UNIVERSIDADE DE CAXIAS DO SUL
ÁREA DE CONHECIMENTO DE CIÊNCIAS EXATAS E ENGENHARIAS

MATTHIAS BEBBER BREDA

ANÁLISE ESTÁTICA DE CÓDIGO PARA O PORTAL DE ALGORITMOS DA UCS

CAXIAS DO SUL
2018

MATTHIAS BEBBER BREDA

ANÁLISE ESTÁTICA DE CÓDIGO PARA O PORTAL DE ALGORITMOS DA UCS

Trabalho de Conclusão de Curso para obtenção do título de Bacharel em Ciência da Computação pela Universidade de Caxias do Sul.

Orientador Prof. Dr. Ricardo Vargas Dorneles

CAXIAS DO SUL
2018

MATTHIAS BEBBER BREDA

ANÁLISE ESTÁTICA DE CÓDIGO PARA O PORTAL DE ALGORITMOS DA UCS

Trabalho de Conclusão de Curso para obtenção do título de Bacharel em Ciência da Computação pela Universidade de Caxias do Sul.

Aprovado em:___/___/_____

Banca Examinadora

Prof Dr. Ricardo Vargas Dorneles
Universidade de Caxias do Sul - UCS

Prof Dr. Andre Luis Martinotto
Universidade de Caxias do Sul - UCS

Prof MSc. Alexandre Erasmo Krohn Nascimento
Universidade de Caxias do Sul - UCS

“Os problemas significativos que enfrentamos não podem ser resolvidos no mesmo nível de pensamento em que estávamos quando os criamos.”

ALBERT EINSTEIN

AGRADECIMENTOS

Agradecimentos.

Agradeço principalmente a minha família, amigos e ao meu orientador Ricardo pela ajuda ao longo deste trabalho.

Matthias Bebber Breda

RESUMO

Análise estática de código é realizada sem a execução real do algoritmo, que é o oposto de análise dinâmica que é realizada quando o algoritmo está executando. O objetivo deste trabalho é desenvolver um analisador estático de código para o Portal de Algoritmos da UCS. Neste trabalho são abordadas as técnicas mais conhecidas de análise estática, *SSA* (*Static Single Assignment*) (do inglês, *Static Single Assignment* - Atribuição Única Estática) e Análise de Fluxo de Dados e Análise de Intervalo de Valores. Estas técnicas foram utilizadas para reconhecer erros que o analisador léxico e sintático do Portal de Algoritmos da UCS não consegue identificar (variáveis não inicializadas, variáveis não utilizadas, função que não define valor de retorno, código inacessível entre outros), ajudando assim o aluno, indicando onde existem possíveis erros no código.

Palavras-chave: Análise estática de código, Portal de Algoritmo da UCS, SSA, Análise de Fluxo de Dados.

ABSTRACT

Static code analysis is the analysis of code that is executed without the actual execution of the algorithm, which is the opposite of dynamic analysis that is executed when the algorithm is running. The objective of this work is to develop a static code analyzer for the Web-based Environment for Learning Introductory Programming. This work discusses the most known techniques of static analysis, *SSA* (*Static Single Assignment*), Data Flow Analysis and Values Interval Analysis. These techniques were be used to acknowledge errors which the lexical and syntactic parser can not identify (uninitialized variables, unused variables, function that does not define the return value, unreachable code and others), thus helping the student, indicating where there are possible errors in the code.

Keywords: Static Code Analysis, Web-based Environment for Learning Introductory Programming, SSA, Data Flow Analysis .

Lista de Figuras

Lista de Tabelas

Lista de Trechos de Código

LISTA DE SIGLAS

SSA	*Static Single Assignment*
e-SSA	*Extended Static Single Assignment*
UCS	*Universidade de Caxias do Sul*
CFC	*Componentes Fortemente Conexos*
JDK	*Java Development Kit*
JAR	*Java ARchive*

Sumário

1 INTRODUÇÃO

É prática comum nos cursos da área de computação começar o ensinamento de programação de computadores por uma disciplina na qual são discutidos conceitos básicos como tipos de dados, variáveis e constantes, ao mesmo tempo em que são apresentados comandos de entrada e saída de dados, operadores de atribuição, aritméticos, relacionais e lógicos (SOUZA, 2009).

Entre os meios que podem ser utilizados, estão os ambientes de aprendizagem de programação. O uso deste ambiente nos estágios iniciais do ensino de programação tem-se mostrado bastante produtivo por permitir que desde o início os estudantes tenham contato com um ambiente de desenvolvimento próximo ao que encontrarão em sua vida profissional, embora mais simples e usando uma linguagem de programação sem tantos recursos (SOUZA, 2009).

Porém os ambientes de aprendizagem de programação somente mostram erros léxicos, sintáticos e alguns semânticos (erros de tipo). Existem alguns erros que poderiam ser identificados mas que requerem uma análise mais complexa como uma análise de fluxo de dados.

Análise estática de código é a análise de código que é executada sem a execução real dos programas construídos a partir desse algoritmo, que é o oposto de análise dinâmica (GOMES et al., 2009). Para a maioria dos casos a análise é realizada em alguma versão do código fonte e em outros casos em forma de código de representação intermediária (GOMES et al., 2009).

Os tipos de erros que o analisador estático pode detectar são bastante diversificados, variando de erros de codificação, como por exemplo, variáveis não inicializadas, variáveis não utilizadas, funções que não definem o valor de retorno, código inacessível, expressões constantes, erros de índices de vetores e até padrões códigos como MIRSA-C/C++ ou CERT C/C++ (BIGGIN, 2012).

O Portal de Algoritmos da *UCS* (*Universidade de Caxias do Sul*) é um portal de ensino e aprendizagem onde o aluno tem a oportunidade de aprender e melhorar suas habilidades desenvolvendo algoritmos a partir de problemas que estão cadastrados no portfólio. É um ambiente *WEB* composto por quatro partes: um conjunto de problemas algorítmicos, o que se refere ao portfólio de problema, um editor de algoritmo, uma ferramenta de avaliação de solução e uma interface do professor (DORNELES; JR; ADAMI, 2010).

Porém o ambiente de desenvolvimento não tem uma ferramenta que auxilia o usuário a detectar erros que não são encontrados pela análise léxica e sintática. Este trabalho pretende encontrar estes erros:

- Variáveis não inicializadas.
- Variáveis não utilizadas.
- Funções que não define valor de retorno.
- Código inacessível.
- Expressões constantes.
- Erros de índices de vetores.
- *Loop* em situações específicas (nas situações em que é possível calcular a variável de controle do *loop*).

1.1 OBJETIVOS

Este trabalho apresenta como principal objetivo o desenvolvimento de um analisador estático de código para o Portal de Algoritmo da *UCS* capaz de encontrar erros que não são encontrados pela análise sintática e indica ao aluno onde está o erro para ajudá-lo no aprendizado de algoritmos.

1.2 ESTRUTURA DO TRABALHO

No Capítulo 2 é apresentada detalhadamente a descrição das técnicas de análise estática.

No Capítulo 3 é descrita a técnica utilizada para identificar cada erro.

No Capítulo 4 é apresentada a forma como o projeto foi desenvolvido em sua fase de implementação.

No Capítulo 5 são apresentados os testes realizados.

No Capítulo 6 são apresentadas as conclusões deste trabalho.

2 DESCRIÇÃO DAS TÉCNICAS

Qualquer análise de programa requer alguma representação abstrata do código para que seja possível extrair características importantes que possam ser usadas para realizar otimizações e encontrar erros no código (ATKINSON; GRISWOLD, 2001; HAVLAK, 1995). Linguagens de programação de alto nível são valiosas ferramentas porque permitem a especificação dos algoritmos em notações mais naturais para expressar os conceitos abstratos envolvidos (KENNEDY, 1981).

A linguagem de código intermediária é o fator chave para a eficiência das análises para otimização de código e identificação de erros e tem sido amplamente estudada (KOSCHKE; GIRAD; WURTHNER, 1998).

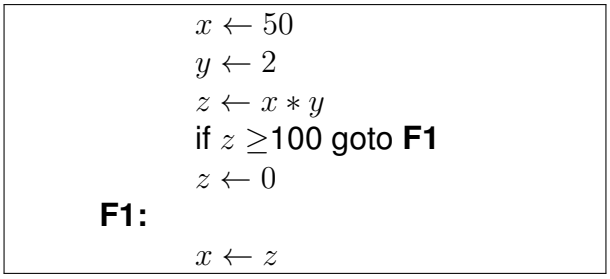
O *código de três endereços* é a linguagem intermediária utilizada neste trabalho. Foi escolhida pois é utilizada na maioria dos artigos e livros em que este trabalho se baseia. No *código de três endereços* cada instrução tem no máximo três operandos e é normalmente uma combinação de atribuição e um operador binário. Como por exemplo *x := y op z*, onde *x*, *y* e *z* são nomes, constantes ou objetos de dados temporários criado pelo compilador e *op* é um operador aritmético ou um operador lógico (AHO; ULLMAN; SETHI, 2008).

O *código de três endereços* tem semelhança a *Assembly* (linguagem de montagem). Os enunciados podem ter rótulos simbólicos (como *Labels* em *C*) e enunciados para fluxo de controle (*goto*, em português, *ir para*) (LEUPERS, 2001; AHO; ULLMAN; SETHI, 2008).

Também possui instrução de desvio condicional tal como *if x relop y goto L*. Onde *relop* é um operador relacional ($<, =, \geq, \leq, etc$). Se a condição *x relop y* for verdadeira o comando *goto L* é executado, senão segue o próximo comando da lista de enunciados (AHO; ULLMAN; SETHI, 2008).

