Geração automática de casos de teste para *auto-graders* baseada em execução simbólica

Loham Santos da Silva[1], **Reinaldo Silva Fortes**[1], **Rodrigo Geraldo Ribeiro**[1]

[1]Departamento de Computação (DECOM) - Universidade Federal de Ouro Preto (UFOP)
Caixa Postal 140 – 35400-000 - Ouro Preto/MG.

loham.silva@aluno.ufop.edu.br, {rodrigo.ribeiro, reifortes}@ufop.edu.br

Abstract. *This article presents a symbolic execution-based approach to automatically generate test cases for automated assessment in an introductory programming course. The tool focuses on topics covered in Brazilian universities. The article details the tool's inner workings and the entire process of generating automated test cases. The proposed case study revealed that, through the tool, it was possible to generate test cases in an automated way, exploring all execution paths of the teacher's solution, which can reduce the teacher's burden, ensuring the completeness of the tests.*

Resumo. *Este artigo apresenta uma abordagem de geração automática de casos de teste para ferramentas de correção de exercícios práticos de programação introdutória, utilizando Execução Simbólica, contemplando os principais tópicos abordados em universidades brasileiras. O artigo detalha o funcionamento da ferramenta construída, bem como todo o processo de geração automática de casos de teste. O estudo de caso proposto revelou que, através da ferramenta, foi possível gerar os casos de teste, de forma automatizada, explorando todos os caminhos de execução da solução algorítmica do professor, o que pode diminuir a sobrecarga dos docentes que utilizam essas ferramentas, garantindo a completude dos testes.*

1. Introdução

Em disciplinas introdutórias de programação de computadores, geralmente, há um elevado número de matrículas e o processo de aprendizado deve envolver a resolução de uma grande quantidade de exercícios práticos. A necessidade de preparação, disponibilidade e retorno sobre a solução de todos os exercícios, pode sobrecarregar docentes que ministram essas disciplinas, retardando ou limitando o retorno aos alunos.

Nesse sentido, diversos pesquisadores propõem utilizar ferramentas para correção automática de exercícios de programação introdutória, ou *auto-graders* [Ala-Mutka 2005]. Tais ferramentas auxiliam docentes na avaliação, classificação e gerenciamento de exercícios de programação de computadores. Algumas das características compartilhadas pela maioria das ferramentas de correção automática encontradas na literatura é a necessidade de inserção manual de casos de teste por parte dos professores e a necessidade de suporte a diferentes linguagens de programação [Ala-Mutka 2005].

Com respeito à necessidade de inserção de casos de testes, tais ferramentas demandam o cadastro de possíveis valores de entrada de dados e os respectivos resultados esperados, para que o exercício seja corrigido. Um grande inconveniente dessa abordagem

DOI: 10.5753/sbie.2024.242433

é a necessidade de inserir casos de teste para cada um dos exercícios propostos. Como exemplo, se um docente deseja incluir 5 casos de teste para cada exercício proposto, em um curso que propõe 50 exercícios, deverão ser cadastrados 250 casos de teste, o que pode gerar uma sobrecarga de trabalho por parte do professor. Outro fator relevante, é que apenas 5 casos de teste podem não ser suficientes para determinar se todos os caminhos de execução serão devidamente considerados pelo processo de correção.

No contexto brasileiro, dentre as ferramentas de correção automática mais citadas na literatura (veja Seção 3), não há a utilização de geração de casos de testes automatizados. Com base no descrito, o presente trabalho apresenta uma abordagem de geração automática de casos de teste, baseada em execução simbólica, implementada em uma ferramenta para correção automática de exercícios de programação introdutória denominada **Symb**. Com a referida abordagem, há a exploração de todos os caminhos de execução do gabarito por meio de valores simbólicos gerados automaticamente, substituindo a geração de casos de testes manuais. Vale ressaltar que, a dimensão dos programas implementados em disciplinas de programação introdutória são pequenos, e podem não configurar o problema de explosão de caminho, melhor descrito na Seção 2.

Outro problema relevante com os atuais *auto-graders* é que eles consideram um conjunto limitado de linguagens de programação e/ou não são facilmente extensíveis para novas linguagens. Neste sentido, o presente trabalho propõe que todo o processo de geração de testes e correção seja feito considerando a linguagem da própria ferramenta, denominada Symb, proposta neste trabalho, que contém recursos presentes na maioria dos cursos de programação introdutória de diversas universidades brasileiras. O suporte a novas linguagens se dá por *plug-ins* que são responsáveis por traduzir a nova linguagem para a linguagem Symb.

Mais especificamente, as principais contribuições deste trabalho são:

(a) Geração de casos de teste automatizados, com base na estrutura algorítmica do gabarito do professor. Pelo fato de o processo de geração tomar por base a técnica de execução simbólica, os testes gerados tentam garantir a cobertura do código do gabarito, através da exploração dos seus diferentes caminhos de execução. Além disso, o processo de geração de casos de teste automatizados da referida abordagem, pode ser utilizado em quaisquer *auto-graders*, pelo fato de somente utilizar o arquivo de gabarito do professor;
(b) A criação de uma linguagem de programação que abrange os principais conteúdos das disciplinas de programação introdutória de diversas universidades brasileiras, e que permite, através dela, que qualquer linguagem de programação utilizada para o gabarito e para a solução dos alunos possa ser utilizada para a geração dos casos de teste, desde que utilize as estruturas da linguagem proposta;
(c) A implementação da abordagem proposta em um protótipo, denominado **Symb**, para correção automática de trabalhos e a realização de testes baseados em uma de lista de exercícios contendo 36 exercícios de uma universidade brasileira. Como parte desse teste, foram construídos o gabarito, soluções corretas e incorretas para cada um destes exercícios. Essa bateria de testes foi submetida à ferramenta que identificou todas as soluções inválidas corretamente.

