[\![c_0; c_1]\!] = C[\![c_1]\!] \circ C[\![c_0]\!]$$

A execução de comandos condicionais corresponde ao resultado da execução dos seus operandos de acordo com a condição:

$$C[\![\boldsymbol{if}\ b\ \boldsymbol{then}\ c_0\ \boldsymbol{else}\ c_1]\!] = \{(\sigma, \sigma') \mid B[\![b]\!]\sigma = \boldsymbol{true} \,\&\, (\sigma, \sigma') \in C[\![c_0]\!]\}\, U \\ \{(\sigma, \sigma') \mid B[\![b]\!]\sigma = \boldsymbol{false} \,\&\, (\sigma, \sigma') \in C[\![c_1]\!]\}$$

6 – Síntese de programas em Alloy

O sintetizador desenvolvido por esse trabalho utiliza a linguagem Alloy e sua poderosa orquestra de estruturação de modelos para modelar a sintaxe e semântica da linguagem IMP. Com isso, ele é capaz de gerar programas imperativos baseados em especificações através de contratos ou exemplos de entrada e saída.

O modelo utilizado para criar o sintetizador é composto de quatro especificações: o módulo de síntese que contém entidades genéricas, um módulo que modela a sintaxe da linguagem sintetizada, um módulo que modela a semântica da linguagem sintetizada e um módulo de restrições sobre a estrutura e propriedades dos programas sintetizados.

Na próxima seção será detalhado o módulo de síntese, que pode ser reusado para modelagem de outras linguagens. As seções seguintes descrevem os módulos de sintaxe e de semântica para a linguagem IMP desenvolvidos como parte do objetivo desse trabalho.

Vale notar que a mesma abordagem utilizada para construir o sintetizador IMP pode ser aplicada para criar sintetizadores para outras linguagens, desde que a sintaxe e semântica dessas linguagens sejam modeladas em Alloy. O módulo de síntese apresentado na próxima seção pode ser diretamente reusado para esse propósito.

6.1 Módulo de síntese

O módulo *synt/main.als* corresponde ao módulo principal de síntese. Nele são definidas as estruturas genéricas para definição de linguagens em Alloy e predicados de síntese que podem ser usados para criar sintetizadores para outras linguagens de programação.

A Figura 20 mostra o meta-modelo gerado pelo analisador Alloy sobre as estruturas definidas nesse módulo.

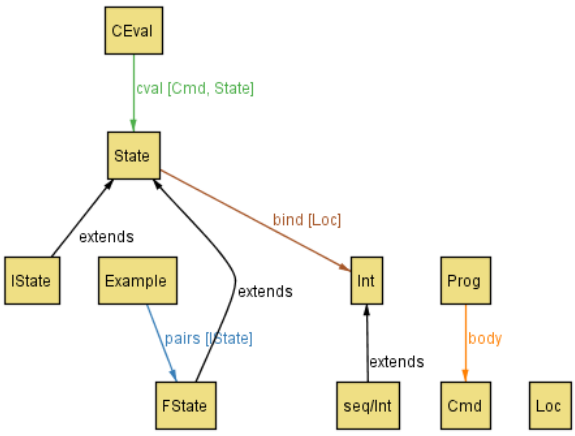

Figura 20: Meta-modelo das estruturas básicas para definição de linguagens

6.1.1 Estruturas genéricas para definição de linguagens

```
1  -- States
2  // An integer variable (X from Loc)
3  abstract sig Loc { }
4  // A generic State maping Locs to ints
5  sig State { bind: Loc -> one Int}

6  -- Syntax
7  // A generic Command (Statement), the root of the ast
8  abstract sig Cmd { }
9  // Main program, contains the root of the ast
10 one sig Prog { body: one Cmd }

11 -- Semantics
12 // A semantic evaluation of commands on states
13 sig CEval {cval: Cmd ->  State -> State} {}
14 fact { #CEval = 1}
```

Figura 21: Estruturas genéricas para definição de linguagens

A Figura 21 apresenta a primeira seção desse módulo, que descreve estruturas genéricas para definição de linguagens. Ela possui declarações de assinaturas divididas em três subseções: estados de programas, sintaxe e semântica de linguagens.

Para estados de programa, a entidade básica é a assinatura **`Loc`**, definida na linha 3, que representa o conjunto de variáveis de programa ou espaços de memória. Em seguida, na linha 5, é definida a assinatura **`State`**, que representa o conjunto de possíveis estados de um programa, que também pode ser visto como uma relação entre variáveis e valores concretos, realizado pelo seu campo `bind`. Neste trabalho, o conjunto de valores concretos que podem ser mapeados em variáveis é equivalente ao conjunto dos número inteiros. Strings ou outros átomos podem ser usados, mas para isso eles devem estar disponíveis na biblioteca padrão de Alloy ou devem ser modelados pelo desenvolvedor.

Para sintaxe de linguagens, são definidas as assinaturas **`Cmd`** e **`Prog`** nas linhas 8 e 10, respectivamente. **`Cmd`** representa a raiz da árvore sintática abstrata de uma linguagem. Em outras palavras, linguagens modelados nesse sintetizador devem possuir a assinatura **`Cmd`** como regra inicial de suas gramáticas. **`Prog`** é a abstração do conjunto de programas pertencentes à linguagem e possui um campo `body`, que mapeia programas aos seus comandos iniciais.

Por último, uma única assinatura **`CEval`** é definida para especificar a semântica de linguagens. Essa assinatura contém o campo `cval`, que tem o papel de uma função que mapeia comandos e estados iniciais a estados finais resultantes da aplicação de um comando em um determinado estado inicial. As linguagens mapeadas nesse sintetizador devem definir o comportamento da relação `cval`.

6.1.2 Predicados de síntese

```
1  -- Program synthesis by examples
2  // Input states
3  sig IState extends State {}
4  // Output states
5  sig FState extends State {}
6  // Input -> Output mapping
7  one sig Example {
8      pairs: IState -> one FState
9  }
10 // Predicate
11 pred syntByExample[p: Prog] {
12     all iSt: State, fSt: State |
13       iSt -> fSt in Example.pairs implies
14         Prog.body -> iSt -> fSt in CEval.cval
15 }

16 -- Program synthesis by contracts
17 // Predicate.
18 pred syntByContract[p: Prog] {
19     all iSt:  State |  (PreC[iSt] and
20       PosC[iSt, CEval.cval[p.body][iSt]])
21 }
```

Figura 22: Predicados para síntese de programas em Alloy

Na segunda seção do módulo de síntese, apresentada na Figura 22, são definidos os predicados para síntese de programas através de exemplos e de contratos.

