Especificações Alloy de Elementos da Lógica Proposicional

Filipe Oliveira[1,2], **Elthon Oliveira**[1]

[1] Universidade Federal de Alagoas (UFAL) - Campus Arapiraca
Avenida Manoel Severino Barbosa, s/n – Bom Sucesso, 57309-005 – Arapiraca – AL
[2]Bolsista PIBIC Edital 2020-2021 UFAL/CNPq/FAPEAL.

{filipe.oliveira,elthon}@arapiraca.ufal.br

Resumo. *A elaboração de problemas únicos e com características específicas em disciplinas formais de graduação é uma tarefa tediosa, principalmente quando a disciplina é ofertada em Massive Open Online Courses (MOOCs). Diante deste cenário, foi desenvolvida uma técnica à geração automática de fórmulas e argumentos válidos da Lógica Proposicional. O maior desafio da técnica foi gerar argumentos válidos. Pois, estes argumentos são improváveis de surgir a partir da geração totalmente aleatória. A técnica faz uso de conceitos da área de Síntese de Programas. Para isto, os conceitos de síntese são utilizados em conjunto com especificações formais descritas em Alloy. Neste artigo, são apresentadas as especificações Alloy necessárias à geração dos elementos da Lógica.*

Abstract. *The elaboration of unique problems with specific characteristics in formal undergraduate courses is a tedious task, especially when the course is offered in Massive Open Online Courses (MOOCs). In this scenario, a technique for the automatic generation of propositional logic formulas and valid arguments was developed. The major challenge of the technique was to generate valid arguments. These arguments are unlikely to arise from totally random generation. The technique makes use of concepts from the Program Synthesis area. For this, the synthesis concepts are used together with formal specifications described in Alloy. In this article, Alloy specifications necessary for the generation of Logic elements are presented.*

1. Introdução

Escrever manualmente novos problemas que tenham específicas características de solução, como um determinado nível de dificuldade ou que envolva o uso de um determinado conjunto de conceitos, é uma tarefa tediosa para o professor. A automatização deste processo traz alguns benefícios.

Uma ferramenta de geração de problemas pode fornecer aos professores uma fonte de problemas a serem usados em suas tarefas ou anotações de aula. A geração de problemas pode ajudar também a evitar fraudes em salas de aula ou MOOCs (*Massive Open Online Courses*) [Mozgovoy et al. 2010]. Além disso, o nível de dificuldade do problema gerado e apresentado ao aluno pode ser controlado de acordo com o seu desempenho, podendo ajudar a criar fluxos de trabalho personalizados para os alunos. Por exemplo, se um estudante resolve um problema corretamente, pode ser apresentado a ele um problema mais difícil do que o primeiro, ou que envolva outros conceitos.

Porém, no contexto da disciplina de Lógica, é preciso evitar ou tratar alguns problemas na geração de fórmulas e argumentos: ambiguidades na especificação do professor, questões com diferentes níveis de dificuldade erroneamente postas dentro de um mesmo conjunto, argumentos desconhecidamente inválidos, dentre outros. Esta geração de fórmulas/argumentos pode ser feita utilizando conceitos de *Síntese de Programas* [Gulwani et al. 2017].

Síntese de Programas é a tarefa de encontrar automaticamente um programa, numa determinada linguagem de programação, a partir da intenção do usuário. Esta intenção deve estar descrita na forma de alguma especificação. Alguns pesquisadores consideram a síntese de programas como um dos problemas centrais da teoria da programação [Pnueli and Rosner 1989]. Para possibilitar a geração de fórmulas e argumentos válidos, é utilizada e adaptada a abordagem de *Geração de Esboço*, da área de Síntese, junto com a programação por restrições. Assim, neste trabalho, são apresentadas as especificações formais, descritas em linguagem Alloy [Jackson 2012], necessárias à correta e adequada síntese dos elementos lógicos.