O código do algoritmo para geração de casos de teste, a implementação do protótipo **Symb** e detalhes para reprodução do estudo de caso reportado neste trabalho

podem ser encontrados no seguinte repositório on-line:

`https://github.com/lohamsilva-ufop/Symb-Autograder`

O restante deste trabalho está estruturado da seguinte forma: A seção 2 aborda os conceitos básicos de execução simbólica utilizados neste trabalho. A seção 3 descreve os trabalhos que possuem relação com a proposta. A seção 4 descreve a arquitetura do **Symb** e a geração automática de casos de teste. A seção 5 descreve o estudo de caso realizado com o objetivo de verificar a eficácia do processo de geração na ferramenta implementada, e por fim, a seção 6 descreve a conclusão deste trabalho.

2. Execução Simbólica

Execução simbólica é uma técnica de análise de programas utilizada para verificar se certas propriedades podem ou não ser violadas [King 1976]. Através dela, pode-se explorar múltiplos fluxos de execução de um programa, o que pode auxiliar na verificação de propriedades a serem testadas [Boyer et al. 1975, Baldoni et al. 2018].

Para isso, ao invés de utilizar entradas concretas, a execução simbólica utiliza entradas simbólicas e mantém uma condição de caminho, atualizada sempre que uma instrução de desvio é executada. Isso faz com que algum ponto específico do programa seja atingido e que diversos caminhos de um programa possam ser explorados, ao invés de um único fluxo de execução [Ábrahám 2015]. O objetivo primordial é analisar o programa, para determinar quais conjuntos de entradas fazem com que cada ponto específico do programa seja alcançado pelo menos uma vez.

Através da execução simbólica, são selecionados todos os caminhos de execução de um programa, para que cada caminho selecionado possa ser executado. Ao tentar executar um caminho, verifica-se qual é a sua restrição e um resolvedor SMT (*Satisfiability Modulo Theories*) é utilizado para verificar se essa restrição pode ser satisfeita, encontrando valores adequados, que a satisfaçam [Anand et al. 2009]. Denomina-se restrição, uma fórmula da lógica de primeira ordem que representa as condições que devem ser verdadeiras para que um determinado caminho de programa seja executado.

[Baldoni et al. 2018] e [Trtík 2013] citam o problema de explosão de caminhos como um desafio no uso de execução simbólica. Dependendo do tamanho de um programa, como há a exploração de todos os caminhos, pode-se aumentar o número de caminhos exponencialmente, representando um problema sério em relação ao tempo de resposta e eficiência. Porém, é importante ressaltar que o escopo e dimensão dos programas implementados em disciplinas de programação introdutória são pequenos, e podem não configurar o problema relatado.

Existem diversos resolvedores modernos, e neste trabalho foi utilizado o *SMT Solver Z3*[1] desenvolvido pela *Microsoft*. Este resolvedor suporta operadores aritméticos, vetores de bits de tamanho fixo, matrizes, tipos de dados, funções não interpretadas e quantificadores, o que a torna ideal para o escopo deste trabalho.

Para exemplificar os conceitos abordados, considere na Figura 1, um programa em Python que lê um número e mostra o quadrado desse número se for maior que 3. Caso contrário, mostra a sua raiz quadrada.

[1]Site oficial: `https://www.microsoft.com/en-us/research/project/z3-3/`.

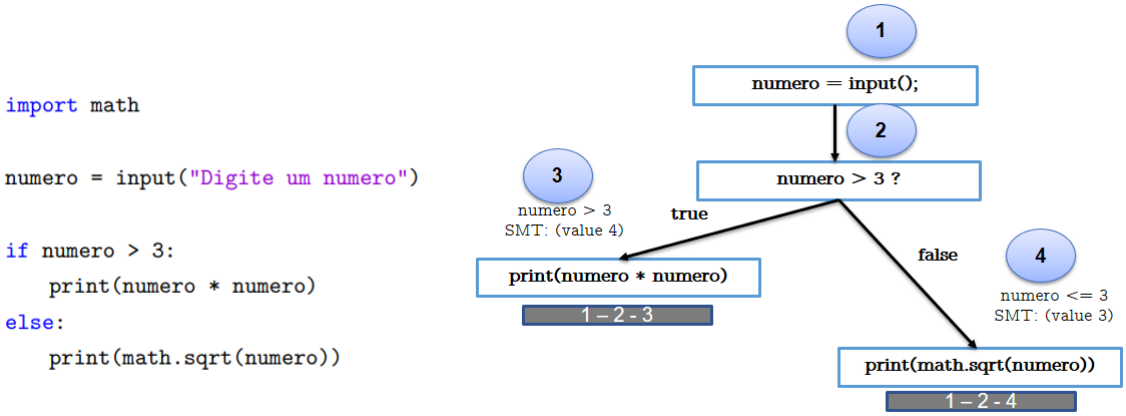

Figura 1. Demonstração dos caminhos de execução de um programa.