Nas linhas 3 e 4 são definidas as assinaturas **`IState`** e **`FState`**, que representam o conjunto de estados de entrada e saída de programas, respectivamente. Esses estados são associados entre si através da relação `pairs` na assinatura **`Example`**, definida nas linhas de 7 a 9, que representa o conjunto de exemplos fornecidos pelo usuário para o processo de síntese. Especificações de comportamento que usam modelos devem definir subconjuntos ou instâncias de **`Example`** que contenham os pares de entrada e saída corretamente mapeados. Finalmente, nas linhas de 11 a 15, o predicado de síntese é definido. A restrição que esse predicado impõe sobre o modelo é que todos os pares de entradas e saídas estejam presentes na função semântica representada pela assinatura **`CEval`**. Assim, todos os estados iniciais serão usados durante a síntese e, desde que a função semântica tenha sido especificada corretamente, o programa gerado tem a garantia de gerar as saídas corretas para as respectivas entradas fornecidas como exemplo.

Já nas linhas de 18 a 21, o predicado para síntese de programas baseado em contratos é definido. Ele utiliza outros dois predicados, `PreC` e `PosC`, que descrevem as restrições e propriedades desejadas sobre os estados iniciais e finais do programa, respectivamente. Ambos `PreC` e `PosC` devem ser escritos pelo usuário como parte da especificação do comportamento do programa a ser sintetizado através de contratos.

6.2 Módulo de sintaxe

O módulo *sintax.als* contém a modelagem da sintaxe abstrata da linguagem a ser sintetizada. Com isso, os elementos da gramática da linguagem são definidos e o universo de átomos que podem ser gerados é criado. Tomando como exemplo a linguagem IMP, o

módulo de sintaxe define elementos como **AExp**, representando regras de formação de expressões aritméticas e BExp, representando regras de formação de expressões booleanas. Além desses conjuntos, o módulo de sintaxe também define a estrutura dos relacionamentos entre esses elementos.

Nas subseções seguintes será apresentada a modelagem de um subconjunto da sintaxe da linguagem IMP como apresentada no Capítulo 5, com a adição de registros específicos.

Para os conjuntos sintáticos dos números **N** e o de valores booleanos **T**, são utilizadas as assinaturas **Int** e **Bool**, providas por padrão pelos módulos `util/integer` e `util/boolean` de Alloy. Resta definir os outros conjuntos sintáticos e as relações entre eles.

6.2.1 Registros

```
// Delta Loc (Variables that can be used everywhere)
sig DLoc extends Loc { }
// Aux Loc (Variables used in the left-hand side of assignments)
sig ALoc extends Loc { }
// Xi Loc (Variables that cannot be changed)
sig XLoc extends Loc { }
```

O primeiro e mais simples conjunto sintático é o de registros, ou variáveis, do programa. Para isso, a assinatura abstrata **Loc** definida no módulo de síntese é usada e os conjuntos **DLoc**, **ALoc** e **XLoc** são definidos. Variáveis do tipo **DLoc** não possuem restrições, mas variáveis dos tipos **ALoc** não podem ser usadas em expressões condicionais e normalmente são usadas como registros de atribuições auxiliares. Variáveis **XLoc**, por sua vez, não podem ter seus valores alterados durante a execução do programa.

Um usuário ao definir a especificação sobre um programa pode usar variáveis **XLoc** e **ALoc** para especificar entradas e saídas, respectivamente, e, por causa das restrições associadas a essas, acelerar o processo de síntese.

6.2.2 Expressões aritméticas

```
// Arithmetic expressions
abstract sig AExp { } // a ::=
sig IntVal extends AExp { val: one Int } // n
sig IntVar extends AExp { name: one Loc } // X
abstract sig BinAExp extends AExp{op1: one AExp, op2: one AExp}
sig Add extends BinAExp { } // a0+a1
sig Sub extends BinAExp { } // a0-a1
sig Mult extends BinAExp { } // a0*a1
```

Figura 23: Alloy - Sintaxe de expressões aritméticas em IMP

Como apresentado no Capítulo 5, o conjunto sintático das expressões aritméticas da linguagem IMP é definido através da seguinte regra de formação:

$$a ::= \ n \mid X \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 \times a_1$$

Na modelagem da sintaxe como apresentada na Figura 23, a assinatura abstrata **`AExp`** corresponde ao símbolo não terminal a da regra de formação, representando os elementos do conjunto de expressões aritméticas. As assinaturas concretas **`IntVal`**, **`IntVar`**, **`Add`**, **`Sub`** e **`Mult`** representam as cinco possíveis derivações n, X, $a_0 + a_1$, $a_0 - a_1$ e $a_0 \times a_1$ definidas na regra de formação, respectivamente. A assinatura **`IntVal`** possui um único campo `val` com tipo **`Int`** que contém seu valor inteiro n. A assinatura **`IntVar`** possui um único campo `name` com tipo **`Loc`** que representa qual registro X está associado àquela expressão. As assinaturas **`Add`**, **`Sub`** e **`Mult`** possuem os campos `op1` e `op2` do tipo **`AExp`** que correspondem aos operandos a_0 e a_1 da regra de formação.

6.2.3 Expressões booleanas

```
// Boolean expressions
abstract sig BExp { lhs, rhs: one AExp } {
    lhs.name != rhs.name and
    lhs.name !in ALoc and
    rhs.name !in ALoc }
sig EQ extends BExp {  }
sig NEQ extends BExp {  }
sig LEQ extends BExp {  }
sig GEQ extends BExp {  }
sig GTH extends BExp {  }
```

Figura 24: Alloy - Sintaxe de expressões booleanas em IMP

Como apresentado no Capítulo 5, o conjunto sintático das expressões booleanas da linguagem IMP é definido através da seguinte regra de formação:

$$b ::= \ \boldsymbol{true} \mid \boldsymbol{false} \mid a_0 = a_1 \mid a_0 \leq a_1 \mid \neg b \mid b_0 \wedge b_1 \mid b_0 \vee b_1$$

No entanto, na modelagem da sintaxe como apresentada na Figura 24, apenas a derivação $a_0 = a_1$ foi modelada na assinatura **`EQ`** (em relação a sintaxe definida na Seção 5.1) pois as demais derivações não foram explicitamente necessárias para os programas sintetizados como experimento. A assinatura abstrata **`BExp`** corresponde ao símbolo não terminal b, representando os elementos do conjunto de expressões booleanas e possui os campos `lhs` e `rhs` para indicar os respectivos operandos de cada regra.