O restante deste artigo está organizado da seguinte forma. Alguns trabalhos relacionados são apresentados na Seção 2. A base teórica ao entendimento deste artigo é apresentada na Seção 3. As especificações desenvolvidas são detalhadas e um exemplo é ilustrado na Seção 4. Por fim, na Seção 5 são apresentadas as conclusões e o direcionamento aos trabalhos futuros.

2. Trabalhos Relacionados

A abordagem de *Geração por Esboço* possui diversas aplicações na Computação, principalmente na *Síntese de Programas*. Dentre os trabalhos que usam tal abordagem, pode-se citar o [Wang et al. 2019], onde os autores propõem uma abordagem para sintetizar automaticamente uma nova versão de um programa de banco de dados, dada sua versão original e os esquemas de origem e destino.

Já no trabalho [Yaghmazadeh et al. 2017], é apresentada a síntese de consultas SQL a partir da linguagem natural. É obtido o esboço do programa a partir de técnicas de análise padrão. Em outro trabalho [Nori et al. 2015], é apresentada a síntese de programas probabilísticos a partir de conjuntos de dados reais. Os autores utilizam geração por esboço para especificar os esqueletos do programas contendo *buracos* que são preenchidos por instâncias geradas usando Cadeia de Markov de Monte Carlo.

O trabalho apresentado em [Ahmed et al. 2013] assemelha-se ao trabalho aqui apresentado no que diz respeito à síntese alvo: argumentos da Lógica Proposicional. Os autores usam o chamado *Grafo Universal de provas (UPG)*, estrutura de dados chave do trabalho, que codifica as aplicações das regras de inferência sobre as pequenas proposições abstraídas usando sua representação de tabela de verdade baseada em um vetor de bits. O trabalho é capaz de gerar soluções e problemas com características de solução em comum. Para isso, utilizam algoritmos de busca na estrutura *UPG*.

Neste último trabalho, operações no vetor de bits garantem a síntese de argumentos válidos e o descarte de inválidos. O diferencial do trabalho aqui apresentado é que, graças às especificações Alloy e ao uso de seu analisador na geração de modelos de problemas, a validade dos argumentos é garantida por construção. Além disso, os modelos gerados pelo analisador Alloy contém os problemas gerados e as respectivas soluções.

3. Fundamentação teórica

3.1. Síntese de Programas

Síntese de Programas é a tarefa de encontrar um programa automaticamente através de uma linguagem de programação subjacente e que satisfaça uma determinada especificação [Gulwani et al. 2017]. Esta tarefa executa buscas sobre um espaço de programas para gerar um programa que seja consistente com uma determinada variedade de restrições. Os dois principais desafios desta área são: *espaço de programas* - o número de programas candidatos cresce exponencialmente com o tamanho do programa; e *intenção do usuário* - muitos domínios de aplicações são bem complexos para serem descritos completamente por especificações formais ou informais.

Há diversas técnicas envolvidas na sintetização de programas, a saber: geração de esboço, pesquisa enumerativa, pesquisa estocástica e programação por exemplos. Neste trabalho utiliza-se uma abordagem baseada na *Geração de Esboço*. Esta técnica permite que os usuários expressem atividades de alto nível de um problema escrevendo um esboço. O esboço é um programa parcial que codifica a estrutura de uma solução e deixa seus detalhes (lacunas) sem especificação [Solar-Lezama 2009].

3.2. Linguagem Proposicional

A linguagem proposicional L_{LP} é o conjunto das chamadas *fórmulas bem formadas - fbf's*. De acordo com [Silva et al. 2006], L_{LP} é definida indutivamente sendo o menor conjunto satisfazendo às seguintes regras de formação:

1. Todas as variáveis proposicionais estão em L_{LP} e são chamadas de fórmulas atômicas ou átomos.
2. Se $\alpha \in L_{LP}$, então $\neg\alpha \in L_{LP}$.
3. Se $\alpha, \beta \in L_{LP}$, então $\alpha \wedge \beta \in L_{LP}$, $\alpha \vee \beta \in L_{LP}$, $\alpha \rightarrow \beta \in L_{LP}$ e $\alpha \leftrightarrow \beta \in L_{LP}$.