Há dois fluxos de execução no programa: um resultado para um número maior que 3 e outro resultado para um número menor ou igual a 3. Devido isso, seleciona-se o caminho de execução `1-2-3` para o primeiro o fluxo e `1-2-4` para o segundo fluxo.

Após selecionar todos os caminhos de execução do programa, para que cada caminho possa ser executado, verifica-se a sua restrição. Para executar o caminho `1-2-3`, é necessário que `numero` seja maior que 3. Diante disso, um resolvedor SMT é utilizado para verificar se essa restrição pode ser satisfeita, encontrando valores adequados, que a satisfaçam. No exemplo, foi encontrado o valor 4 para que ao executar o programa, o caminho seja explorado. O mesmo processo se repete para o caminho `1-2-4`, onde foi encontrado o valor 3.

3. Trabalhos relacionados

Nesta seção, são apresentados alguns trabalhos referentes a *auto-graders* de programação, com geração automática de casos testes utilizando Execução Simbólica.

[Ihantola 2006] propõe, através da ferramenta *JavaPathFinder*, a geração automatizada, utilizando bibliotecas da linguagem de programação Java, que utilizam execução simbólica. Como limitação, o docente deve manipular diversas funções em Java para gerar os valores de teste. Diferente de *JavaPathFinder*, a proposta deste trabalho não restringe a somente a utilização de linguagem de programação Java e manipulação de funções no gabarito para gerar casos de teste.

[Tillmann et al. 2013] propuseram a plataforma *Pex4Fun* para cursos MOOC (Cursos Online, Abertos e Massivos - do inglês: *Massive Open Online Courses*), utilizando a metodologia de aprendizagem orientada a jogos, para exercícios que utilizam o paradigma Orientado a Objetos. Os testes são gerados automaticamente, utilizando execução simbólica. A presente proposta visa atender disciplinas de programação introdutória que possui os principais tópicos abordados nas Universidades Brasileiras para o paradigma estruturado e não Orientado a Objetos e não orientada a metodologia de jogos.

[Bariansyah et al. 2021] e [Liu et al. 2019] apresentaram uma abordagem, que, utilizando execução simbólica, verifica a diferença semântica entre o modelo de referência do professor e a implementação do aluno, analisando-se os caminhos explorados em

ambas as implementações. Diferentemente, o foco principal da presente proposta é a exploração de caminhos utilizando somente o modelo de referência do professor, para a geração automática de casos de teste, tendo a execução contemplada com os principais tópicos utilizados por universidades brasileiras.

[Song et al. 2019] propuseram a geração automatizada de casos de teste para exercícios de programação funcional. Dado um programa de referência e a submissão de um aluno, a técnica gera um contraexemplo que captura a diferença semântica dos dois programas automaticamente. A abordagem utiliza um algoritmo de geração de contra-exemplos que combina busca enumerativa e técnicas de verificação simbólica de forma sinérgica. Utiliza a técnica de execução simbólica, somente para inteiros e strings. A presente proposta visa atender disciplinas de programação introdutória para o paradigma estruturado, foco de grande parte das ferramentas de correção de exercícios, e que não envolve o paradigma funcional, mas sim o estruturado.

No contexto brasileiro, dentre as ferramentas de correção automáticas mais citadas na literatura: Caderno de Algoritmos [Webber et al. 2010]; Prog-Test [De Souza et al. 2011]; [Pelz et al. 2012]; MOJO [Chaves et al. 2013]; AlgorithmOnline [Serafim and Zanini 2014]; BOCA [Galasso and Moreira 2014]; PCodigo [Oliveira et al. 2015]; CodeBrench [Galvão et al. 2016]; CodeTeacher [Santos et al. 2017]; [Francisco et al. 2018]; [Brito and Fortes 2019] e CodeTesting [Mabel et al. 2023]; não há a utilização de geração de casos de testes automatizados, tampouco a técnica de Execução Simbólica para gerar esses casos.

4. Arquitetura do Symb e a geração automática de casos de teste

Para projetar o **Symb**, mediante o objetivo de correção automática de exercícios de programação introdutória no contexto brasileiro, uma pesquisa foi realizada para identificar quais tópicos são comumente abordados nas ementas das disciplinas de Programação Introdutória dos cursos de Ciência da Computação de diversas universidades do país. A Tabela 1 apresenta os principais tópicos identificados e essas Universidades.

Observa-se, na Tabela 1, que os tópicos: (a) Instruções de E/S; (b) Operadores; (c) Estrutura de condição; (d) Estrutura de repetição; (e) vetores e matrizes; e (f) funções; são contemplados em todas as ementas. Com base nisso, a ferramenta proposta foi projetada contemplando esses tópicos. Adicionalmente, como cada universidade pode usar diferentes linguagens, a ferramenta foi projetada para ser facilmente extensível. O mecanismo de extensibilidade oferecido se dá por *plug-ins* que traduzem a sintaxe de diferentes linguagens de programação para uma linguagem interna à ferramenta, denominada Symb, que provê suporte a todos os recursos listados anteriormente. A versão atual do **Symb** oferece um *plug-in* para Python.