Na Figura 1 é representado um código de três endereços com as instruções que foram descritas acima.

Figura 1 — Exemplo de Código de três endereços



Fonte: Próprio autor

Existem várias técnicas de *análise estática de código* e estas podem ser caracterizadas pela sua natureza e profundidade (GERMAN, 2003). Natureza se refere aos objetivos gerais da análise, como por exemplo, portabilidade do código ou encontrar erros, e a profundidade significa a profundidade da análise da técnica. A seguir são citadas algumas técnicas.

- Análise de Fluxo de Controle.
- Análise de Fluxo de Dados.
- Análise de Fluxo de Informações.
- Análise de Caminho de Função.
- Verificação Formal.
- Análise de Intervalo de Valores.
- Atribuição Única Estática.

O objetivo deste capítulo é descrever os métodos mais conhecidos de análise estática para detectar os erros não encontrados pela análise léxica e sintática.

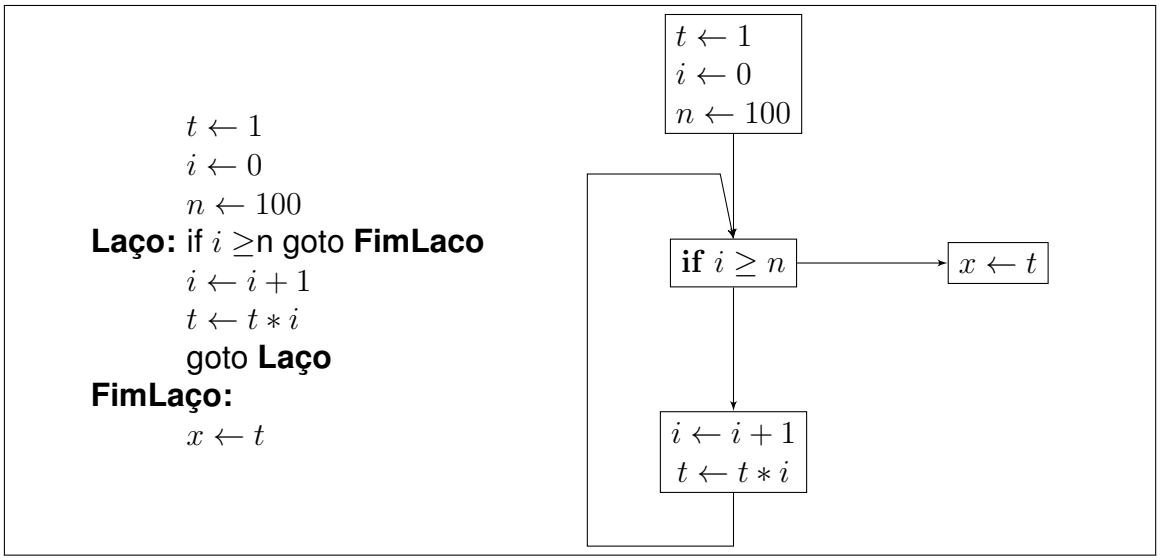
2.1 ANÁLISE DE FLUXO DE DADOS

Análise de fluxo de dados (em inglês, *Data Flow Analysis*) tem se mostrado como uma ferramenta útil para demonstrar a presença ou ausência de uma série de erros de programação (COOPER; HARVEY; KENNEDY, 2004). É uma importante técnica de verificação de software, pois é barata computacionalmente e confiável. (TAYLOR; OSTERWEIL, 1980).

Na análise de fluxo de dados o programa é analisado de um modo sistemático e as informações de uso de variáveis são coletadas de modo que cer-

tas deduções podem ser feitas sobre o efeito desses usos em outros pontos do código (FOSDICK; OSTERWEIL, 1976). O grafo de fluxo de controle (em inglês, *Control Flow Graph*) é usado para determinar até onde um determinado valor atribuído a uma variável pode alcançar (ZADECK, 1988). A Figura 2 mostra um código de três endereços e a sua representação em grafo de fluxo de controle.

Figura 2 — Código de três endereços e sua representação em grafo de fluxo de controle



Fonte: Próprio autor

Os nodos do grafo de fluxo de controle são blocos básicos (MOHNEN, 2002). Um bloco básico é uma sequencia linear de instruções de código que tem um único ponto de entrada (a primeira instrução executada) e um único ponto de saída (a última instrução executada) (KNOOP; KOSHCHUTZKI; STEFFEN, 1997).

Existem dois tipos de análise de fluxo de dados, uma se chama *forward analysis* (análise prospectiva) e *backward analysis* (análise para trás) (COOPER; HARVEY; KENNEDY, 2004). O algoritmo *reaching definitions analysis* (análise de definições incidentes) é do tipo de análise prospectiva, esse algoritmo calcula para cada ponto do código o conjunto de definições que potencialmente alcançariam aquele ponto do programa. Já o algoritmo chamado *live variable analysis* (análise de vida de variável) é do tipo análise para trás, e calcula para cada ponto do código as variáveis que potencialmente serão utilizadas posteriormente antes de sua próxima atribuição (KENNEDY, 1981). Com estas análises é possível descobrir uma série de erros, como detecção de variáveis não inicializadas e variáveis não utilizadas que são uns dos erros que este trabalho pretende identificar (COOPER; HARVEY; KENNEDY, 2004; MOHNEN, 2002).

2.1.1 Definições Incidentes

Uma definição *d* incide um ponto *p* (ou chega em um ponto *p*) se existir um caminho do ponto que se segue imediatamente a *d* até *p*, tal que *d* não seja *"morta"* ao longo do caminho. Então, se uma definição *d* de determinada variável *a* incide em um ponto *p*, *d* poderia ser o local no qual o valor de *a* usado em *p* teria ficado definido, mas *"matamos"* uma definição de uma variável *a* se, entre dois pontos ao longo do caminho, existir uma nova definição de *a* (AHO; ULLMAN; SETHI, 2008). A Figura 3 mostra a equação de definição incidente.

Figura 3 — Equação de Definição Incidente

$$EntradaBloco(B) \leftarrow \bigcup_{p\ em\ pred(B)} SaidaBloco(P)$$
$$SaidaBloco(B) \leftarrow DefinicoesGeradas(B) \bigcup (EntradaBloco(B) - Mortas(B))$$

Fonte: (AHO; ULLMAN; SETHI, 2008, p. 272)

A equação $DefinicoesGeradas(B)$ contém as definições geradas no bloco B e $Mortas(B)$ contém as definições mortas.

Para calcular as definições incidentes é necessário aplicar as equações da Figura 3 em cada bloco básico enquanto exista alguma alteração na equação $SaidaBloco(B)$, um algoritmo iterativo é apresentado abaixo no Trecho de Código 1.

Trecho de Código 1 — Algoritmo iterativo para análise de definições incidentes

```
1     //inicializa todos o bloco de entrada com vazio
2     //e a saída com as definições geradas em B
3     para cada bloco básico B faça
4         EntradaBloco(B) = estadoVazio;
5         SaidaBloco(B)   = DefinicoesGeradas(B);
6     fimpara
7
8     alterado = verdadeiro;
9     //enquanto houver alteração o algoritmo continua executando
10    enquanto (alterado) faça
11        alterado = falso;
12        para cada bloco básico B faça
13            //entrada do bloco recebe a saida dos precessores de B
14            EntradaBloco(B) = ⋃_{p∈pred(B)} SaidaBloco(P)
15
16            //grava o estado de saída do bloco em uma variável auxiliar
17            SaidaBlocoAux(B) = SaidaBloco(B);
18
19            //a saida do bloco B recebe a união das definições geradas com a
                  entrada do bloco menos as definições mortas
20            SaidaBloco(B) = DefinicoesGeradas(B)⋃(EntradaBloco(B) − DefinocoesMortas(B))
                  ;
```

```
21
22              se (SaidaBloco(B) <> SaidaBlocoAux(B)) então
23                  alterado = verdadeiro;
24              fimse
25          fimenquanto
26      fimenquanto
```

Fonte: (AHO; ULLMAN; SETHI, 2008, p. 272)

2.1.2 Análise de Vida de Variáveis

Uma variável está viva em uma aresta se há um caminho da aresta até um uso da variável que não passar por nenhuma outra definição da variável (AHO; ULLMAN; SETHI, 2008).

Para calcular a vida das variáveis são definidas equações em cada bloco, b, com dois conjuntos, $VivasNaEntrada(b)$ e $VivasNaSaida(b)$. O conjunto $VivasNaEntrada(b)$ contém todas as variáveis que estão vivas na entrada de b enquanto o conjunto $VivasNaSaida(b)$ contém todas as variáveis que estão vivas na saída de b (COOPER; HARVEY; KENNEDY, 2004; ALLEN; COCKE, 1976). As equações que definem esses conjuntos são apresentados na Figura 4.

Figura 4 — Equação para a análise de vida de variáveis

$$VivasNaEntrada(b) \leftarrow LocaisVivas(b) \bigcup (VivasNaSaida(b) - Mortas(b))$$
$$VivasNaSaida(b) \leftarrow \{\emptyset \text{ se b é o bloco de saída,}$$
$$\text{senão, } \bigcup_{s \in succs(b)} VivasNaEntrada(S) \}$$

Fonte: (AHO; ULLMAN; SETHI, 2008, p. 275)

O termo $LocaisVivas(b)$ contém as variáveis que foram usadas no bloco b e $Mortas(b)$ contém cada variável que foi definida em b. O termo $succs(b)$ contém os blocos sucessores de b.