Uma possível otimização para o processo de síntese é mapear expressões complexas que são usadas frequentemente em termos de operadores simples com uma semântica correspondente. Dessa maneira, apesar do universo de possíveis átomos aumentar, a complexidade da estrutura do modelo final diminui, facilitando o processo de síntese. Por essa causa, como parte deste trabalho, a sintaxe de IMP foi estendida com as seguintes expressões booleanas:

$$a_0 \neq a_1 \equiv \neg(a_0 = \ a_1)$$

$$a_0 \geq a_1 \equiv \neg(a_0 \leq \ a_1) \ \vee (a = b)$$

$$a_0 > a_1 \equiv \neg(a_0 \leq \ a_1)$$

$$a_0 < a_1 \equiv a_0 \leq a_1 \ \wedge \neg(a_0 = \ a_1)$$

Assim, a regra de formação de expressões booleanas passa a ser:

$$b ::= \ \textbf{\textit{true}} \mid \textbf{\textit{false}} \mid a_0 = a_1 \mid a_0 \leq a_1 \mid a_0 \geq a_1 | a_0 > a_1 | a_0 < a_1 |$$
$$a_0 \neq a_1 | \neg b \mid b_0 \wedge b_1 \mid b_0 \vee b_1$$

As novas regras criadas foram modeladas através das assinaturas **`NEQ`**, **`GEQ`**, **`GTH`** e **`LEQ`**, apresentadas na Figura 24.

6.2.4 Comandos

```
// A generic Command (Statement)
abstract sig Cmd { }
// The Skip statement
lone sig Skip extends Cmd { }
// Assignment statement
sig Assign extends Cmd { lhs: one IntVar, rhs: one AExp }
// Conditional statement (IF THEN ELSE)
sig CondS extends Cmd { cond: one BExp, then, elsen: one Cmd }
// Loop statement
sig While extends Cmd { wcond: one BExp, wbody: one (Cmd - While) }
// Sequential composition
sig SComp extends Cmd { cur, next: one Cmd }
```

Figura 25: Alloy - Sintaxe de comandos em IMP

O último conjunto sintático a ser definido é o conjunto que define comandos na linguagem IMP. Como mostrado no Capítulo 5, a regra de formação para comandos é:

$$c ::= \ \textbf{\textit{skip}} \mid X := a \mid c_0; c_1 \mid \textbf{\textit{if}}\ b\ \textbf{\textit{then}}\ c_0\ \textbf{\textit{else}}\ c_1 \mid \textbf{\textit{while}}\ b\ \textbf{\textit{do}}\ c$$

Na modelagem como apresentada na Figura 25, a assinatura abstrata **`Cmd`** corresponde ao símbolo não terminal c, representando os elementos do conjunto de comandos. A assinatura **`skip`** corresponde ao símbolo terminal **`skip`.** A assinatura **`Assign`** corresponde à regra de formação $X := a$ e possui os campos `lhs` e `rhs`, que correspondem aos símbolos X e a, respectivamente**.** A assinatura **`CondS`** corresponde à derivação $\textbf{\textit{if}}\ b\ \textbf{\textit{then}}\ c_0\ \textbf{\textit{else}}\ c_1$. Essa assinatura possui os membros: `cond`, `then` e `elsen`, que correspondem aos símbolos b, c_0 e c_1, respectivamente. A regra $\textbf{\textit{while}}\ b\ \textbf{\textit{do}}\ c$ é representada através da assinatura **`While`**, que possui o campo `wcond` correspondente ao símbolo b e o campo `wbody` correspondente ao símbolo c. Por último, a regra de composição $c_0; c_1$ é definida através da assinatura **`SComp`**, com os campos *cur* e *next*, do tipo **`Cmd`**, que correspondem aos símbolos c_0 e c_1.

6.3 Módulo de semântica

```
1  sig AEval {aval : AExp -> State -> Int} {}
2  sig BEval {bval: BExp -> State -> Bool} {}

3  fact semantics_fact {
4    #AEval = 1
5    #BEval = 1
6    all exp : AEval.aval.univ.univ, curSt: AEval.aval[exp].univ |
7  arithmetics_semantics[exp, curSt]
8    all exp: BEval.bval.univ.univ, curSt: BEval.bval[exp].univ |
9  boolean_semantics[exp, curSt]
10   all cmd: CEval.cval.univ.univ, curSt: CEval.cval[cmd].univ |
11 command_semantics[cmd, curSt]
12 }
```

Figura 26: Alloy - Fato da semântica da linguagem IMP

O módulo *semantics.als* define a semântica da linguagem a ser sintetizada. Para o modelo desenvolvido nesse trabalho, a semântica da linguagem IMP foi dividida em três partes: semântica sobre expressões aritméticas, semântica sobre expressões booleanas e semântica sobre comandos. Cada uma das partes define um predicado sobre a relação de avaliação de um estado com as respectivas expressões e comandos do universo definido pela sintaxe da linguagem. O fato `semantics_fact` apresentado na Figura 26 consolida os três predicados.

Nas linhas 1 e 2 são definidas a assinaturas que contêm as relações de semântica sobre expressões aritméticas e booleanas, respectivamente. As linhas 4 e 5 especificam que essas assinaturas possuem um único elemento. Assim, todas as relações de semântica partem de um mesmo átomo do modelo. As demais linhas iteram sobre as possíveis expressões, comandos e estados do modelo consolidando os três predicados que definem as semânticas sobre expressões aritméticas, expressões booleanas e comandos.