A arquitetura do **Symb** é mostrada na Figura 2, na qual é dividida em 3 módulos: (1) **Multilinguagem**, na qual há o processo de conversão da linguagem de origem para a linguagem **Symb**; (2) **Geração de casos de teste**, onde através da análise estática do código de gabarito, selecionam-se os caminhos para gerar os casos de teste; e (3) **Correção** dos exercícios.

Como o foco deste trabalho é a de geração automática de casos de teste, o modulo de geração de casos de teste será explicado com mais detalhes. O mecanismo de funci-

Tabela 1. Tópicos abordados em disciplinas.

Instituição	Instruções de E/S	Operadores	Estrutura de condição	Estrutura de repetição	Vetores/ Matrizes	Funções	Registros	Arquivos	Recursão
PUC-RIO	X	X	X	X	X	X		X	X
PUC-SP	X	X	X	X	X	X	X	X	
UERJ	X	X	X	X	X	X			
UFES	X	X	X	X	X	X			
UFF	X	X	X	X	X	X			
UFJF	X	X	X	X	X	X	X	X	
UFLA	X	X	X	X	X	X			
UFMG	X	X	X	X	X	X			
UFOP	X	X	X	X	X	X	X	X	
UFPE	X	X	X	X	X	X			X
UFRGS	X	X	X	X	X	X	X	X	
UFRJ	X	X	X	X	X	X	X	X	
UFSCAR	X	X	X	X	X	X	X	X	
UFSJ	X	X	X	X	X	X	X		
UFV	X	X	X	X	X	X			
UNICAMP	X	X	X	X	X	X			
USP	X	X	X	X	X	X		X	

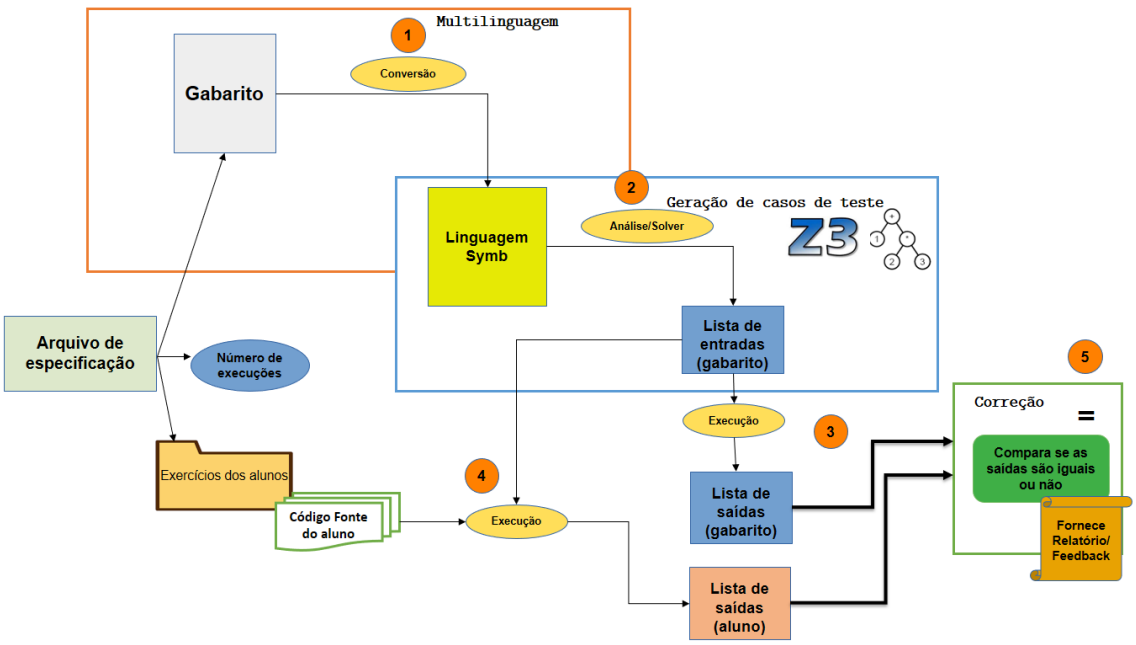

Figura 2. Arquitetura do Symb.

onamento da geração automática de casos de teste se divide em 5 etapas, como mostra a Figura 3, detalhadas a seguir.

Na etapa de **especificação**, o professor fornece duas informações: a solução do gabarito e o número de execuções para cada correção em um arquivo de especificação. Entende-se como o número de execuções, a quantidade de valores que devem ser gerados para cada caminho de execução do programa.

A seguir, no módulo de Multilinguagem, a linguagem de programação utilizada no gabarito é traduzida para a linguagem **Symb**, em que a estrutura do programa é analisada para produzir um conjunto de entradas para cada caminho de execução. A criação

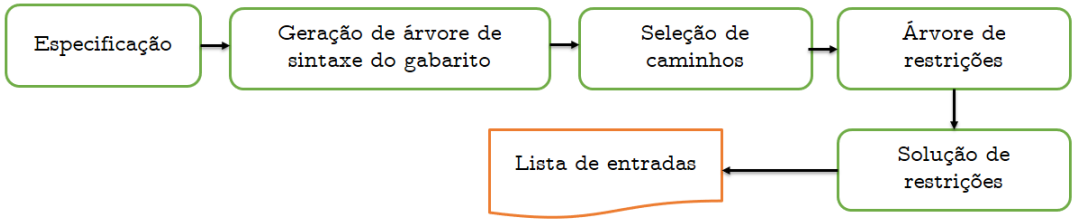

Figura 3. Etapas para geração automatizada de casos de testes.