Para resolver as equações da Figura 4 é necessário aplicar em cada bloco básico enquanto exista alguma alteração nas equações $VivasNaEntrada(b)$ e $VivasNaSaida(B)$, um algoritmo iterativo que é descrito no Trecho de Código 2.

Trecho de Código 2 — Algoritmo iterativo para análise de vida de variáveis

```
1     //inicializa todos os blocos com vazio
2     para cada bloco básico B faça
3         VivasNaEntrada(B) = estadoVazio;
4         VivasNaSaida(B) = estadoVazio;
5     fimpara
6
7     alterado = verdadeiro;
8     //enquanto houver alteração o algoritmo continua executando
9     enquanto (alterado) faça
10        alterado = falso;
11        para cada bloco básico B faça
12            //grava o estado de entrada do bloco em uma variável auxiliar
13            VivasNaEntradaAux(B) = VivasNaEntrada(B);
14            //grava o estado de saída do bloco em uma variável auxiliar
15            VivasNaSaidaAux(B) = VivasNaSaida(B);
16
17            //recebe as variaveis vivas das entradas dos blocos sucessores
18            //se o bloco for de saída não recebe nenhuma variável pois não tem
                  sucessor
19            VivasNaSaida(B) = ⋃_{s∈succs(b)} VivasNaEntrada(S)
20
21            VivasNaEntrada(B) = LocaisVivas(b) ⋃ (VivasNaSaida(b) − Mortas(b))
22
23            //verifica se foi alterado
24            se ((VivasNaEntrada(B) <> VivasNaEntradaAux(B)) ou (VivasNaSaida(B)
                  <> VivasNaSaidaAux(B)))
25                alterado = verdadeiro;
26            fimse
27        fimpara
28    fimpara
```

Fonte: (AHO; ULLMAN; SETHI, 2008, p. 275)

2.2 SSA

SSA (*Static Single Assignment*), em português *Atribuição Única Estática*, é uma representação intermediária que compiladores utilizam para facilitar a análise do programa e otimização (BRIGGS et al., 1998).

A propriedade essencial do *SSA* é que cada uso de uma variável se refere a exatamente uma definição ou, de forma equivalente, que a definição de uma variável domina cada utilização (STAIGER et al., 2007). Isto significa que a variável é um nome de um valor (no sentido matemático) e não de um local de armazenamento (KESLEY, 1995; STAIGER et al., 2007).

A forma *SSA* e o grafo de controle de dependências são propostos para representar o fluxo de dados e as propriedades de fluxo de controle de programas (STAIGER et al., 2007; CYTRON et al., 1991). *SSA* pode ser vista como uma

representação esparsa para a informação contida nas cadeias de uso-definição e definição-uso clássicas, e se tornou em muitas aplicações a representação primária do programa (BRIGGS et al., 1998).

Com *SSA* é possível identificar código morto utilizando o algoritmo de propagação de cópias (WEGMAN; ZADECK, 1995).

Figura 5 — Bloco de código sequencial e sua conversão para a forma *SSA*

$i \leftarrow .....$	$i_0 \leftarrow .....$
$j \leftarrow i + i$	$j_0 \leftarrow i_0 + i_0$
$i \leftarrow i + j$	$i_1 \leftarrow i_0 + j_0$
$k \leftarrow i + j$	$k_0 \leftarrow i_1 + j_0$
Antes	**Depois**

Fonte: Próprio autor

A Figura 5 mostra em um simples código a propriedade de atribuição única que a forma *SSA* aplica. Como houve uma atribuição ao i_0 ele passou a ser i_1 no próximo uso.

A função ø(phi) representa a junção de múltiplos nomes SSA em um único nome e é inserida em cada ponto de junção do fluxo de controle (SREEDHAR et al., 1999; HAVLAK, 1995). Por exemplo, quanto mais funções ø existirem mais processamento é necessário (STAIGER et al., 2007).

Existem três algoritmos que convertem o código na forma SSA, um deles se chama *SSA "mínimo"*, o segundo se chama *SSA "podado"* e o terceiro se chama *SSA "semi podado"*. Em ordem de complexidade vem o algoritmo *"mínimo"*, depois o *"semi podado"* e por último o *"podado"*. O que diferencia entre os três algoritmos é a quantidade de funções ø que são colocadas no código (BRIGGS et al., 1998).

O algoritmo *"mínimo"* insere funções ø em todo ponto em que o fluxo de controle se funde e reúne dois ou mais nomes *SSA* para um único nome. Nesse caso pode ser inserido um função ø para juntar dois valores que nunca vão ser utilizados depois da fusão no fluxo de controle pois os valores não estão vivos (terminologia de análise de fluxo de dados) (BRIGGS et al., 1998).

Já o algoritmo *"podado"* utiliza um análise de fluxo de dados que indica se o valor é potenciavelmente vivo, diminuindo o número de funções ø e, assim, diminuir o número de nomes *SSA*, porém este algoritmo tem um custo elevado

pois necessita calcular a informação de vida de cada variável (BRIGGS et al., 1998).

E por último o algoritmo *"semi podado"*, este por sua vez tenta reduzir o número de funções ø diminuindo o custo elevado para calcular a informação da vida de cada variável. Se uma variável nunca estiver viva na entrada de um bloco básico não é necessária uma função ø. Então durante a construção da forma *SSA* funções ø para quaisquer variáveis de bloco local são omitidas (BRIGGS et al., 1998).

Basicamente os três algoritmos diferem no seu tempo e complexidade, o algoritmo *"mínimo"* não processa a informação de fluxo de dados então é potenciavelmente menos custoso que o algoritmo *"podado"* e *"semi podado"*, porem resulta em uma forma *SSA* com muitas funções ø (BRIGGS et al., 1998). Nas próximas seções são descritos cada algoritmo.

Na Seção 2.2.5 é descrito o algoritmo chamado de *propagação de constantes*. Este algoritmo é aplicado na forma *SSA* convertida de forma que possa ser identificado código morto que é um dos erros que este trabalho pretende identificar (STAIGER et al., 2007).

2.2.1 SSA mínimo

O Trecho de Código 3 mostra um algoritmo simples para construção da forma *SSA* mínima a partir de uma representação de grafo de fluxo de controle. O algoritmo tem duas etapas básicas, que são determinar os lugares das funções ø e renomear as variáveis.

Trecho de Código 3 — Algoritmo para construir SSA mínima

```
1 procedimentoCalculaSSAMinimo()
2     /* Passo 1: Determinar os locais para colocar as funções phi */
3     Calcular a árvore de dominância e a fronteira de dominância
4
5     /* Passo 2: Colocar as funções phi */
6     Para cada variavel, v, faça
7         A(v) = {blocos que contém atribuição a v};
8         Coloque uma função phi na fronteira de dominância de A(v)
9     fimpara
10
11    /* Passo 3: Renomear cada variável, trocando v, com o nome apropriado criado
          pelas funções phi */
12    Para cada variavel, v, faça
13        Contadores(v) = 0;
14        Pilhas(v) = pilhaVazia();
15    fimpara
16
```

```
17      Pesquisa(BlocoInicial);
18  end
19
20  /* Recursivamente pesquisa pela árvore de dominância renomeando as variáveis */
21  procedimento Pesquisa(Bloco B)
22  begin
23      para cada nodo phi, v = phi(...) no bloco B faça
24          i = Contadores(V);
25          Troca v por v_i;
26          empilha(i,Pilhas(v));
27          Contadores[v] = i + 1;
28      fimpara
29
30      para cada instrução v = x op y no Bloco B faça
31          Troca x por x_i, onde i = topo(Pilhas(x));
32          Troca y por y_i, onde i = topo(Pilhas(y));
33          i = Contadores(v);
34          Troca v por v_i;
35          empilha(i,Pilhas(v));
36          Contadores(v) = i + 1;
37      fimpara
38
39      para cada sucessor, S, do bloco B faça
40          j = qualPred(s , B);
41          para cada função phi, P , em S faça
42              v = j^th operando de p;
43              Troca v por v_i, onde i = topo(Pilhas[v]);
44          fimpara
45      fimpara
46
47      para cada filho, C , do bloco na árvore de dominância faça
48          Pesquisa(C);
49      fimpara
50
51      para cada instrução, v = x op y, ou nodo phi, v = phi(...), no Blobo B faça
52          desempilha(Pilhas(v));
53      fimpara
54
55  end
```

Fonte: (BRIGGS et al., 1998, p. 8)

O primeiro passo para inserir as funções ø é construir a árvore de dominância do grafo de controle de fluxo e calcular a fronteira de dominância para cada nodo. A fronteira de dominância de um nodo N é um conjunto de nodos X em que N domina o predecessor de X, mas N não domina estritamente o X. Isto define precisamente os pontos onde as funções ø tem que ser inseridas, já que a árvore de dominância inclui somente blocos que podem ser alcançados ao longo de caminhos diferentes do fluxo de controle (BRIGGS et al., 1998).

Após a inserção das funções ø as variáveis tem que ser renomeadas para o novo nome criado. Isto é realizado num único passeio recursivo na

árvore de dominância, mostrado no procedimento *Pesquisa()*. Para cada nome no código original, *Pesquisa()* mantém duas estruturas de dados. A primeira, *Contadores(v)*, contém o índice que será atribuído na próxima definição de *v*. A segunda, *Pilhas(v)*, mantém o índice atual para o *v*. A cada definição de *v*, *Pesquisa()* renomeia *v* com o índice do *Contadores(v)*, empilha o valor na *Pilhas(v)*, e incrementa o *Contadores(v)*.