Além das *fbf's*, há também o conceito de argumento da *Lógica Proposicional*. No contexto deste trabalho, um argumento é definido como $\Delta \vdash \delta$, onde Δ e δ são subconjunto não vazio de L_{LP} (premissas) e elemento de L_{LP} (conclusão), respectivamente. Um argumento é dito válido quando a conclusão decorre do conjunto de premissas e é dito inválido, caso contrário. Para demonstrar a validade de um argumento é apresentada uma sequência de aplicações de regras de inferência denominada de prova. Tais regras são omitidas neste trabalho devido às restrições de espaço.

3.3. Alloy

Alloy é uma linguagem declarativa de especificação de modelos, na qual é possível utilizar formalismos lógicos para produzir restrições para estruturar um determinado problema [Jackson 2012]. Além da linguagem, há também um analisador adjacente utilizado na criação de modelos de sistemas de software, muito empregados pela Engenharia de Software. Para isso, Alloy utiliza conceitos para produzir restrições para especificar corretamente o comportamento de um determinado sistema. Dentre esses conceitos, destacam-se teoria de conjuntos, orientação a objetos e lógica de primeira ordem.

A linguagem Alloy possui diversos recursos utilizados para definir especificações. Os mais básicos são: **assinatura** - palavra reservada *sig* (*signature*): usada para definir entidades que compõem o sistema especificado; **fato** - palavra reservada *fact*: usado para

definir restrições explícitas aplicadas aos relacionamentos das entidades; e **predicado** - palavra reservada *pred*: usado para definir expressões que podem ser usadas na definição de fatos ou na verificação de propriedades.

4. Especificações Alloy

Combinando a técnica de programação por restrições usando a linguagem Alloy com a técnica de *Geração por Esboço* é possível gerar fórmulas e argumentos válidos da Lógica Proposicional. Há uma solução de software desenvolvida composta por um aplicativo móvel e uma API (*Application Programming Interface*) disponível no formato de serviço web. De forma resumida, o processo de síntese desenvolvido consiste em três etapas, a saber: (i) intenção do usuário sobre as características da fórmula (ou argumento) fornecida por meio do aplicativo móvel; (ii) parametrização a partir das especificações Alloy; e (iii) geração dos modelos usando API do analisador Alloy.

Nesta seção são apresentadas as especificações Alloy para fórmula-bem-formada e para argumento válido. A validade dos argumentos gerados é garantida por construção.

A especificação referente à *fbf* pode ser vista no Código 1. Na linha 1 é definida a entidade abstrata $Formula$ que pode ser $Unaria$ ou $Binaria$ (linhas 2 e 3). Nas linhas 4 a 6 são definidas as entidades que podem ser instanciadas: átomo, negação, conjunção, disjunção, implicação e biimplicação.

```
1 abstract sig Formula{}
2 abstract sig Unary extends Formula{ child: Formula }
3 abstract sig Binary extends Formula{ left, right: Formula }
4 sig Atom extends Formula {}
5 sig Not extends Unary{}
6 sig And, Or, Imply, BiImply extends Binary {  }
7 one sig FBF{ mainOperator: one Formula }
```

Código 1: Entidades básicas para *fbf's*.

O analisador Alloy busca modelos que satisfaçam as especificações. A linha 7 indica ao analisador que, no processo de busca, ele encontre modelos que tenham apenas uma instância de FBF, que por sua vez está relacionada a apenas um objeto do tipo $Formula$. A instância única FBF é a raiz da fórmula. Contudo, apenas com esta especificação, o analisador poderá encontrar modelos no quais haverá fórmulas desconexas entre si ou fórmulas binárias onde alguma parte seja a própria fórmula, uma autorreferência produzindo um *loop* infinito.