de entradas se dá por representar diferentes condições de caminhos do programa como fórmulas da lógica expressas utilizando a linguagem do resolvedor *SMT Z3*. Entradas são produzidas executando o *Z3* sobre a fórmula, produzindo um conjunto de valores que a satisfaz. Este processo de geração é repetido para cada um dos diferentes caminhos do programa. Como resultado, gera-se uma lista de entradas com os valores obtidos pelo resolvedor. Nesta etapa, o docente pode utilizar esses valores para que qualquer auto-grader possa realizar as correções de seus exercícios.

Porém, se o docente optar por realizar a correção no **Symb**, deve-se inserir no arquivo de especificação, já mencionado, a localização das soluções dos discentes. De posse da lista de entradas do gabarito, o **Symb** executa o gabarito do professor e as soluções dos alunos com essas entradas e captura as saídas da execução desses programas em: uma lista de saídas do gabarito e uma lista de saídas para as soluções de cada aluno. Em seguida, as listas são comparadas e o módulo de Correção fornece *feedback* sobre os acertos e erros encontrados.

A Seção 5, a seguir, exemplifica todo o processo de geração de casos de teste descrito anteriormente.

5. Estudo de caso

Nesta seção, descreve-se um estudo de caso conduzido com o objetivo de validar a abordagem proposta para geração de casos de teste. Para isso, considerou-se uma lista contendo 36 exercícios da disciplina BCC701 - Programação de Computadores I do curso de Bacharelado em Ciência da Computação da Universidade Federal de Ouro Preto (UFOP). Para cada exercício, foi produzido o gabarito, soluções corretas e incorretas. Esse conjunto de soluções e o respectivo gabarito foram submetidos ao **Symb** que foi capaz de identificar soluções corretas e incorretas para todos os exercícios considerados. Ressalta-se que foi disponibilizado um repositório *online* contendo o código-fonte do **Symb**, bem como o estudo de caso completo (o link para acesso encontra-se na introdução).

A seguir, reporta-se o contexto de uso de *auto-graders* pelo professor e a abordagem por ele utilizada para construção de casos de teste, considerando apenas um de seus exercícios propostos, além da limitação de espaço do artigo, há uma grande semelhança entre os resultados de todos os exercícios. Finalmente, apresenta-se como o uso do **Symb** pode automatizar de forma apropriada a tarefa de construção de testes em *auto-graders*.

Para correção de exercícios, o professor utiliza a ferramenta *opCoders Judge*, proposta por [Brito and Fortes 2019], onde insere manualmente 15 casos de teste para cada exercício. Visando uma melhor cobertura dos exercícios corrigidos, o professor relatou que desenvolveu geradores de entradas aleatórias para cada um dos exercícios propostos.

Porém, gerar entradas aleatórias não garante que todos os possíveis caminhos do código serão considerados. Para ilustrar essa situação, considera-se um código Python que é a solução para um exercício que pede a construção de um programa para determinar se um valor de entrada é divisível por 10, 5 ou 2. Uma mensagem deve ser apresentada caso o número informado não seja divisível por nenhum destes valores. O código é apresentado a seguir:

```
n = input("Digite um número: ")
if n%10 == 0:
   print("Divisível por 10")
elif n%5 == 0:
   print("Divisível por 5")
elif n%2 == 0:
   print("Divisível por 2")
else:
   print("Não é divisível por 10, 5 ou 2")
```

Para geração de valores aleatórios, o professor implementou uma rotina simples que gera 15 números aleatórios no intervalo entre 1 e 100, como apresentado a seguir:

```
import random
for numero in range(15):
   savefile(random.randint(1,100))
```

A Tabela 2 mostra os resultados encontrados em uma execução do algoritmo e os distribui entre os diferentes de grupos de valores esperados e suas respectivas saídas.

Tabela 2. Valores de entrada e saída do gabarito com a geração aleatória.

Variáveis	Saídas do programa	Agrupamento de entradas
n	“Divisível por 10”	-
	“Divisível por 5”	55, 45, 45, 25
	“Divisível por 2”	34, 56, 4, 42
	“Não é divisível por 10, 5 ou 2”	57, 87, 51, 29, 1, 87, 97

Observa-se que, ao gerar os valores aleatoriamente para a execução do gabarito, os valores não cobriram satisfatoriamente todas as possíveis soluções esperadas. O professor relatou que dificilmente a geração aleatória cobre todas as possibilidades de suas soluções, sendo necessário repetir o processo até que um conjunto de entradas satisfatório seja produzido ou a implementação de um gerador de entradas específico, considerando todas as possibilidades de valores, que demanda maior esforço de codificação e tempo.

Para exemplificar o processo de geração de casos de teste e sua eficácia, considera-se o mesmo exercício apresentado anteriormente. Após a especificação de localização do gabarito e o número de execuções igual a 2, o **Symb** produz a estrutura com os diferentes caminhos do programa, conforme mostra a Figura 4.