Durante a primeira etapa são renomeadas as variáveis, sendo incrementados os vários contadores e empilhados os novos nomes em suas apropriadas pilhas. Em seguida, reescreve funções ø em cada bloco sucessor no grafo de fluxo de controle para que o nome seja herdado do bloco atual e tenha o índice corrente.

A função *qualPred()* determina qual função ø no sucessor corresponde no bloco atual. Para continuar a pesquisa, o algoritmo procura em cada filho na árvore de fronteira de dominância. Ao retornar da recursão, processa o bloco atual novamente para desempilhar cada pilha e contadores adicionados enquanto processava.

2.2.2 SSA podado

Para construir a forma *SSA podada* é necessário primeiramente executar a *análise de vida de variáveis* para definir as variáveis vivas na entrada de cada bloco básico (CHOI; CYTRON; FERRANTE, 1991). Na Seção 2.1.2 foi abordado um algoritmo para este cálculo.

A construção do *SSA podada* é muito similar à construção do *SSA mínimo*. No Trecho de Código 3 é necessário somente adicionar um passo que calcula a *análise de vida de variáveis* e modifica o primeiro passo que define aonde as funções ø serão inseridas. O *SSA* mínimo insere a função ø para cada *v* em cada nodo na fronteira de dominância dos blocos básicos que contenha a atribuição a *v*. No *SSA* podado isto muda, é somente inserido em blocos onde *v* esteja vivo na entrada do bloco. O algoritmo do *SSA* podado pode ter um custo maior que o *SSA* mínimo, pois para inserir as funções ø exige a *análise de vida de variáveis* que tem um custo computacional elevado.

2.2.3 SSA semi podado

A vantagem de velocidade e eficiência da forma *semi podado* sobre os outros dois algoritmos se baseia na observação de que muitos nomes em uma rotina são definidos e usados inteiramente em um único bloco básico (BRIGGS et al., 1998). Por exemplo, o compilador tipicamente gera nomes temporários para executar passos intermediários, esses nomes gerados frequentemente tem vidas curtas. A forma *SSA semi podado* aproveita esta informação pelo cálculo do conjunto de nomes que são vivos na entrada para *alguns* blocos básicos. São chamados de *não-locais* esses nomes.

O número resultante de *funções ø* fica entre o algoritmo *mínimo* e o *podado*, mas o cálculo dos nomes *não-locais* é muito mais barato do que a *análise de vida de variáveis* (BRIGGS et al., 1998). Portanto, a forma *semi podado* consegue ter um custo de processamento consideravelmente menor que a versão *podado* mas com o número de *funções ø* parecido.

Trecho de Código 4 — Algoritmo para identificar nomes não-locais

```
1     //inicializa conjunto nao locais
2     nao_locais = estadoVazio;
3
4     //para cada bloco básico
5     para cada bloco B faça
6         mortos = estadoVazio;
7         para cada intrução v = x op y no bloco B faça
8             se x não pertence ao mortos então
9                 nao_locais = nao_locais ⋃ {x}
10            fimse
11            se y não pertence ao mortos então
12                nao_locais = nao_locais ⋃ {y}
13            fimse
14
15            mortos = mortos ⋃ {v}
16        fimpara
17    fimpara
```

Fonte: (BRIGGS et al., 1998, p. 12)

Para descobrir os nomes *não-locais* é utilizado o algoritmo mostrado no Trecho de Código 4. O compilador faz uma análise simples em cada bloco básico, quando descobre um operando que não foi *morto* no bloco ele é definido como um nome *não-local*, e o nome que foi definido na atribuição é colocado no conjunto *mortos*. O conjunto de *não-locais* é inicializado somente uma vez, o conjunto *mortos* é reinicializado para cada bloco. Portanto é muito mais simples do que calcular a *análise de vida de variáveis* completa.

2.2.4 Comparação entre os algoritmos mínimo, podado e semi podado

Nas ultimas seções foram descritos 3 algoritmos de *SSA*, *mínimo*, *podado* e *semi podado* que utilizam meios diferentes para definir onde as funções ø serão colocadas. Na Tabela 1 temos uma comparação com várias rotinas em cada algoritmo, comparando número de funções ø adicionadas e tempo para construção da forma *SSA*. Os testes foram executados em um computador *Sparc10 modelo 512* rodando a 50 MHz com 1 MB de *cache* e 115 MB de memória RAM (BRIGGS et al., 1998). O número de funções ø da forma *semi podada* esta sempre entre a versão *mínima* e *podada* e o tempo para construir a forma *semi podada* é sempre menor que a versão *mínima* e *podada*.

Tabela 1 — Comparação entre os algoritmos SSA

	Número de nodos ø			Tempo para construir a forma SSA		
Rotina	Mínimo	Semi	Podada	Mínimo	Semi	Podada
twldrv	73778	11989	9886	1.265	0.332	0.427
deseco	8610	2216	1842	0.231	0.172	0.232
ddeflu	5852	1560	1222	0.116	0.070	0.091
iniset	5364	1080	462	0.127	0.101	0.157
debflu	4715	1748	1542	0.098	0.069	0.087
paroi	3597	767	632	0.079	0.060	0.077
efill	3170	357	74	0.047	0.020	0.025
inisla	2722	267	141	0.048	0.025	0.034
tomcatv	2699	365	145	0.051	0.033	0.042
pastem	2584	374	62	0.055	0.036	0.049
prophy	2021	436	401	0.054	0.042	0.054
inithx	1967	267	85	0.042	0.033	0.044
debico	1880	171	112	0.045	0.030	0.039
repvid	1094	141	45	0.029	0.020	0.031
bilan	1080	70	34	0.028	0.020	0.028

Fonte: (BRIGGS et al., 1998, p. 16)

2.2.5 Propagação de Constantes

O algoritmo de propagação de constantes tem como princípio substituir os valores conhecidos de variáveis em expressões em que são utilizadas e assim detectando código morto (STAIGER et al., 2007).

Trecho de Código 5 — Algoritmo exemplo para propagação de constantes

```
1    //
2    se X = verdadeiro então
3        Y1 = 8;
4    senão
```

```
5            Y2 = 10;
6
7        Y3 = phi(Y1, Y2);
```

Fonte: Próprio autor

Inicialmente o algoritmo assume que cada bloco básico não é executável (nunca foi executado) e cada variável é uma constante com um valor até agora desconhecido (denotado $\top$) (STAIGER et al., 2007). Suponha que o algoritmo encontre que a variável *X* não é uma constante (denotado $\bot$) portanto se assume que os dois caminhos do condicional podem ser tomados. Quando $Y_1 = 8$ é executado, a suposição sobre o Y_1 é trocada de $\top$ por 8 e em todos seus usos são notificados com o valor 8 (cada uso de Y_1 realmente tem o valor 8 graças à propriedade de atribuição única). Então a função ø é combinação de 8 com Y_2. O segundo operando da função ø no entanto está associado no ponto de junção que corresponde à ramificação do *senão*, que até então era considerada não executável, o algoritmo utiliza $\top$ para o segundo operando. Como a variável *X* pode tomar o valor *falso* pois não é uma constante então o segundo caminho é tomado, e o comando $Y_2 = 10$ é executado e o Y_2 recebe o valor de 10. Por fim onde o Y_2 é utilizado é trocado pelo valor 10, então a função ø agora tem 8 e 10 como valores possíveis. Se no final da análise algum bloco estiver marcado como não executável este bloco é declarado como morto e pode ser removido do código (WEGMAN; ZADECK, 1995).

Com isso é possível simular o algoritmo propagando as constantes e eliminando o código morto encontrado (STAIGER et al., 2007).

2.3 ANÁLISE DE INTERVALO DE VALORES

Análise de Intervalo de Valores (do *inglês, Value Range Analysis*) é utilizada para determinar os limites de intervalo de valores assumidos pelas variáveis em vários pontos do código de um programa (XU, 2001). A informação de intervalos pode ser usada para remover testes redundantes, escolher representação de dados, verificar a operação correta e prover informações para diagnósticos (BIRCH; ENGELEN; GALLIVAN, 2004; HARRISON, 1977).

Informações sobre o intervalo de valores de variáveis podem ser propagadas por um programa do mesmo jeito que a propagação de constantes (BLUME; EIGENMANN, 1994).

2.3.1 Intervalos de Valores

Um intervalo de uma variável *x* pode ser definido como *x* = $\perp$ (intervalo vazio) ou *x = [l,u]* com $l \leq u$, chamando-se *l* de limite inferior e *u* de limite superior (GOUGH; KLAEREN, 1996). Para denotar o intervalo inferior de *x* utilizamos $x\downarrow$ e para o superior $x\uparrow$ (HARRISON, 1977).

No intervalo de valores assumimos que as operações aritméticas estão disponíveis para tipos ordinais (GOUGH; KLAEREN, 1996). Operações como $\bigcup$ e $\bigcap$ também entram para a solução de intervalos. No entanto, as operações aritméticas em intervalo de valores utilizadas neste trabalho são diferentes de sua semântica usual e os intervalos podem estar entre $-\infty$ e $+\infty$.