No Código 2, a linha 1 apresenta um fato evitando que fórmulas possuam ciclos na árvore gerada pelo analisador Alloy, o que impede o *loop* infinito. Já o fato da linha 2 define que todo objeto $Formula$ deve estar na mesma árvore gerada a partir do único objeto FBF, evitando assim objetos $Formula$ sem ligação.

```
1 fact NoCycle{no n,n':Formula | n in n'.^(child+left+right) and n' in n.^(child+left+
    ↪ right)}
2 fact EveryNodeAtAFBF{all n:Formula | one t:FBF | n in t.mainOperator.*(child+left+
    ↪ right)}
3 pred Config(){ #And>0  #Or>0  #Not>0  #Imply=0  #BiImply=0  (#Atom≥3 ∧ #Atom≤6)}
```

Código 2: Fatos sobre *fbf's* e predicado parametrizável.

Na linha 3 é apresentada uma configuração para a síntese de fórmulas. Enquanto a especificação já apresentada se mantém a mesma sempre, esta parte é gerada a partir da

customização feita pelo usuário por meio do aplicativo móvel desenvolvido. Neste caso, o usuário definiu que devem ser geradas fórmulas com ao menos uma conjunção, ao menos uma disjunção, ao menos uma negação, nenhuma implicação e nenhuma biimplicação. Definiu também que todas as *fbf's* devem ter entre três e seis átomos distintos.

No Código 3 é apresentada a especificação referente às estruturas das regras de inferência, sendo a linha 1 a entidade abstrata $Rule$. Cada assinatura corresponde a uma regra, a saber: NE e NI - exclusão e inclusão da negação; CE e CI - exclusão e inclusão da conjunção; DE e DI - exclusão e inclusão da disjunção; BE e BI - exclusão e inclusão da biimplicação; MP - *modus ponens*; MT - *modus tollens*; e SD - silogismo disjuntivo.

```
1  abstract sig Rule { }
2  sig NE extends Rule {p1:Not, r:Formula}
3  sig NI extends Rule {p1:Formula, r:Not}
4  sig CI extends Rule {p1:Formula, p2:Formula, r:And}
5  sig CE extends Rule {p1:And, r:Formula}
6  sig DI extends Rule {p1:Formula, r:Or}
7  sig DE extends Rule {p1:Imply, p2:Imply, p3:Or, r:Formula}
8  sig BI extends Rule {p1:Imply, p2:Imply, r:BiImply}
9  sig BE extends Rule {p1:BiImply, r:Imply}
10 sig MP extends Rule {p1:Formula, p2:Imply, r:Formula}
11 sig MT extends Rule {p1:Formula, p2:Imply, r:Formula}
12 sig SD extends Rule {p1:Formula, p2:Or, r:Formula}
```

Código 3: Estruturas das regras de inferência.

Para exemplificar a uma das regras, tem-se na linha 4 a inclusão da conjunção. A estrutura desta regra é composta de três partes: duas *fbf's* quaisquer e uma *fbf* resultante que deve ser uma conjunção. Contudo, ainda não está especificado que esta última deva ser uma conjunção entre as duas primeiras.

O Código 4 apresenta os fatos que definem como cada regra de inferência funciona. Na linha 4, para a inclusão da conjunção, é definido que para qualquer instância CI, seu resultado (uma instância de And) deve ter as partes esquerda e direita iguais as primeiras partes de CI, independentemente da ordem.

```
1  fact rules{
2     all ne:NE | ne.p1.child.child=ne.r
3     all ni:NI | ni.p1=ni.r.child.child
4     all ci:CI | (ci.r.left=ci.p1 and ci.r.right=ci.p2) or (ci.r.left=ci.p2 and ci.r.
          ↪ right=ci.p1)
5     all ce:CE | ce.r = ce.p1.left or ce.r = ce.p1.right
6     all di:DI | di.p1 in di.r.(right+left)
7     all de:DE | ((de.p1.left=de.p3.left and de.p2.left=de.p3.right) or (de.p1.left=de.
          ↪ p3.right and de.p2.left=de.p3.left))
8        and de.p1.right=de.p2.right and de.r=de.p2.right
9     all bi:BI | bi.p1.right=bi.p2.left and bi.p2.right=bi.p1.left
10       and ((bi.r.right=bi.p2.right and bi.r.left=bi.p2.left) or (bi.r.right=bi.p2.
             ↪ left and bi.r.left=bi.p2.right))
11    all be:BE | (be.r.left=be.p1.left and be.r.right=be.p1.right) or (be.r.left=be.p1.
          ↪ right and be.r.right=be.p1.left)
12    all mp:MP | mp.p1 = mp.p2.left and mp.r = mp.p2.right
13    all mt:MT | (mt.p1.child = mt.p2.right and mt.r.child = mt.p2.left) or (mt.p1 = mt
          ↪ .p2.right.child and mt.r = mt.p2.left.child)
14    all sd:SD | (sd.p1.child = sd.p2.left and sd.r = sd.p2.right) or (sd.p1.child = sd
          ↪ .p2.right and sd.r = sd.p2.left)
15 }
```