De posse da estrutura, para cada caminho de execução, foi construído um *script* Z3 para satisfazer às suas respectivas restrições. Ao executar o *script*, se houver valores que o satisfaçam, esses são retornados como possíveis entradas para o programa. Esse

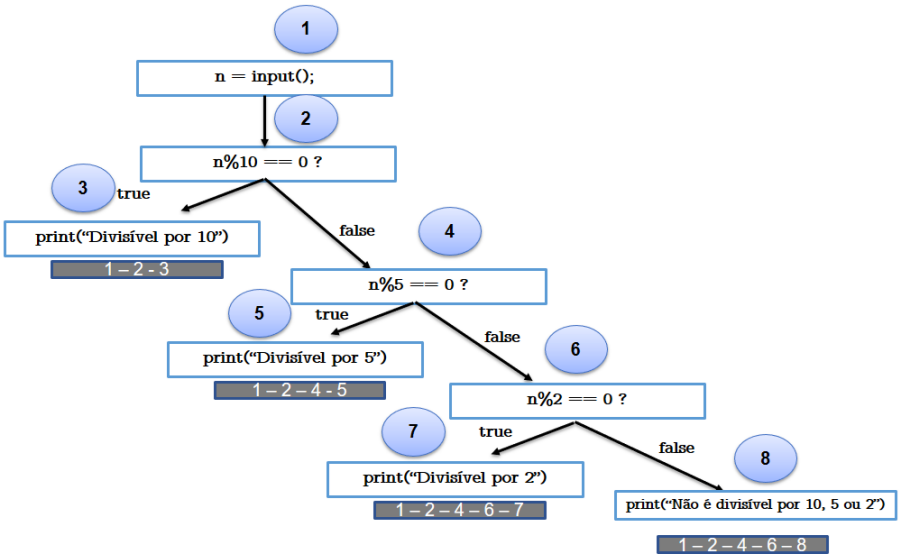

Figura 4. Estrutura dos diferentes caminhos do programa.

processo é repetido para cada um dos diferentes caminhos de execução. A Tabela 3, apresenta casos de teste produzidos para cada um dos caminhos apresentados na Figura 4.

Tabela 3. Caminhos e restrições do programa - 1ª execução.

Caminho	Restrição	Entrada gerada (n)
1-2	$n\%10 == 0$	10
1-3-4	$n\%10 != 0 \wedge n\%5 == 0$	5
1-3-5-6	$n\%10 != 0 \wedge n\%5 != 0 \wedge n\%2 == 0$	2
1-3-5-7	$n\%10 != 0 \wedge n\%5 != 0 \wedge n\%2 != 0$	3

Como apresentado na Tabela 3, valores de entradas foram gerados pelo *Z3 solver* para executar os caminhos indicados, com base nas restrições do programa. Por exemplo, para que seja executado o caminho `1-3-4`, o resto de divisão de `n` com `5` deve ser igual a `0`, para que a mensagem "`Divisível por 5`" seja mostrada, desconsiderando que `n` seja divisível por `10`, pois este caminho já foi explorado.

De posse de todos os valores de entrada gerados, uma lista de entradas foi criada. Como o número de execuções é igual a 2, outros valores de entrada diferentes dos valores computados anteriormente deverão ser gerados. Devido a isso, um novo *script* Z3 é gerado, com uma nova condição de caminho. Por exemplo: para acessar o caminho `1-3-4`, além do resto da divisão de `n` com `5` ser igual a `0`, `n` deve ser diferente de `5`, já que este valor foi gerado anteriormente. Se a restrição for satisfazível, um novo valor é gerado, e o caminho pode ser executado normalmente e a mensagem "`Divisível por 5`" é mostrada. A Tabela 4, mostra o esquema referente à segunda execução.

Diante disso, uma estrutura de dados foi criada, com a associação da variável do programa com a lista de entradas, que contém todos os valores obtidos, gerados automaticamente, por meio da exploração de todos os caminhos de execução, como mostra a Tabela 5.

De posse da estrutura, com os valores de entrada, qualquer *auto-grader* pode uti-

Tabela 4. Caminhos e restrições do programa - 2ª execução.

Caminho	Restrição	Entrada gerada (n)
1-2	n % 10 == 0 ∧ n ! = 10	20
1-3-4	n%10 ! = 0 ∧ n%5 == 0 ∧ n ! = 5	15
1-3-5-6	n%10 ! = 0 ∧ n%5 ! = 0 ∧ n%2 == 0 ∧ n ! = 2	4
1-3-5-7	n%10 ! = 0 ∧ n%5 ! = 0 ∧ n%2 ! = 0 ∧ n ! = 3	1

Tabela 5. Valores de entrada gerados.

Variável	Entradas
n	{10, 5, 2, 3, 20, 15, 4, 1}

lizá-los, e realizar a correção dos programas dos alunos. Neste trabalho, utiliza-se o próprio **Symb**, e seu interpretador executou o gabarito com cada entrada obtida e cada saída é registrada. A Tabela 6 mostra as saídas obtidas após a execução.

Tabela 6. Valores de entrada e saída do gabarito.