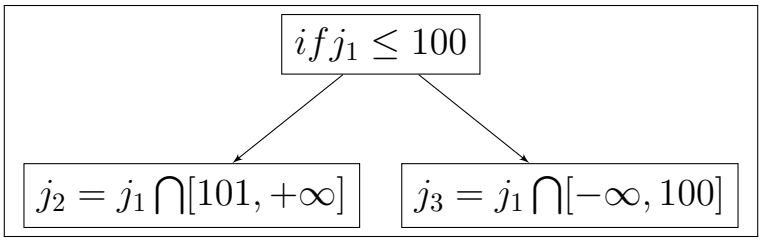
2.3.2 e-SSA

A forma *e-SSA* (*Extended Static Single Assignment*), em português, atribuição única estática estendida, é uma versão estendida do *SSA* como o próprio nome diz. O *e-SSA* foi criado para um único propósito, facilitar o processamento de intervalo de valores. No artigo de HARRISON que trata sobre análise de intervalo de valores, ele destaca dois problemas difíceis de se tratar, um deles é encontrar *pontos de corte* de um laço e o outro problema é a propriedade de intervalos de expressões booleanas complexas que ocorrem em testes (HARRISON, 1977). Ambos problemas se tornam praticamente triviais utilizando o *e-SSA* (GOUGH; KLAEREN, 1996).

A diferença entre *SSA* e *e-SSA* é que uma nova variável é criada com instruções que definem o intervalo. Por exemplo, $i_1 = [10, +\infty]$ nos diz que os valores possíveis para a variável i_1 estão contidos no intervalo de 10 e $+\infty$ (RODRIGUES; CAMPOS; PEREIRA, 2013).

Desvios condicionais contém informações que podem ser usadas para fazer com que a análise de intervalo seja mais precisa (GOUGH; KLAEREN, 1996). Com essas informações é possível, por exemplo, dizer que na Figura 6 após o teste condicional $if\ j_1 \leq 100$ resultar em verdadeiro o intervalo de valores da variável é $j_1 \bigcap [-\infty, 100]$, se resultar falso o intervalo é $j_1 \bigcap [101, +\infty]$.

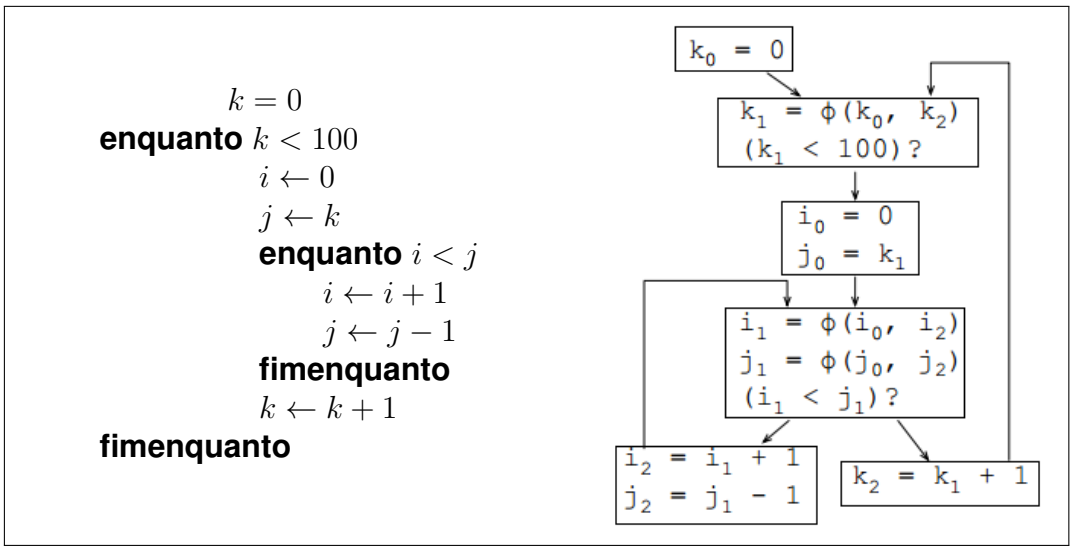
Figura 6 — Exemplo de condicional.



Fonte: Próprio autor

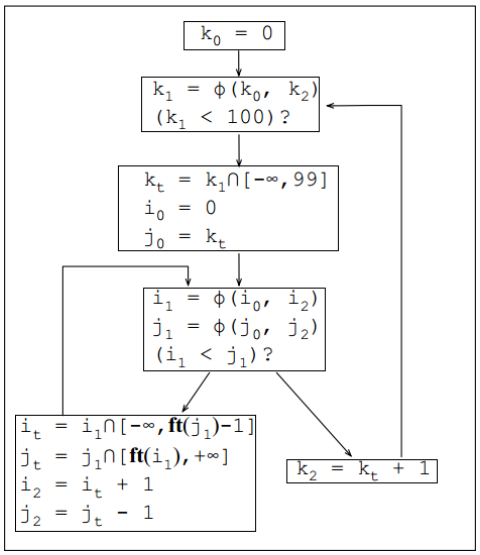
Na Figura 7 temos um código em português estruturado e sua representação na forma *SSA*. E na Figura 8 temos a representação desse código na forma *e-SSA*. Pode-se perceber que quando há uma comparação entre duas variáveis é colocado o rótulo *ft*, indicando que o valor de uma variável depende do valor de outra, mas esse valor vai ser obtido posteriormente no algoritmo de propagação de intervalos (GOUGH; KLAEREN, 1996).

Figura 7 — Código em português estruturado e sua representação em *SSA*



Fonte: (RODRIGUES; CAMPOS; PEREIRA, 2013, p. 3)

Figura 8 — Representação em *e-SSA* do código da Figura 7 (RODRIGUES; CAMPOS; PEREIRA, 2013)



Fonte: (RODRIGUES; CAMPOS; PEREIRA, 2013, p. 3)

Para gerar a forma *e-SSA* é preciso fazer uma pequena alteração no algoritmo da forma *SSA* descrito no Capítulo 2.2. Para cada derivação de um condicional será criada uma nova variável (somente para as que estiverem vivas) com a atribuição dos intervalos possíveis, como exemplificado na Figura 6 (RODRIGUES; CAMPOS; PEREIRA, 2013).

2.3.3 Propagação de intervalos de valores

Para fazer análise de intervalo de valores é necessário resolver um sistema de restrição (RODRIGUES; CAMPOS; PEREIRA, 2013). As restrições são construídas em torno de uma função de avaliação *e*. O sistema de restrições para resolver o problema de intervalo de valores é apresentado na Figura 9.

Figura 9 — Sistema de restrições

$$Y = [l, u] \qquad e(Y) = [l, u]$$

$$Y = \phi(X_1, X_2) \qquad \frac{I[X_1] = [l_1, u_1] \qquad I[X_2] = [l_2, u_2]}{e(Y) = [l_1, u_1] \bigcup [l_2, u_2]}$$

$$Y = X_1 + X_2 \qquad \frac{I[X_1] = [l_1, u_1] \qquad I[X_2] = [l_2, u_2]}{e(Y) = [l_1 + l_2, u_1 + u_2]}$$

$$Y = X_1 \times X_2 \qquad \frac{\begin{array}{c} L = \{l_1 l_2, l_1 u_2, u_1 l_2, u_1 u_2\} \\ I[X_1] = [l_1, u_1] \qquad I[X_2] = [l_2, u_2] \end{array}}{e(Y) = [\min(L), \max(L)]}$$

$$Y = aX + b \qquad \frac{I[X] = [l, u] \qquad k_l = al + b \qquad k_u = au + b}{e(Y) = [\min(k_l, k_u), \max(k_l, k_u)]}$$

$$Y = X \bigcap [l', u'] \qquad \frac{I[X] = [l, u]}{e(Y) = [l, u] \bigcup [l', u']}$$

Fonte: (RODRIGUES; CAMPOS; PEREIRA, 2013, p. 2)

Cada restrição é extraída de uma instrução da forma *e-SSA* do programa. A solução para esse sistema de restrição é a solução da análise de intervalo de valores (RODRIGUES; CAMPOS; PEREIRA, 2013). As restrições extraídas do código *e-SSA* da Figura 7 são apresentadas na Figura 10.

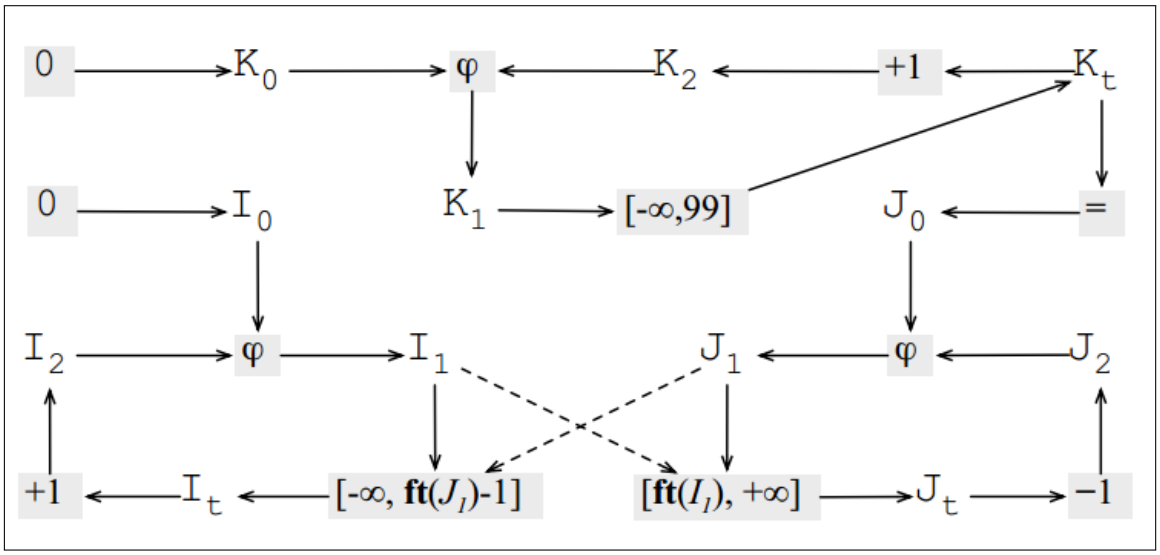
Figura 10 — Restrições extraídas do código *e-SSA* da Figura 7

$$
\begin{aligned}
k_0 &= 0 \\
k_1 &= \phi(k_0,\ k_2) \\
k_t &= k_1 \cap [-\infty, 99] \\
i_0 &= 0 \\
j_0 &= k_t \\
i_1 &= \phi(i_0,\ i_2) \\
j_1 &= \phi(j_0,\ j_2) \\
i_2 &= i_t + 1 \\
j_2 &= j_t - 1 \\
k_2 &= k_t + 1 \\
j_t &= j_1 \cap [\mathbf{ft}(i_1), +\infty] \\
i_t &= i_1 \cap [-\infty, \mathbf{ft}(j_1) - 1]
\end{aligned}
$$