Código 4: Funcionamento/Comportamento das regras de inferência.

No Código 5 são apresentados fatos considerados importantes à geração de argumentos. Nas linhas 1 a 3 são definidos predicados usados nestes fatos, a saber: linha

1 - uma fórmula não é igual a outra; linha 2 - lado direito é diferente do lado esquerdo da fórmula; e linha 3 - o lado direito da fórmula não é a negação do lado esquerdo, e vice-versa.

```
1  pred isNotEqualTo[a:Formula,a':Formula]{ (a.right≠a'.right or a.left≠a'.left) and (a.
       ↪ right≠a'.left or a.left≠a'.right)}
2  pred avoidA_A[a:Formula]{ a.right≠a.left }
3  pred avoidA_noA[a:Formula]{ (a.right.child≠a.left) and (a.right≠a.left.child) }
4  fact {
5     all a,a':Not | a.child=a'.child implies a=a'
6     all a,a':And | a.isNotEqualTo[a']
7     all a,a':Or | a.isNotEqualTo[a']
8     all a,a':BiImply | a.isNotEqualTo[a']
9     all a,a':Imply | (a.right=a'.right and a.left=a'.left) implies a=a'
10    all x:And | x.avoidA_A
11    all x:Or | x.avoidA_A
12    all x:Imply | x.avoidA_A
13    all x:BiImply | x.avoidA_A
14    all x:And | x.avoidA_noA
15    all x:Or | x.avoidA_noA
16    all x:Imply | x.avoidA_noA
17    all x:BiImply | x.avoidA_noA
18 }
```

Código 5: Fatos relevantes à geração de argumentos.

Os fatos apresentados no Código 5 evitam que duas ou mais instâncias de uma implicação gerem a mesma fórmula, por exemplo. Isto torna a geração de fórmulas mais eficiente e mais eficaz. A implicação poderá ocorrer mais de uma vez entre premissas e conclusão, mas terá a mesma instância no modelo encontrado pelo analisador Alloy, só que com mais de uma relação.