Variável	Execução	Entradas	Saídas obtidas
n	1	{10,5,2,3}	{“Divisível por 10”, “Divisível por 5”, “Divisível por 2”, “Não é divisível por 10, 5 ou 2”}
n	2	{20,15,4,1}	{“Divisível por 10”, “Divisível por 5”, “Divisível por 2”, “Não é divisível por 10, 5 ou 2”}

Em seguida, o **Symb** executa o código a ser corrigido, utilizando as entradas obtidas no passo anterior e cada saída é registrada. O objetivo é verificar se as entradas geram as mesmas saídas geradas pela execução do gabarito. A Tabela 7 mostra as saídas obtidas, considerando a solução correta do código do aluno, mostrada a seguir:

```
1  n = input("Digite um número: ")
2  div10 = n%10
3  div5 = n%5
4  div2 = n%2
5  if div10 == 0:
6     print("Divisível por 10")
7  elif div5 == 0:
8     print("Divisível por 5")
9  elif div2 == 0:
10    print("Divisível por 2")
11 else:
12    print("Não é divisível por 10, 5 ou 2")
```

Tabela 7. Valores de saída do código do aluno - Solução correta.

Variável	Execução	Entradas	Saídas - Gab.	Saídas - Aluno
n	1	{10,5,2,3}	{“Divisível por 10”, “Divisível por 5”, “Divisível por 2”, “Não é divisível por 10, 5 ou 2”}	{“Divisível por 10”, “Divisível por 5”, “Divisível por 2”, “Não é divisível por 10, 5 ou 2”}
n	2	{20,15,4,1}	{“Divisível por 10”, “Divisível por 5”, “Divisível por 2”, “Não é divisível por 10, 5 ou 2”}	{“Divisível por 10”, “Divisível por 5”, “Divisível por 2”, “Não é divisível por 10, 5 ou 2”}

Após a execução do código do aluno, compara-se se as saídas do gabarito e da execução do código do aluno são iguais. Se sim, o exercício está correto. Caso contrário, o exercício está incorreto. Neste caso, o **Symb**, informa que o exercício está correto. Considere o código a seguir, que representa a solução incorreta do aluno, na qual, ao invés de utilizar o operador %, utilizou-se /:

```
1 n = input("Digite um número: ")
2 if n/10 == 0:
3    print("Divisível por 10")
4 elif n/5 == 0:
5    print("Divisível por 5")
6 elif n/2 == 0:
7    print("Divisível por 2")
8 else:
9    print("Não é divisível por 10, 5 ou 2")
```

A Tabela 8 mostra as saídas obtidas, considerando a solução incorreta, com saídas diferentes do gabarito. Neste caso, o **Symb**, informa que o exercício está incorreto.

6. Conclusão

Neste artigo foi apresentada uma abordagem para geração automática de casos de teste utilizando execução simbólica a partir de um gabarito fornecido pelo professor. Esta técnica foi implementada na ferramenta **Symb** que foi usada em um estudo de caso envolvendo uma lista de exercícios de uma universidade brasileira.

O estudo de caso mostrou que a técnica proposta é promissora, pois conseguiu gerar casos de teste para todos os exercícios considerados e identificar todas as soluções corretas e incorretas para estas questões, incluindo a cobertura de todos os caminhos de execução em relação ao gabarito do professor.

Como trabalhos futuros, pretende-se: (1) continuar o desenvolvimento da ferramenta **Symb** para a implementação dos tópicos de estrutura de repetição e funções; (2) Conduzir experimentos que possam evidenciar empiricamente se a geração de casos de teste automatizados proposta pela ferramenta, diminui a sobrecarga dos docentes que utilizam auto-graders; (3) Implementar versões da ferramenta para plataformas

Tabela 8. Valores de saída do código do aluno - Solução incorreta.

Variável	Execução	Entradas	Saídas - Gab.	Saídas - Aluno
n	1	{10,5,2,3}	{“Divisível por 10”, “Divisível por 5”, “Divisível por 2”, “Não é divisível por 10, 5 ou 2”}	{“Não é divisível por 10, 5 ou 2”, “Não é divisível por 10, 5 ou 2”, “Não é divisível por 10, 5 ou 2”, “Não é divisível por 10, 5 ou 2”}
n	2	{20,15,4,1}	{“Divisível por 10”, “Divisível por 5”, “Divisível por 2”, “Não é divisível por 10, 5 ou 2”}	{“Não é divisível por 10, 5 ou 2”, “Não é divisível por 10, 5 ou 2”, “Não é divisível por 10, 5 ou 2”, “Não é divisível por 10, 5 ou 2”}

web e integração ao Ambiente Virtual de Aprendizagem (AVA), devido ao fato de atualmente **Symb** possuir somente a versão *desktop*; (4) Realizar experimentos em cursos de programação introdutória com o objetivo de avaliar o impacto de seu uso.

Agradecimentos

Os autores agradecem a Universidade Federal de Ouro Preto (PROPPI/UFOP) e a Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) pelo apoio ao desenvolvimento deste trabalho.

Referências

[Ábrahám 2015] Ábrahám, E. (2015). Building bridges between symbolic computation and satisfiability checking. In *Proceedings of the 2015 ACM on International Symposium on Symbolic and Algebraic Computation*, pages 1–6.

[Ala-Mutka 2005] Ala-Mutka, K. M. (2005). A Survey of Automated Assessment Approaches for Programming Assignments. *Computer Science Education*, 15(2):83–102. Publisher: Routledge _eprint: https://doi.org/10.1080/08993400500150747.

[Anand et al. 2009] Anand, S., Păsăreanu, C. S., and Visser, W. (2009). Symbolic execution with abstraction. *International Journal on Software Tools for Technology Transfer*, 11:53–67.