Nas linhas 6 e 7, por exemplo, o quantificador **`all`** restringe que todos os átomos de expressões `exp` e estados `curSt` presentes na instância do modelo devem respeitar o predicado `arithmetics_semantics`. Como definido na linha 6, as expressões `exp` pertencem ao conjunto **`AEval`**`.aval.univ.univ`, enquanto os estados `curSt` pertencem a **`AEval`**`.aval[exp].univ`. Como foi apresentado na Seção 4.2, os operadores ponto "." e colchetes "[-]" são os operadores de composição de relações em Alloy, enquanto `univ` é o conjunto universo que contém todas as relações e átomos do modelo. Em mais detalhes, a interpretação das expressões `exp` é a seguinte:

- `aval` é uma relação do tipo $AEval \rightarrow AExp \rightarrow State \rightarrow Int$
- **`AEval`**`.aval` é uma relação do tipo $AExp \rightarrow State \rightarrow Int$
- **`AEval`**`.aval.univ` é uma relação do tipo $AExp \rightarrow State$
- Finalmente, **`AEval`**`.aval.univ.univ` é um conjunto do tipo $AExp$, que contém todas as expressões aritméticas `exp` que fazem parte de alguma avaliação de estado de programa. Vale notar que o conjunto **`AEval`**`.aval.univ.univ` é sempre um subconjunto do conjunto **`AExp`**, mas eles podem ser diferentes caso existam expressões aritméticas na instância do modelo que não foram avaliadas e, logo, não estão mapeadas na relação de semântica.

De maneira semelhante, para os estados `curSt`:

- `aval` é uma relação do tipo $AEval \rightarrow AExp \rightarrow State \rightarrow Int$.
- **`AEval`**`.aval` é uma relação do tipo $AExp \rightarrow State \rightarrow Int$.
- **`AEval`**`.aval[exp]` é uma relação do tipo $State \rightarrow Int$.
- **`AEval`**`.aval[exp].univ` é um conjunto do tipo $State$, que contém todos os possíveis estados `curSt` que são avaliados por alguma expressão aritmética.

6.3.1 Semântica sobre expressões aritméticas

```
pred arithmetics_semantics[exp: AExp, curSt : State]{
  exp -> curSt in AEval.aval.Int implies
  exp in IntVal implies AEval.aval[exp][curSt] = exp.val else
  exp in IntVar implies
    AEval.aval[exp][curSt] = curSt.bind[exp.name] else
  exp in BinAExp implies
    let result1 = AEval.aval[exp.op1][curSt],
        result2 = AEval.aval[exp.op2][curSt] |
          some result1 and some result2 and (
          exp in Add implies
             AEval.aval[exp][curSt] = plus[ result1, result2] else
          exp in Sub implies
             AEval.aval[exp][curSt] = minus[ result1, result2] else
          exp in Mult implies
             AEval.aval[exp][curSt] = mul[ result1, result2]
          )
}
```

Figura 27: Alloy - Predicado de semântica sobre expressões aritméticas

A semântica sobre expressões aritméticas está definida para os operadores **`Add`**, **`Sub`**, **`Mult`** e para os valores atômicos **`IntVar`** e **`IntVal`** como na Figura 27. O predicado é formado por uma sequência de implicações que usam as funções semânticas definidas no Capítulo 5 para cada tipo de expressão.

6.3.2 Semântica sobre expressões booleanas

```
pred boolean_semantics[exp: BExp, curSt : State]{
  exp -> curSt in BEval.bval.Bool implies
  let result1 = AEval.aval[exp.lhs][curSt],
      result2 = AEval.aval[exp.rhs][curSt] |
      some result1 and some result2 and (
      exp in GEQ implies
        (AEval.aval[exp.lhs][curSt] >= AEval.aval[exp.rhs][curSt]
         implies BEval.bval[exp][curSt] = True else
         BEval.bval[exp][curSt] = False) else
      exp in EQ implies
        (AEval.aval[exp.lhs][curSt] = AEval.aval[exp.rhs][curSt]
         implies BEval.bval[exp][curSt] = True else
         BEval.bval[exp][curSt] = False) else
      exp in NEQ implies
        (AEval.aval[exp.lhs][curSt] != AEval.aval[exp.rhs][curSt]
         implies BEval.bval[exp][curSt] = True else
         BEval.bval[exp][curSt] = False) else
      exp in LEQ implies
        (AEval.aval[exp.lhs][curSt] <= AEval.aval[exp.rhs][curSt]
         implies BEval.bval[exp][curSt] = True else
         BEval.bval[exp][curSt] = False) else
      exp in GTH implies
        (AEval.aval[exp.lhs][curSt] > AEval.aval[exp.rhs][curSt]
        implies BEval.bval[exp][curSt] = True else
        BEval.bval[exp][curSt] = False)
      )
}
```

Figura 28: Alloy - Predicado de semântica sobre expressões booleanas

A semântica sobre expressões booleanas está definida para as assinaturas **`GEQ`**, **`EQ`**, **`NEQ`**, **`LEQ`** e **`GTH`**. Da mesma maneira que a semântica de expressões aritméticas, o predicado é formado por uma sequência de implicações que usam a semântica definida no Capítulo 5 para os respectivos tipos de expressões.

6.3.3 Semântica sobre execução de comandos

De maneira semelhante, a modelagem da semântica sobre os comandos da linguagem IMP é apresentada na Figura 29.

```
pred command_semantics[cmd: Cmd, curSt : State]{
  cmd -> curSt in CEval.cval.State implies
  let nextSt = CEval.cval[cmd][curSt] |
    cmd in Skip implies nextSt = curSt else
    cmd in Assign implies
      (let result = AEval.aval[cmd.rhs][curSt] |
       some result and
       nextSt.bind = curSt.bind ++ {cmd.lhs.name -> result}) else
    cmd in CondS implies
      (let condRes = BEval.bval[cmd.cond][curSt] |
       some condRes and
         (condRes = True implies
           (some CEval.cval[cmd.then][curSt] and
            nextSt =  CEval.cval[cmd.then][curSt]) else
           (some CEval.cval[cmd.elsen][curSt] and
            nextSt = CEval.cval[cmd.elsen][curSt]))) else
    cmd in SComp implies
      some CEval.cval[cmd.cur][curSt] and
      some CEval.cval[cmd.next][(CEval.cval[cmd.cur][curSt])] and
      nextSt = CEval.cval[cmd.next][(CEval.cval[cmd.cur][curSt])]
}
```

Figura 29: Alloy - Predicado de semântica sobre comandos

7 – Exemplos de síntese

7.1 Sintetizando constantes

O sintetizador criado nesse trabalho é capaz de sintetizar constantes desconhecidas em implementações parciais de programas. Tomando como exemplo o *sketch*, referenciado daqui em diante como "*double*", escrito em SKETCH apresentado na Figura 5, no Capítulo 2, que calcula o dobro de um número dado como entrada, o processo de síntese é apresentado a seguir:

Primeiramente, a especificação do comportamento do programa é definida através de exemplos ou de contratos. Em ambas as abordagens, são definidas as variáveis de entrada **X** e de saída **Res** para o programa, apresentadas no fragmento de código abaixo:

```
one sig X extends XLoc {}
one sig Res extends ALoc {}
```

Figura 30: Alloy - Especificação inicial do programa double

Uma possível especificação através de exemplos pode ser dada por uma função $double_x: Int \rightarrow Int$ definida, por exemplo, para os valores um e dois:

$$double_x(1) = 2$$
$$double_x(2) = 4$$

A modelagem dessa função para exemplos no sintetizador corresponde a dois estados iniciais, que especificam os valores de entrada; dois estados finais, que especificam as saídas esperadas; e um elemento do conjunto de exemplos, que mapeia os estados iniciais e finais entre si, como mostrado na Figura 31.

```
one sig iSt1 extends IState {} { bind = {X -> 1} + {Res -> 0}}
one sig fSt1 extends FState {} { bind = {X -> 1} + {Res -> 2}}
one sig iSt2 extends IState {} { bind = {X -> 2} + {Res -> 0}}
one sig fSt2 extends FState {} { bind = {X -> 2}  + {Res -> 4}}
one sig Ex1 extends Example{}{pairs = iSt1->fSt1 + iSt2->fSt2}
```

Figura 31: Alloy - Especificação através de exemplos para o programa double

Já uma possível especificação através de contratos para o programa pode ser dada por uma pré-condição vazia. No caso desse exemplo, devido as limitações de operações com inteiros em Alloy, também é restringido o escopo dos números usados no programa para o intervalo [-10,15]. Além disso, devido a natureza desse exemplo em particular, o número zero também é retirado do escopo com o objetivo de guiar o analisador a explorar estados que especifiquem melhor o problema durante o processo de síntese. A pós-condição, por sua vez, pode ser retirada diretamente do *assert* descrito no *sketch* mostrado originalmente na Figura 5. Esses predicados são definidos na Figura 32.

```
pred PreC[iSt: one IState] {
      integerScope[-10,15, {0}]
}
pred PosC[iSt: one IState, fSt: one FState] {
      fSt.bind[Res] = plus[iSt.bind[X], iSt.bind[X]]
}
```

Figura 32: Alloy - Especificação através de contrato para o programa double

Completada a especificação, seja através de exemplos ou através de contratos, o comando de síntese pode ser executado e um programa candidato será encontrado. Para este exemplo, no entanto, já é dado um programa parcial e o objetivo é apenas sintetizar um valor para a constante desconhecida. Dessa maneira, a modelagem do código do *sketch* mostrado na Figura 5 é apresentada na Figura 33 a seguir e um diagrama dessa modelagem é apresentado na Figura 35. O buraco ?? do *sketch* original é representado aqui como a assinatura **`UnknownConstant`**.

```
one sig VariableX extends IntVar {} {name = X}
one sig VariableRes extends IntVar {} {name = Res}
one sig UnknownConstant extends IntVal {}
one sig Mult1 extends Mult{} {op1 = VariableX and op2 = IntVal}
one sig Assign1 extends Assign {} {lhs = VariableRes and
      rhs = Mult1}
```

Figura 33: Alloy – Modelagem do programa parcial double

Finalmente, com a especificação do programa e a modelagem do *sketch* completadas, o comando de síntese é executado. Para executar o predicado de síntese, o escopo do modelo deve ser especificado, indicando quantos átomos de cada assinatura devem ser usados pelo analisador Alloy. A Figura 34 apresenta o comando e o escopo usados para esse exemplo. O escopo das assinaturas **`IntVar`**, **`IntVal`**, **`CondS`**, **`Skip`**, **`SComp`**, **`Assign`** e **`While`** é derivado diretamente da modelagem parcial do programa apresentada na Figura 33. O escopo da assinatura **`Int`** define o número de bits usados pelo analisador Alloy para representar os números inteiros. No caso desse exemplo, "**`6 Int`**" indica que seis bits serão usados para representar inteiros, resultando em no máximo 2^6 átomos dessa assinatura. Por último, o escopo geral "**`for 6`**" indica que no máximo seis átomos de todas as outras assinaturas serão usados. Para o escopo geral e para assinaturas onde uma implementação parcial não é fornecida, o limite do escopo atualmente é escolhido através de experimentos, onde o escopo do comando é aumentado em cada execução até que o analisador consiga encontrar uma instância do modelo.

A Figura 36 mostra o diagrama de uma instância do programa final sintetizado, com a constante desconhecida preenchida corretamente com o valor dois.

```
run syntByContract for 6 but 6 Int, exactly 2 IntVar,
    exactly 1 IntVal, exactly 0 CondS, exactly 0 Skip,
    exactly 0 SComp, exactly 1 Assign, exactly 0 While
```

Figura 34: Alloy - Comando de síntese para o programa double

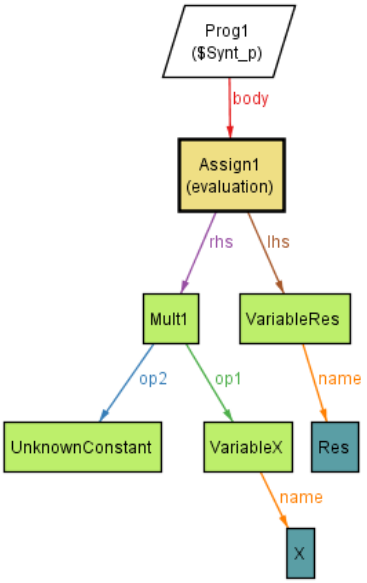

Figura 35 - Diagrama do modelo sketch do programa double

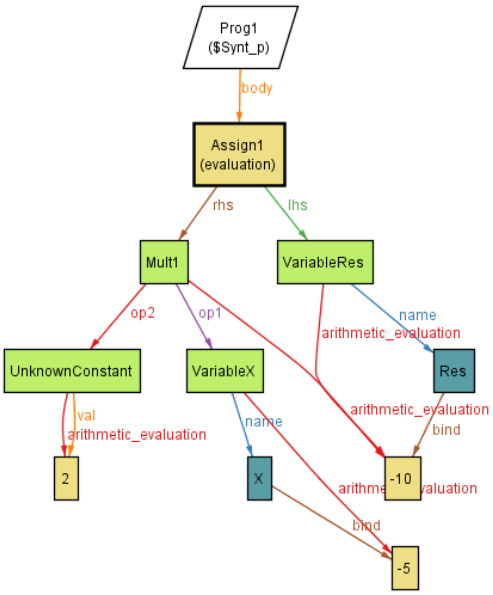

Figura 36 - Diagrama de uma instância do programa double

7.2 Sintetizando expressões

O sintetizador também é capaz de sintetizar expressões aritméticas ou booleanas para um programa parcialmente definido. É apresentado como exemplo o *sketch* da Figura 37. Nele é definido o programa "max2", que recebe dois inteiros *x* e y como entrada, calcula o máximo entre esses dois números e o armazena na variável *res*.

```
max2(x, y){
    if (??){
        res = x;
    } else {
        res = y;
    }
    assert ans >= x and ans >= y and (ans == x or ans == y)
}
```