As especificações apresentadas no Código 6 são divididas em cinco partes. Entre as linhas 1 e 13 são apresentados os fatos que evitam que a mesma aplicação de uma regra ocorra mais de uma vez. A exclusão da negação pode ocorrer mais de uma vez, mas não gerando o mesmo resultado, por exemplo. Nas linhas 14 a 16 são definidas variáveis para simplificar a especificação do fato a seguir.

```
1  fact {
2     all a,a':NE | (a.r=a'.r) implies a=a'
3     all a,a':NI | (a.r=a'.r) implies a=a'
4     all a,a':CE | (a.p1=a'.p1 and a.r=a'.r) implies a=a'
5     all a,a':CI | (a.r=a'.r) implies a=a'
6     all a,a':DI | (a.p1=a'.p1 and a.r=a'.r) implies a=a'
7     all a,a':DE | ((a.p1.isNotEqualTo[a'.p1] and a.p2.isNotEqualTo[a'.p2]) or (a.p1.
          ↪ isNotEqualTo[a'.p2] and a.p2.isNotEqualTo[a'.p1]))
8     and a.p3.isNotEqualTo[a'.p3] implies a=a'
9     all a,a':BE | (a.p1=a'.p1 and a.r=a'.r) implies a=a'
10    all a,a':SD | (a.p1=a'.p1 and a.p2=a'.p2) implies a=a'
11    all a,a':MP | (a.p1=a'.p1 and a.p2=a'.p2) implies a=a'
12    all a,a':MT | (a.p1=a'.p1 and a.p2=a'.p2) implies a=a'
13 }
14 let P1 = NE<:p1+NI<:p1+CI<:p1+CE<:p1+DI<:p1+DE<:p1+BI<:p1+BE<:p1+MP<:p1+MT<:p1+SD<:p1
15 let P2 = CI<:p2+DE<:p2+BI<:p2+MP<:p2+MT<:p2+SD<:p2
16 let R = NE<:r+NI<:r+CI<:r+CE<:r+DI<:r+DE<:r+BI<:r+BE<:r+MP<:r+MT<:r+SD<:r
17 fact OneOrigin{
18    one rule: Rule | all f: Formula | f in rule.(P1+P2+p3+R).*(child +Binary<:left +
          ↪ Binary<:right) or f=rule.P1 or f=rule.P2 or f=rule.p3 or f=rule.R
19 }
20 one sig Argument{ premisse: set Formula, conclusion: one Formula }{
21    #premisse=3 and not (conclusion in premisse)
22 }
23 run Config for 4
```

Código 6: Restrições para a geração de argumentos e a assinatura de argumento.

Na linhas 17 a 19 é especificado o fato de que existe uma única regra a qual é aplicada às fórmulas (premissas) e/ou a partir da qual as demais fórmulas (premissas não básicas) podem ser alcançadas, mesmo que com aplicações de outras regras em passos intermediários. Este fato fornece duas garantias. Primeiramente, haverá uma origem única no processo de prova. E em segundo lugar, principalmente, que a conclusão possa ser inferida a partir do conjunto de premissas, ou seja, que o argumento seja válido.

A linha 20 define que deve existir apenas um argumento com premissas e conclusão. Já na linha 21 é definido que há três premissas e nenhuma delas deve ser a conclusão. Por fim, na linha 23 é definida uma configuração vazia (predicado $Config$) sem mais restrições, indicando ao analisador que gere até quatro instâncias de cada entidade.

Exemplo: Na Figura 1 são ilustrados dois exemplos de geração. Em 1 são apresentados os passos à geração de *fbf's*, contendo apenas conjunção, disjunção, negação e de três a seis átomos diferentes. Em 2 são apresentados os passos à geração de argumentos que necessitem ser resolvidos com ao menos uma aplicação de *modus ponens*, não haja aplicação de qualquer outra regra e cujas *fbf's* possuam até dois átomos diferentes.