[Baldoni et al. 2018] Baldoni, R., Coppa, E., D’elia, D. C., Demetrescu, C., and Finocchi, I. (2018). A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):1–39.

[Bariansyah et al. 2021] Bariansyah, M. R. I., Rukmono, S. A., and Perdana, R. S. (2021). Semantic approach for increasing test case coverage in automated grading of programming exercise. In *2021 International Conference on Data and Software Engineering (ICoDSE)*, pages 1–6. IEEE.

[Boyer et al. 1975] Boyer, R. S., Elspas, B., and Levitt, K. N. (1975). Select—a formal system for testing and debugging programs by symbolic execution. *ACM SigPlan Notices*, 10(6):234–245.

[Brito and Fortes 2019] Brito, P. and Fortes, R. (2019). O uso de corretores automáticos para o ensino de programação de computadores para alunos de engenharia. In *Brazilian Symposium on Computers in Education (Simpósio Brasileiro de Informática na Educação-SBIE)*, volume 30, page 449.

[Chaves et al. 2013] Chaves, J. O. M., Castro, A. F., Lima, R. W., Lima, M. V. A., and Ferreira, K. H. (2013). Mojo: Uma ferramenta de auxílio à elaboração, submissão e correção de atividades em disciplinas de programação. In *XXI Workshop de Educação em Computação (WEI)-SBC 2013*.

[De Souza et al. 2011] De Souza, D. M., Maldonado, J. C., and Barbosa, E. F. (2011). Progtest: An environment for the submission and evaluation of programming assignments based on testing activities. In *2011 24th IEEE-CS Conference on Software Engineering Education and Training (CSEE&T)*, pages 1–10. IEEE.

[Francisco et al. 2018] Francisco, R. E., Ambrósio, A. P. L., Junior, C. X. P., and Fernandes, M. A. (2018). Juiz online no ensino de cs1-lições aprendidas e proposta de uma ferramenta. *Revista Brasileira de Informática na Educação*, 26(03):163.

[Galasso and Moreira 2014] Galasso, R. H. and Moreira, B. G. (2014). Integração do ambiente boca com o ambiente moodle para avaliação automática de algoritmos. *Anais do Computer on the Beach*, 5:22–31.

[Galvão et al. 2016] Galvão, L., Fernandes, D., and Gadelha, B. (2016). Juiz online como ferramenta de apoio a uma metodologia de ensino híbrido em programação. *Brazilian Symposium on Computers in Education (Simpósio Brasileiro de Informática na Educação - SBIE)*, 27(1):140. Number: 1.

[Ihantola 2006] Ihantola, P. (2006). Test data generation for programming exercises with symbolic execution in java pathfinder. In *Proceedings of the 6th Baltic Sea conference on Computing education research: Koli Calling 2006*, pages 87–94.

[King 1976] King, J. C. (1976). Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394.

[Liu et al. 2019] Liu, X., Wang, S., Wang, P., and Wu, D. (2019). Automatic grading of programming assignments: an approach based on formal semantics. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, pages 126–137. IEEE.

[Mabel et al. 2023] Mabel, E., Tavares, J., Buri, Y., and Pereira, M. (2023). Codetesting: Sistema de apoio ao ensino de lógica de programação. In *Anais Estendidos do III Simpósio Brasileiro de Educação em Computação*, pages 22–23. SBC.

[Oliveira et al. 2015] Oliveira, M. d., Nogueira, M., and Oliveira, E. (2015). Sistema de Apoio à Prática Assistida de Programação por Execução em Massa e Análise de Programas. In *Anais do Workshop sobre Educação em Computação (WEI)*, pages 90–99. SBC. ISSN: 2595-6175.

[Pelz et al. 2012] Pelz, F. D., de Jesus, E. A., and Raabe, A. L. (2012). Um mecanismo para correção automática de exercícios práticos de programação introdutória. *Anais do XXIII Simpósio Brasileiro de Informática na Educação*, 23(1).

[Santos et al. 2017] Santos, F. A., Segundo, P., and Telvina, M. (2017). CodeTeacher: Uma Ferramenta para Correção Automática de Trabalhos Acadêmicos de Programação em Java. page 1152, Recife, Pernambuco, Brasil.

[Serafim and Zanini 2014] Serafim, J. and Zanini, A. (2014). Ambiente Virtual de aprendizagem para algoritmos. In *Anais do Workshop de Desafios da Computação aplicada à Educação (DesafIE!)*, pages 38–46. SBC. ISSN: 0000-0000.

[Song et al. 2019] Song, D., Lee, M., and Oh, H. (2019). Automatic and scalable detection of logical errors in functional programming assignments. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–30.

[Tillmann et al. 2013] Tillmann, N., De Halleux, J., Xie, T., Gulwani, S., and Bishop, J. (2013). Teaching and learning programming and software engineering via interactive gaming. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 1117–1126. IEEE.

[Trtík 2013] Trtík, M. (2013). Symbolic execution and program loops. *Brno, Czech Republic*.

[Webber et al. 2010] Webber, C., Zenato, M., and Arrosi, T. (2010). Caderno de Algoritmos: um Ambiente de Resolução de Problemas de Programação. *RENOTE*, 8(3). Number: 3.