Figura 37: Um sketch para o programa max2

Da mesma maneira do exemplo anterior, o processo de síntese começa pela especificação do comportamento esperado do programa. A especificação inicial das variáveis de entrada e saída do programa é mostrada na Figura 38.

```
one sig X, Y extends XLoc {}
one sig Res extends ALoc {}
```

Figura 38: Alloy - Especificação inicial para o programa max2

Novamente, uma possível especificação através de exemplos para esse programa pode ser dada por uma função $max_2: Int \rightarrow Int \rightarrow Int$ definida parcialmente para os valores um e dois:

$$max_2(1,2) = 2$$
$$max_2(2,1) = 2$$

A modelagem dessa função em exemplos no sintetizador pode ser encontrada no fragmento de código apresentado na Figura 39.

```
one sig iSt1 extends IState {} { bind = {X -> 1}+ {Y -> 2} +
    {Res -> 0}}
one sig fSt1 extends FState {} { bind = {X -> 1} + {Y -> 2}  +
    {Res -> 2}}
one sig iSt2 extends IState {} { bind = {X -> 2}+ {Y -> 1} +
    {Res -> 0}}
one sig fSt2 extends FState {} { bind = {X -> 2} + {Y -> 1}  +
    {Res -> 2}}
one sig Ex1 extends Example {} { pairs = {iSt1 -> fSt1} +
    {iSt2 -> fSt2}}
```

Figura 39: Alloy - Especificação através de exemplos para o programa max2

Já uma possível especificação através de contratos para esse programa pode ser retirada diretamente da asserção contida no *sketch* original apresentado na Figura 37. Além disso, podemos restringir, da mesma forma do exemplo anterior, o escopo dos números inteiros a serem tratados pelo programa gerado. A modelagem desse contrato pode ser encontrada na Figura 40.

```
pred PreC[iSt: one IState] {
    integerScope[0,4, none]
}
pred PosC[iSt: one IState, fSt: one FState] {
    fSt.bind[Res] >= iSt.bind[X] and
    fSt.bind[Res] >= iSt.bind[Y] and
    (fSt.bind[Res] = iSt.bind[X] or
        fSt.bind[Res] = iSt.bind[Y])
}
```

Figura 40: Alloy - Especificação através de contrato para o programa max2

A modelagem do programa parcial é mostrada na Figura 41.

```
one sig VariableX extends IntVar {} {name = X}
one sig VariableY extends IntVar {} {name = Y}
one sig VariableRes extends IntVar {} {name = Res}
one sig Assign1 extends Assign {} {lhs = VariableRes and
    rhs = VariableX}
one sig Assign2 extends Assign {} {lhs = VariableRes and
    rhs = VariableY}
one sig CondS1 extends CondS {} {then = Assign1 and
    elsen = Assign2}
one sig Prog1 extends Prog {} {body = CondS1}
```

Figura 41: Alloy – Modelagem do sketch para o programa max2

Por último, com uma das especificações definidas, o comando de síntese pode ser executado e um programa candidato será obtido. Um exemplo desse programa é apresentado no diagrama da Figura 43. O comando e os limites usados para a síntese desse exemplo são mostrados na Figura 42.

```
run syntByExample for 6 but 6 Int,  exactly 3 IntVar,
    exactly 0 IntVal, exactly 1 CondS, exactly 0 Skip,
    exactly 0 SComp, exactly 2 Assign, exactly 0 While,
    exactly 0 AExp
```

Figura 42: Alloy - Comando de síntese para o programa max2

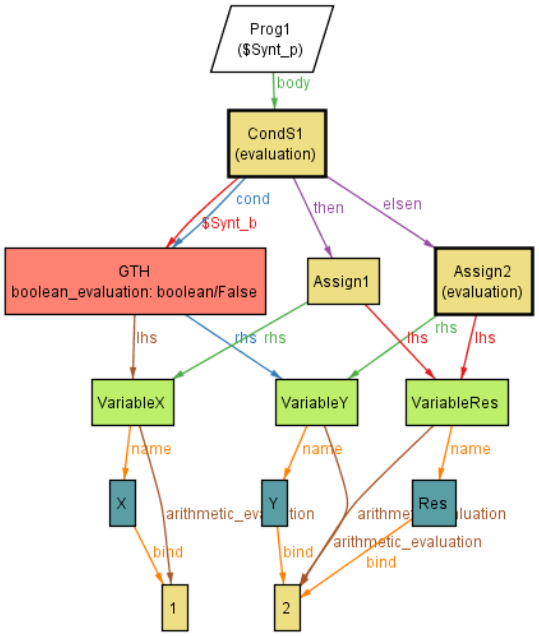

Figura 43: Diagrama do programa max2 sintetizado

7.3 Sintetizando programas

A última e mais abrangente capacidade do sintetizador é a de gerar programas completos, sem depender de um *sketch* parcial de código-fonte. Como um exemplo, é apresentado, nessa seção, o processo de síntese para o programa max3, que recebe três números *x*, *y e z* como entrada, calcula o máximo entre esses três números e o armazena numa variável *res.*

Uma possível especificação para esse programa é a função $\max_3 : Int \rightarrow Int \rightarrow Int$ definida parcialmente para os valores um, dois e três como a seguir:

$$\max_3(1,2,3) = 3 \qquad \max_3(2,3,1) = 3$$
$$\max_3(1,3,2) = 3 \qquad \max_3(3,1,2) = 3$$
$$\max_3(2,1,3) = 3 \qquad \max_3(3,1,3) = 3$$