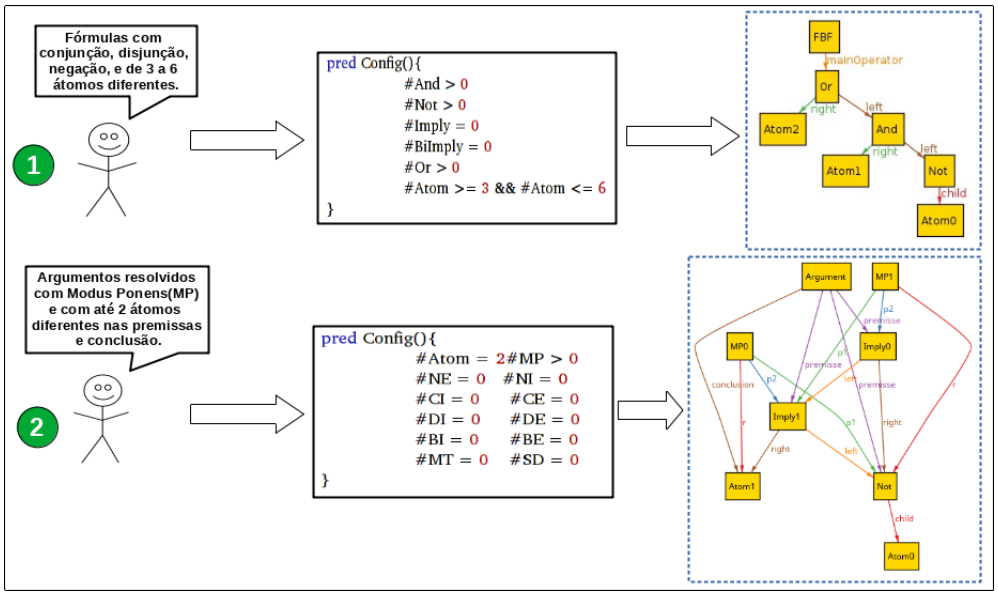

Figura 1. Exemplo de objetos gerados com as especificações.

5. Conclusões e trabalhos e futuros

Neste artigo foram apresentadas as especificações em Alloy necessárias à geração de elementos da *Lógica Proposicional*: fórmulas-bem formadas e argumentos. Considerando a improbabilidade na geração totalmente aleatória de argumentos válidos, estas especificações são um diferencial no processo, pois garantem a validade dos argumentos por construção. Desta forma, sem a necessidade de verificação após a síntese.

Estas especificações e o uso do analisador Alloy estão organizadas numa API na forma de serviço web que se comunica com um aplicativo móvel já desenvolvido. Este aplicativo funciona como ferramenta/interface para que o usuário parametrize a geração de fórmulas e argumentos, obtendo assim elementos da Lógica com características em

comum: complexidade, conceitos envolvidos e estilos de resposta. Esta solução elimina a tarefa tediosa de docentes na concepção de problemas únicos, principalmente quando a disciplina é ofertada em MOOCs.

Entretanto, as especificações apresentadas possuem algumas limitações. Para algumas configurações, o analisador Alloy chega a levar minutos para encontrar e devolver os modelos para que a API desenvolvida possa preparar e devolver as fórmulas e argumentos ao usuário. A solução para este problema já está em andamento. Como trabalho futuro, outro avanço desejado é o suporte às regras hipotéticas em argumentos (*prova do condicional* e *redução ao absurdo*). Também serão desenvolvidos os módulos, no aplicativo e na API, responsáveis por extrair e apresentar dicas de solução ao usuário.

Por fim, tanto a API desenvolvida quanto a ferramenta de software móvel apresentam *bugs*, dos quais alguns já estão sendo tratados enquanto outros ainda precisam ser melhor analisados.

Referências

Ahmed, U. Z., Gulwani, S., and Karkare, A. (2013). Automatically generating problems and solutions for natural deduction. In *IJCAI*, pages 1968–1975. Citeseer.

Gulwani, S., Polozov, A., and Singh, R. (2017). *Program Synthesis*, volume 4. NOW.

Jackson, D. (2012). *Software Abstractions: logic, language, and analysis*. MIT press.

Mozgovoy, M., Kakkonen, T., and Cosma, G. (2010). Automatic student plagiarism detection: future perspectives. *Journal of Educational Computing Research*, 43(4):511–531.

Nori, A. V., Ozair, S., Rajamani, S. K., and Vijaykeerthy, D. (2015). Efficient synthesis of probabilistic programs. *ACM SIGPLAN Notices*, 50(6):208–217.

Pnueli, A. and Rosner, R. (1989). On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 179–190.

Silva, F. S. C. d., Finger, M., and Melo, A. C. V. d. (2006). *Lógica para computação*. Cengage Learning.

Solar-Lezama, A. (2009). The sketching approach to program synthesis. In Hu, Z., editor, *Programming Languages and Systems*, pages 4–13, Berlin, Heidelberg. Springer Berlin Heidelberg.

Wang, Y., Dong, J., Shah, R., and Dillig, I. (2019). Synthesizing database programs for schema refactoring. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 286–300.

Yaghmazadeh, N., Wang, Y., Dillig, I., and Dillig, T. (2017). Sqlizer: query synthesis from natural language. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–26.