Fonte: (RODRIGUES; CAMPOS; PEREIRA, 2013, p. 2)

Para isso, o primeiro passo é construir o grafo de dependências do código em formato *e-SSA* para representar as restrições extraídas do código fonte (RODRIGUES; CAMPOS; PEREIRA, 2013). Para cada variável *V* criamos um vértice variável N_v, e para cada restrição *C*, criamos um vértice restrição N_c. Adicionamos uma aresta de N_v para N_c se *V* é usada em *C* e adicionamos uma aresta de N_c para *V* se a restrição *C* define *V* (COUSOT; COUSOT, 1977; RODRIGUES; CAMPOS; PEREIRA, 2013).

Figura 11 — Restrições extraídas do código *e-SSA* da Figura 7



Fonte: (RODRIGUES; CAMPOS; PEREIRA, 2013, p. 3)

O próximo passo é resolver o grafo de dependências e encontrar os

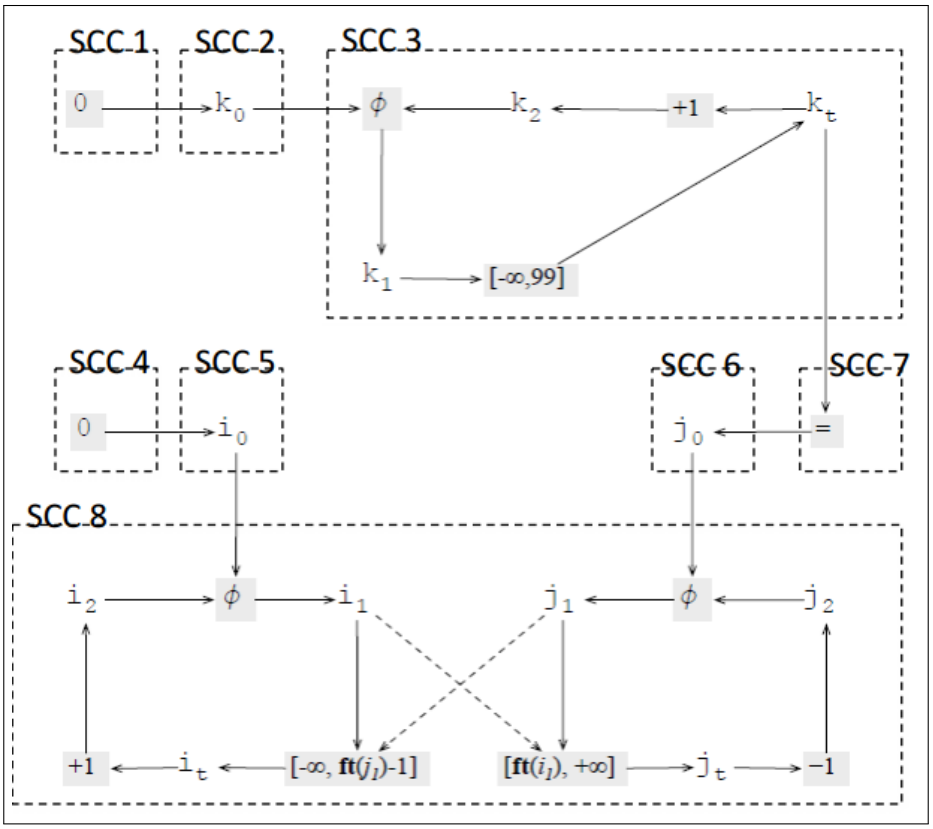
intervalos das variáveis inteiras do código. Essa etapa é dividida em quatro partes. A primeira é encontrar e reduzir *CFC* (*Componentes Fortemente Conexos*) do grafo, a segunda é alargar o intervalo das variáveis, a terceira é colocar os valores corretos nas restrições obtidas nos desvios condicionais e por ultimo a quarta etapa é estreitar o intervalo das variáveis tanto quanto possível (GOUGH; KLAEREN, 1996; RODRIGUES; CAMPOS; PEREIRA, 2013).

2.3.4 Componentes Fortemente Conexos

Para reduzir o tempo de execução da análise é utilizada a técnica de divisão e conquista a fim de dividir o problema original em sub-problemas. Para isso, é utilizada a técnica de encontrar as *CFC*, do grafo e reduzir cada *CFC* a um único vértice (RODRIGUES; CAMPOS; PEREIRA, 2013). Neste ponto, temos um grafo dirigido acíclico, e então esse grafo é ordenado topologicamente. Essa etapas são aplicadas em ordem topológica para cada componente fortemente conexo do grafo (RODRIGUES; CAMPOS; PEREIRA, 2013).

A Figura 12 mostra os componentes fortemente conexos do grafo da Figura 11.

Figura 12 — Componentes fortemente conexos do grafo da Figura 11



Fonte: (RODRIGUES; CAMPOS; PEREIRA, 2013, p. 3)

2.3.5 Alargando o Intervalo das Variáveis

Este passo tem o propósito de determinar como os valores das variáveis do programa mudam ao longo do tempo. Por exemplo, as variáveis às quais somente são atribuídos valores positivos apenas crescem em seu tempo de execução, enquanto as variáveis às quais somente são atribuídos valores negativos apenas decrescem e em algumas situações existem variáveis que crescem em ambas direções (RODRIGUES; CAMPOS; PEREIRA, 2013). Neste passo não são considerados ainda os intervalos impostos pelos condicionais.

Figura 13 — Equação para alargamento de intervalo

$$\frac{I[Y] = [\bot, \bot]}{I[Y] \leftarrow e(Y)} \qquad \frac{e(Y)_\downarrow < I[Y]_\downarrow \qquad e(Y)_\uparrow > I[Y]_\uparrow}{I[Y] \leftarrow [-\infty, +\infty]}$$

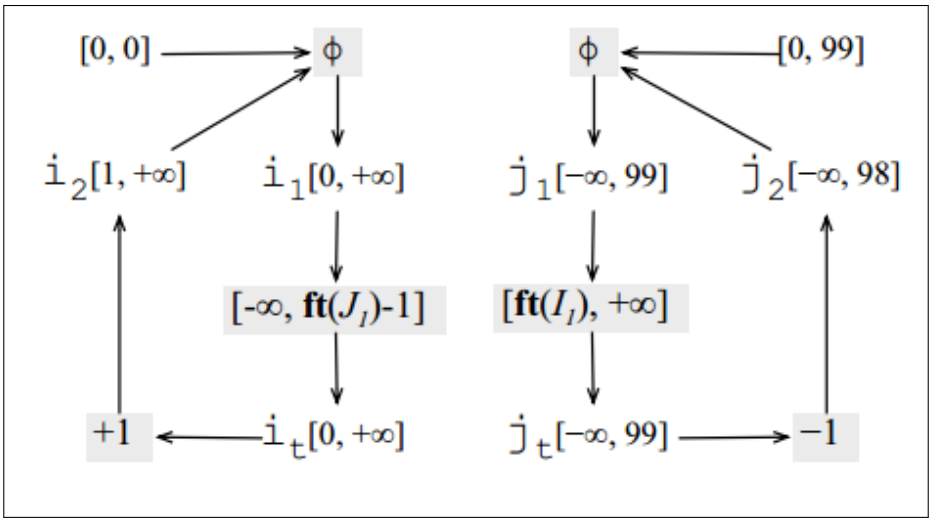
$$\frac{e(Y)_\downarrow < I[Y]_\downarrow}{I[Y] \leftarrow [-\infty, I[Y]_\uparrow]} \qquad \frac{e(Y)_\uparrow > I[Y]_\uparrow}{I[Y] \leftarrow [I[Y]_\downarrow, +\infty]}$$

Fonte: (RODRIGUES; CAMPOS; PEREIRA, 2013, p. 3)

A Figura 13 apresenta a equação para alargamento dos intervalos. Cada variável *Y* é associada com seu atual intervalo *I[Y]*. A semântica abstrata desta variável é dada por uma função de avaliação chamada *e(Y)* (RODRIGUES; CAMPOS; PEREIRA, 2013). Por exemplo, se *e(Y)* for diminuindo o intervalo inferior de *Y*, então é atribuído para esse intervalo inferior $-\infty$.

Para simplificar o exemplo, é utilizado somente o último *CFC* da Figura 11. Após aplicar a equação iterativamente no grafo o resultado é apresentado na Figura 14.

Figura 14 — Último *CFC* após o alargamento de intervalos



Fonte: (RODRIGUES; CAMPOS; PEREIRA, 2013, p. 4)

2.3.6 Corrigindo as Intersecções

Quando os intervalos das variáveis são alargados são encontradas variáveis que tem limites fixos. Com isso podemos utilizar esses limites para corrigir as intersecções que dependem de valores futuros, *ft(V)* (GOUGH; KLAEREN, 1996).