Outra especificação pode ser dada através de um contrato com a seguinte pós condição:

$$res \geq x \ \wedge res \geq y \ \wedge res \geq z \ \wedge (res = x \ \vee res = y \vee res = z)$$

Ambas as especificações estão modeladas na Figura 44, Figura 45 e Figura 46.

```
one sig X, Y extends XLoc {}
one sig Res extends ALoc {}
```

Figura 44: Alloy - Especificação inicial do programa max3

```
one sig iSt1 extends IState {} { bind = {X -> 3}+ {Y -> 2} +
    {Z -> 1}+ {Res -> 0}}
one sig fSt1 extends FState {} { bind = {X -> 3} + {Y -> 2} +
    {Z -> 1} + {Res -> 3}}
one sig iSt2 extends IState {} { bind = {X -> 3}+ {Y -> 1} +
    {Z -> 2}+ {Res -> 0}}
one sig fSt2 extends FState {} { bind = {X -> 3} + {Y -> 1} +
    {Z -> 2} + {Res -> 3}}
one sig iSt3 extends IState {} { bind = {X -> 2}+ {Y -> 3} +
    {Z -> 1}+ {Res -> 0}}
one sig fSt3 extends FState {} { bind = {X -> 2} + {Y -> 3} +
    {Z -> 1}+ {Res -> 3}}
one sig iSt4 extends IState {} { bind = {X -> 1}+ {Y -> 2} +
    {Z -> 3}+ {Res -> 0}}
one sig fSt4 extends FState {} { bind = {X -> 1} + {Y -> 2} +
    {Z -> 3}+ {Res -> 3}}
one sig iSt5 extends IState {} { bind = {X -> 1}+ {Y -> 3} +
    {Z -> 2}+ {Res -> 0}}
one sig fSt5 extends FState {} { bind = {X -> 1} + {Y -> 3} +
    {Z -> 2}+ {Res -> 3}}
one sig iSt6 extends IState {} { bind = {X -> 2}+ {Y -> 1} +
    {Z -> 3}+ {Res -> 0}}
one sig fSt6 extends FState {} { bind = {X -> 2} + {Y -> 1} +
    {Z -> 3}+ {Res -> 3}}
one sig Ex1 extends Example {} { pairs = iSt1 -> fSt1 +
    iSt2 -> fSt2 + iSt3 -> fSt3 + iSt4 -> fSt4 + iSt5 -> fSt5 +
    iSt6 -> fSt6 }
```

Figura 45: Alloy - Especificação através de exemplos para o programa max3

```
pred PreC[iSt: one IState] {

}

pred PosC[iSt: one IState, fSt: one FState] {
    (fSt.bind[Res] = iSt.bind[X] or
        fSt.bind[Res] = iSt.bind[Y] or
        fSt.bind[Res] = iSt.bind[Z]) and
    fSt.bind[Res] >= iSt.bind[X] and
    fSt.bind[Res] >= iSt.bind[Y] and
    fSt.bind[Res] >= iSt.bind[Z]
}
```

Figura 46: Alloy - Especificação através de contrato para o programa max3

Finalmente, com uma das especificações completadas, o predicado de síntese como descrito na Figura 47 pode ser executado. A Figura 48 mostra o diagrama do programa gerado, enquanto a Figura 49 contém o pseudocódigo equivalente ao diagrama.

```
run syntByExample for 6 but 3 int, exactly 3 XLoc,
      exactly 1 ALoc, exactly 0 DLoc, exactly 3 CondS,
      exactly 0 SComp, exactly 3 Assign,
      exactly 0 GEQ, exactly 0 Skip, exactly 0 While,
      exactly 0 NEQ, exactly 4 IntVar, exactly 0 Mult,
      exactly 3 GTH, exactly 0 Add, exactly 0 Sub
```

Figura 47: Alloy - Comando de síntese para o programa max3

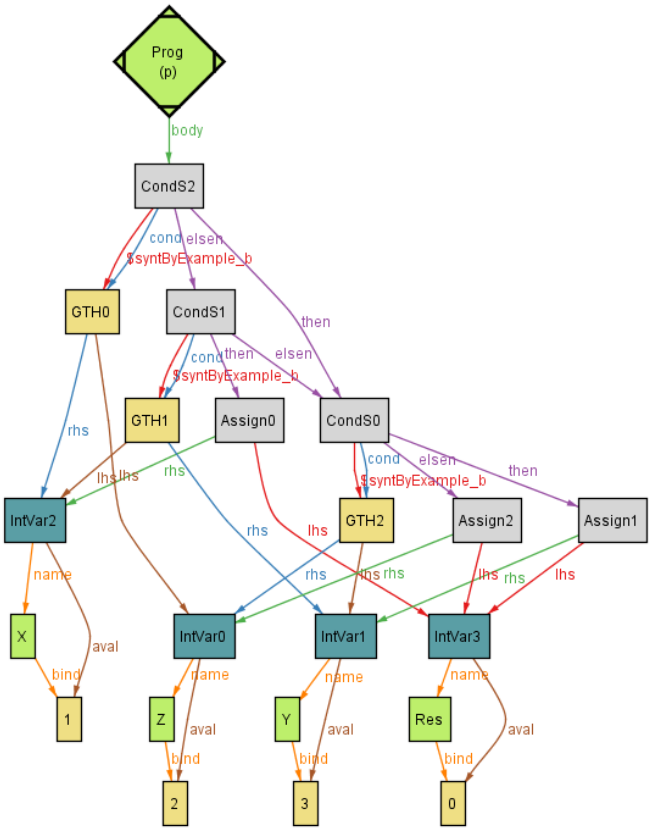

Figura 48: Diagrama do programa max3 sintetizado

```
max3(x,y,z){
   if z > x then
       if y > z then
           res = y
       else
           res = z
   else if x > y then
       res = x
   else if y > z then
       res = y
   else res = z
}
```

Figura 49: Pseudocódigo do programa max3 sintetizado

8 – Conclusão

Síntese de programas é uma técnica poderosa para melhoria do processo de desenvolvimento de software. Através dela, desenvolvedores conseguem verificar e gerar automaticamente partes de programas. Isso permite tanto que novos programas sejam criados de maneira semiautomática como também que algum trecho de código que contenha algum erro possa ser corrigido automaticamente baseado numa especificação do comportamento desejado. Além disso, usuários leigos conseguem usar síntese de programas através de exemplos para gerar algoritmos para suas tarefas sem necessariamente conhecer a linguagem de programação utilizada para executá-los.

Várias ferramentas estão disponíveis atualmente que possuem essas funcionalidades de síntese, como SKETCH, Rosette e o Microsoft PROSE. Nesse trabalho, essas aplicações foram analisadas e notou-se que as principais vantagens que muitas delas têm em comum é a capacidade de gerar programas completos, sem depender de uma implementação parcial, e a facilidade de especificação do comportamento desejado através de pares de entradas e saídas desejadas para o programa.

Além disso, esse trabalho apresenta um sintetizador criado na linguagem Alloy para gerar programas na linguagem IMP que tanto é capaz de gerar programas completos como suporta especificações através de exemplos e contratos. A sintaxe e semântica, tanto de Alloy quanto de IMP, foram apresentadas e usadas para modelar o sintetizador de programas. A principal contribuição desse sintetizador é um módulo genérico para síntese que pode ser reusado na criação de sintetizadores para outras linguagens de programação.

Por último, foram mostrados exemplos de síntese de constantes, expressões e programas utilizando o sintetizador criado. Esses exemplos incluem especificações tanto através de pares de entrada e saída como de contratos especificando propriedades sobre os estados iniciais e finais da execução. Os exemplos apresentados também servem para demonstrar um passo a passo de como utilizar o sintetizador para gerar novos programas.

8.1 Trabalhos futuros

Existem diversas oportunidades de continuação desse trabalho. De maneira independente, o módulo genérico de síntese pode ser usado para criar novos sintetizadores para linguagens atuais como Java, C ou Python através da modelagem da sintaxe e da semântica da linguagem desejada. Já de maneira diretamente relacionada ao trabalho, as seguintes propostas são oportunidades de melhoria para a implementação atual do sintetizador:

- Criação ou adoção de uma linguagem estruturada para especificações do comportamento desejado para programas sintetizados, permitindo assim que usuários não familiarizados com Alloy e até aplicações independentes de Alloy consigam utilizar o sintetizador. A ferramenta PROSE, por exemplo, é capaz de sintetizar programas utilizando exemplos de entrada e saída fornecidos em uma planilha no formato Excel;
- Criação de um módulo de tradução de instâncias de programas gerados pelo sintetizador para programas executáveis e de fácil compreensão para humanos, e vice-versa;

- Estudo para melhorar a definição do escopo a ser utilizado na execução dos predicados de síntese, diminuindo assim a necessidade de interação humana com o sintetizador;
- Estender e adicionar novas restrições e funcionalidades ao modelo do sintetizador criado com o objetivo de suportar outras estruturas sintáticas como Strings e arrays de valores;
- Estender o modelo e adicionar novas restrições para melhorar a velocidade de síntese e facilitar o processo de busca de modelos realizado pelo analisador Alloy durante execuções.

Referências

Basin, D., Deville, Y., Flener, P., Hamfelt, A., & Nilsson, J. F. (2004). Synthesis of Programs in Computational Logic. Em *Program Development in Computational Logic* (Vol. 3049, pp. 30-65). Berlin: Springer.

De Moura, L., & Bjørner, N. (2008). Z3: An efficient SMT solver. *Tools and Algorithms for the Construction and Analysis of Systems* (pp. 337-340). Budapest: Springer.

Deng, X., Lee, J., & Robby. (2006). Bogor/Kiasan: A k-bounded Symbolic Execution for. *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, (pp. 157-166). IEEE: Tokyo.

Dennis, G., Yessenov, K., & Jackson, D. (2008). Bounded Verification of Voting Software. *Proceedings of the 2nd international conference on Verified Software: Theories, Tools, Experiments* (pp. 130-145). Toronto: Springer.

Hoare, C. A. (1985). Communicating Sequential Processes. Em P. Brinch Hansen, *The Origin of Concurrent Programming* (pp. 413-443). New York: Springer.

ISO/IEC 13568. (2002). *ISO/IEC 13568:2002: Information technology -- Z formal specification notation -- Syntax, type system and semantics.* Geneva: ISO/IEC.

Jackson, D. (2002, Abril). Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, pp. 256-290.

Jackson, D. (2006). *Software Abstractions: Logic, Language, and Analysis.* Cambridge: The MIT Press.

Kwiatkowska, M., Norman, G., & Parker, D. (2011). PRISM 4.0: Verification of Probabilistic Real-time Systems. *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)* (pp. 585-591). Nova Iorque: Springer.

McCarthy, J. (1963). A Basis for a Mathematical Theory of Computation. *Studies in Logic and the Foundations of Mathematics*, Vol. 35, pp. 33-70.

Milicevic, A., Near, J. P., Kang, E., & Jackson, D. (2015). Alloy*: A General-Purpose Higher-Order Relational Constraint Solver. *IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE), 2015* . Florence: IEEE.

O'Neill, M., & Ryan, C. (2012). *Grammatical evolution: evolutionary automatic programming in an arbitrary language.* Springer.

Polozov, O., & Gulwani, S. (2015). FlashMeta: A Framework for Inductive Program Synthesis. *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (pp. 107-126). Pittsburgh: ACM.

Solar-Lezama, A. (2003). *Program Synthesis by Sketching.* Berkeley: UNIVERSITY OF CALIFORNIA.

Solar-Lezama, A., Tancau, L., Bodik, R., Seshia, S., & Saraswat, V. (2006). Combinatorial sketching for finite programs. *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, (pp. 404-415).

Srivastava, S., Gulwani, S., & Foster, J. S. (2009). VS3: SMT Solvers for Program Verification. *CAV: International Conference on Computer Aided Verification* (pp. 702-708). Grenoble: Springer.

Torlak, E., & Bodik, R. (2013). Growing solver-aided languages with Rosette. *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software* (pp. 135-152). Indianapolis: ACM.

Visser, W., Havelund, K., Brat, G., Park, S., & Lerda, F. (2003, Abril). Model Checking Programs. *Automated Software Engineering*, Vol. *10*, pp. 203-232.

Vladislavleva, E. J., Smits, G. F., & Hertog, D. D. (2008). Order of Nonlinearity as a Complexity Measure for Models Generated by Symbolic Regression via Pareto Genetic Programming. *IEEE Transactions on Evolutionary Computation*, pp. 333-349.

Winskel, G. (1993). *The Formal Semantics of Programming Languages.* London,: The MIT Press.