Figura 15 — Equação para resolver valores futuros

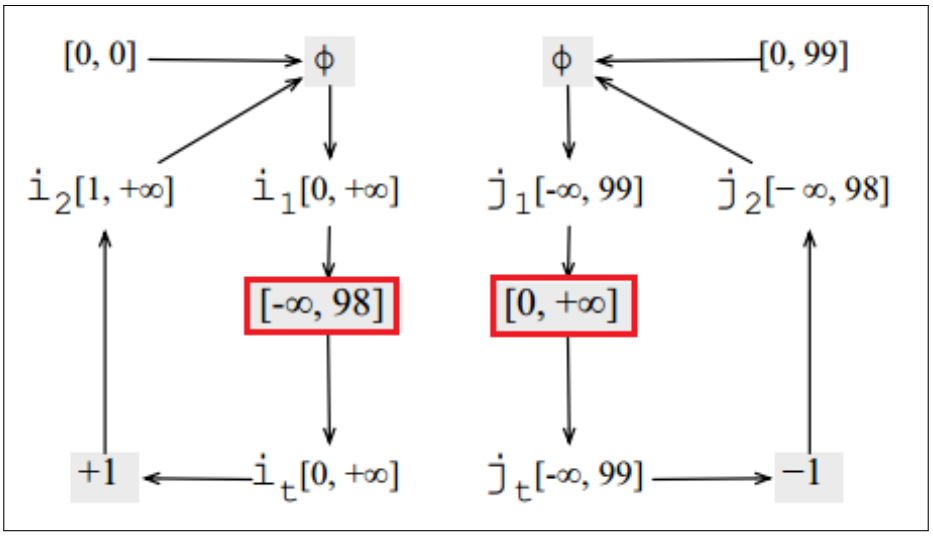
$$\frac{Y = X \bigcap [l, ft(V) + c] \qquad I[V] \uparrow = u}{Y = X \bigcap [l, u + c]}$$

$$\frac{Y = X \bigcap [ft(V) + c, u] \qquad I[V] \downarrow = l}{Y = X \bigcap [l + c, u]}$$

Fonte: (RODRIGUES; CAMPOS; PEREIRA, 2013, p. 4)

Na Figura 15 é apresentada a equação para resolver valores futuros, com a aplicação dela é possível substituir os rótulos *ft[V]* por valores obtidos no alargamento de variáveis. Seguindo o exemplo do último *CFC* o resultado é apresentado na Figura 16.

Figura 16 — Último *CFC* após corrigir intersecções



Fonte: (RODRIGUES; CAMPOS; PEREIRA, 2013, p. 4)

2.3.7 Estreitando o Intervalo das Variáveis

A etapa de alargamento do intervalo das variáveis atribui valores conservativos ao intervalo de cada variável. Esse estreitamento é guiado pelas intersecções que foram derivadas dos testes condicionais (RODRIGUES; CAMPOS; PEREIRA, 2013). Na Figura 17 é apresentada a equação para estreitar os valores das variáveis.

Figura 17 — Equação para estreitamento do intervalo

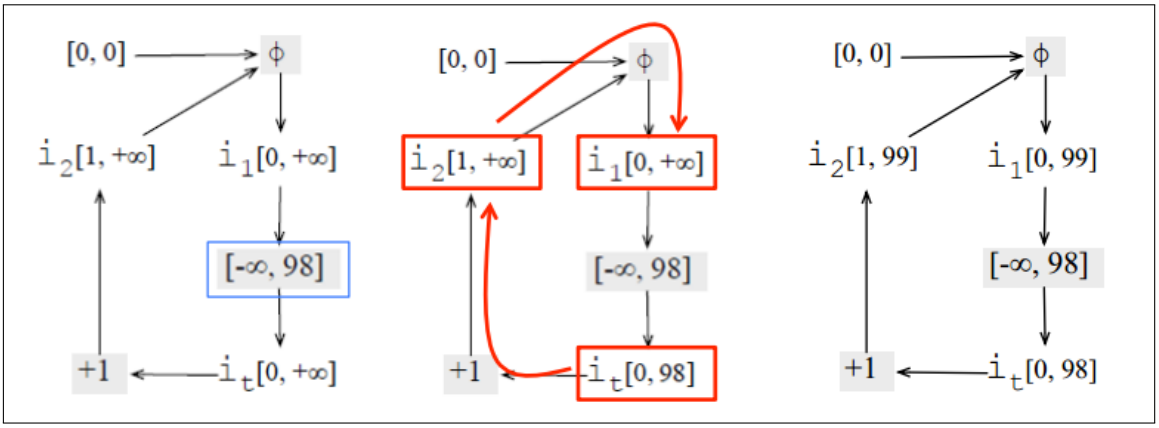
$$\frac{I[Y]_{\downarrow} = -\infty \qquad e(Y)_{\downarrow} > -\infty}{I[Y] \leftarrow [e(Y)_{\downarrow}, I[Y]_{\uparrow}]} \qquad \frac{I[Y]_{\downarrow} > e(Y)_{\downarrow}}{I[Y] \leftarrow [e(Y)_{\downarrow}, I[Y]_{\uparrow}]}$$

$$\frac{I[Y]_{\uparrow} = +\infty \qquad e(Y)_{\uparrow} < +\infty}{I[Y] \leftarrow [I[Y]_{\downarrow}, e(Y)_{\uparrow}]} \qquad \frac{I[Y]_{\uparrow} < e(Y)_{\uparrow}}{I[Y] \leftarrow [I[Y]_{\downarrow}, e(Y)_{\uparrow}]}$$

Fonte: (RODRIGUES; CAMPOS; PEREIRA, 2013, p. 4)

Continuando com o exemplo do último *CFC* após corrigir as intersecções mostrados na Figura 16. Sabemos que o valor de *i* é menor que *99*, esse número foi descoberto na etapa de corrigir as intersecções. Esse valor é propagado por cada caminho do grafo que é influenciado por *i* utilizando a equação de estreitamento do intervalo da Figura 17.

Figura 18 — Exemplo de propagação do valor de *i*

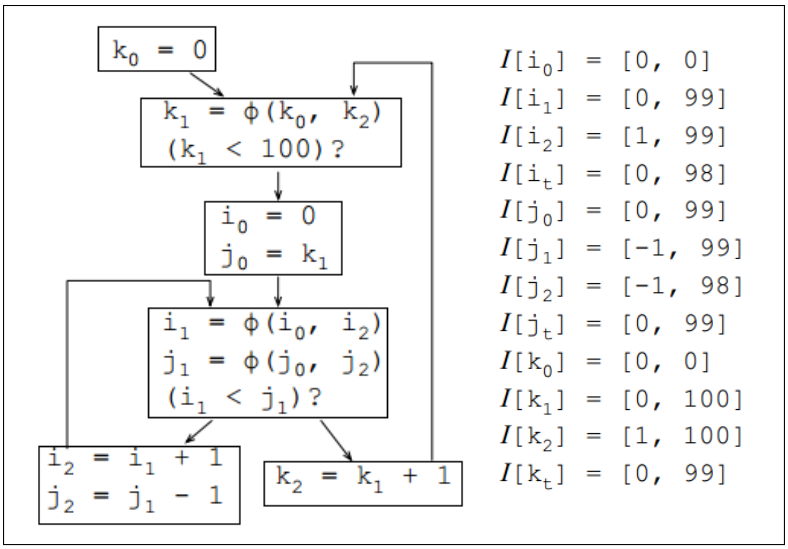


Fonte: (RODRIGUES; CAMPOS; PEREIRA, 2013, p. 5)

A representação de como a variável *i* se propagou em cada caminho é mostrada na Figura 18. É possível perceber que com esse último passo é obtido um resultado dos intervalos de valores da variável *i* (RODRIGUES; CAMPOS; PEREIRA, 2013).

Na Figura 19 é apresentado o resultado de intervalo de cada variável do algoritmo após executar todas estas etapas (Alargando o Intervalo das Variáveis, Corrigindo as Intersecções e Estreitando o Intervalo das Variáveis) em todos os *CFC*.

Figura 19 — Resultado da análise de intervalo de valores



Fonte: (RODRIGUES; CAMPOS; PEREIRA, 2013, p. 3)

Com essas informações é possível identificar se existe acesso a uma posição de um vetor que não existe ou também identificar expressões condicionais que resultariam sempre o mesmo resultado (GOUGH; KLAEREN, 1996).

3 ESCOLHA DA TÉCNICA PARA CADA ERRO

Neste capítulo são descritas as técnicas que são utilizadas para cada erro. Em cada Seção é descrito o erro, como funciona a técnica e um exemplo de código com o respectivo erro.

Uma das técnicas utilizadas é o *SSA*. Nela é utilizada o algoritmo da versão *podada* pois a análise de vida de variáveis é reaproveitada da análise de código morto. O *e-SSA* irá utilizar a estrutura criada pelo *SSA* e será utilizado para análise de intervalo de valores.

3.1 VARIÁVEIS NÃO INICIALIZADAS E FUNÇÕES QUE NÃO DEFINEM O VALOR DE RETORNO

O erro de *variáveis não inicializadas* acontece quando em qualquer caminho do algoritmo uma variável é *usada* sem ter sido pelo menos definida uma vez ao longo do caminho. Para fazer esta detecção foi escolhida a técnica de *definições incidentes* que foi apresentada na Seção 2.1.1 pois apresenta a incidência de cada variável em pontos do algoritmo (COOPER; HARVEY; KENNEDY, 2004). Ao executar o algoritmo de *definições incidentes* teremos as definições que incidem em cada bloco básico. Para detectar as *variáveis não inicializadas* será executado um teste em cada caminho possível separadamente. Para cada variável *usada* (utilizada do lado direito de uma atribuição ou em um teste) deverá haver pelo menos uma *incidência* na saída de algum dos blocos anteriores.

Por exemplo no Trecho de Código 6 assumimos que a variável *x* nunca foi definida e a variável *a* e *b* foram definidas pelo menos uma vez em algum momento até o ponto inicial do exemplo. Se o teste condicional *a = b* resultar em *falso* a variável *x* não será definida em nenhum ponto do caminho e ao ser *usada* na atribuição *y = x * b* ocorrerá um erro pois nunca houve uma incidência

da definição de *x* neste caminho.

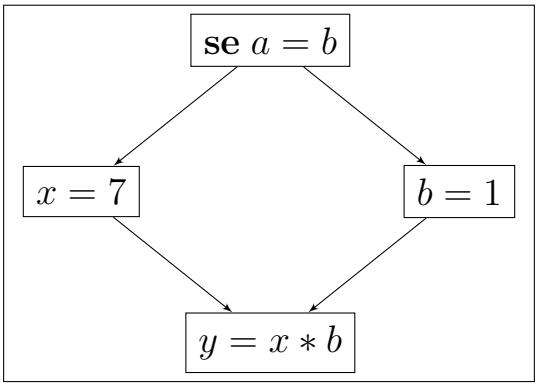
Trecho de Código 6 — Exemplo de algoritmo com variável não inicializada

```
1     //Assumindo que a variável x nunca foi definida e a e b foram definidas pelo
          menos em um momento até este ponto
2     se a = b então
3         x = 7;
4     senão
5         b = 1;
6     fimse
7
8     y = x * b;
```

Fonte: Próprio autor

Pelo grafo de fluxo de controle é mais visível identificar os caminhos que o algoritmo pode tomar. A Figura 20 mostra o grafo de fluxo de controle do Trecho de Código 6. É possível ver que a aresta do lado direito não tem nenhuma definição de *x*.

Figura 20 — Grafo de Fluxo de Controle do Trecho de Código 6



Fonte: Próprio autor

O erro de *funções que não definem o valor de retorno* acontece quando em algum caminho a partir do início da função o valor de retorno da função não é definido. Este problema é similar ao das *variáveis não inicializadas*, pois a variável de retorno da função deve ter pelo menos alguma incidência em qualquer caminho da função. Se existir mais de uma instrução *retorne* na função cada uma tem que ter incidência para o retorno. Portanto, é possível resolver com o mesmo algoritmo descrito anteriormente.

Trecho de Código 7 — Exemplo de algoritmo com função que não define valor de retorno

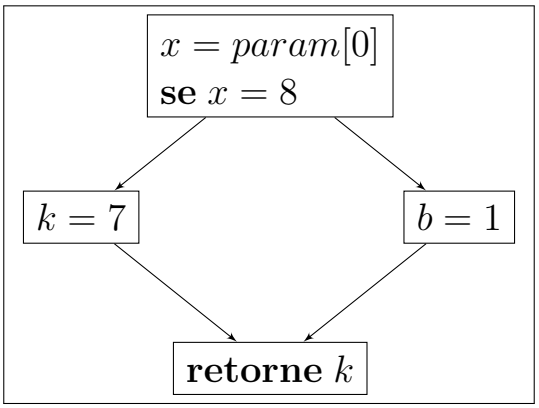
```
1     Funcao exemplo(x : inteiro): inteiro
2     var k,b : inteiro
```

```
 3      inicio
 4          se x = 8 então
 5              k = 7;
 6          senão
 7              b = 1;
 8          fimse
 9
10          retorne k;
11      fimfuncao
```

Fonte: Próprio autor

No grafo de fluxo da Figura 21 é possível visualizar com mais clareza que o erro de *função que não define o valor de retorno* é similar ao erro de *variáveis não inicializadas*.

Figura 21 — Grafo de Fluxo de Controle do Trecho de Código 7



Fonte: Próprio autor

3.2 CÓDIGO MORTO

Código morto podem ser um conjunto de uma ou mais instruções que foram *executadas* ao longo do algoritmo mas nunca foram *usadas* após a sua *definição*. Para a detecção deste erro foi escolhida a técnica de *análise de vida de variáveis* descrita na Seção 2.1.2 pois informa a vida das variáveis em cada ponto do algoritmo (AHO; ULLMAN; SETHI, 2008). Inicialmente é feita a análise de vida em cada bloco básico, após é feito uma análise de baixo para cima para cada atribuição feita, chamada de análise de uso subsequente, com esta análise é possível saber se a variável não é utilizada naquele ponto em diante (AHO; ULLMAN; SETHI, 2008). Cada atribuição com estas propriedades é marcada como *código morto*.

Abaixo no Trecho de Código 8 é apresentado um exemplo onde as

variáveis *k* e *h* foram *definidas* mas não foram *utilizadas* depois no algoritmo.

No Trecho de Código 8 temos um exemplo onde a variável *h* foi *definida* mas não foi *utilizada*, portanto esta instrução é marcada como código morto. A instrução que define a variável *k* também é marcada como código morto pois é somente *usada* na atribuição de *h* que foi marcada como código morto.

Trecho de Código 8 — Exemplo de algoritmo com variáveis não utilizadas

```
1     a = 3;
2     x = 10;
3     y = 8;
4     z = x + y;
5     j = a + x + y;
6     k = y + 10;
7     h = k + 15;
8     procedimento_teste(z,j);
```

Fonte: Próprio autor

3.3 EXPRESSÃO CONSTANTE, CÓDIGO INACESSÍVEL E LOOP

O erro de *expressão constante*, *código inacessível* e *loop* são erros que podem ser encontrados da mesma forma, pois um *código inacessível* ocorre quando existe um condicional que nunca irá passar por uma das *arestas* portanto é uma *expressão constante* e o *loop* é um laço que contem uma *expressão constante* que faz com que o laço nunca encontre a saída. Para detectar estes erros foi escolhida a forma *e-SSA* com o algoritmo de *análise de intervalo de valores* mas adicionando um passo a mais. Ao terminar o cálculo de intervalo de valores o algoritmo irá utilizar os dados dos intervalos para detectar se existem as *expressões constantes* com isso detectando código inacessível e loop (STAIGER et al., 2007; XU, 2001).

O Trecho de Código 9 mostra um exemplo onde há código inacessível, expressão constante e um loop. A expressão *j = 15* vai ser sempre *verdadeira* pois a variável *j* é composta pela multiplicação de *a* e *b* que são *3* e *5* respectivamente, resultando no valor *15*, e assim temos um código inacessível que é o caminho da *aresta-falsa* do teste. No loop a expressão $i <> a$ será sempre verdadeira pois o *i* sempre será *3* e diferente de *a* que possui valor *5*.

Trecho de Código 9 — Exemplo de algoritmo com expressão constante, código inacessível e loop

```
1 algoritmo
2 var
```

```
 3     a,b,j,k,i : inteiro
 4 inicio
 5     a = 5;
 6     b = 3;
 7     j = a * b;
 8
 9     se j = 15 então
10         i = 3;
11     senão
12         i = 1;
13     fimse
14
15     enquanto i <> a faça
16     inicio
17         k = k + 1;
18         j = i + b;
19     fimenquanto
20
21 fimalgoritmo
```

Fonte: Próprio autor

3.4 ÍNDICES DE VETORES

O erro de *índices de vetores* ocorre quando é feita uma tentativa de acesso a uma posição do vetor que não existe. Este erro é detectado utilizando a forma *e-SSA* com o algoritmo de *análise de intervalo de valores*, da mesma forma da Seção 3.3. Ao propagar os intervalos de valores o algoritmo testará o índice do vetor com a declaração dele, se o índice estiver fora do declarado será identificado como erro (BIRCH; ENGELEN; GALLIVAN, 2004).

No Trecho de Código 10 temos um exemplo de erro de *índices de vetores*. A variável *vet* é um vetor com posições de intervalo de *1* a *10*, no laço *repita* o *i* é incrementando de *uma* unidade a cada iteração e o teste de saída é $atei > 10$. Portanto na penúltima iteração o *i* estará valendo *10* mas irá para o início do laço novamente pois *10* não é maior que **10**, com isso *i* será incrementado para o valor *11* e ocorrerá o erro.

Trecho de Código 10 — Exemplo de algoritmo com erro de índice de vetores

```
 1 algoritmo
 2 var
 3     vet : vetor [1..10] de inteiro
 4     i   : inteiro
 5 inicio
 6     i = 0;
 7     //na última iteração o i vai ser incrementado para 11
 8     //assim ocorrendo um erro
 9     repita
10         i = i +1;
```

```
11          vet[i] = i;
12      ate i > 10
13
14 fimlagoritmo
```

Fonte: Próprio autor

4 IMPLEMENTAÇÃO

O analisador estático de código foi desenvolvido em *Java SE* (*Standard Editon*) utilizando a versão 1.8.0_171 *JDK* (*Java Development Kit*) que é mantido pela ORACLE. Maiores informações podem ser encontradas na página de apresentação (ORACLE, 2018) e na documentação da versão 8 (ORACLE, 2017).

Para gerenciamento do projeto foi utilizada a versão 3.5.2 do *Apache Maven*. Essa ferramenta gerencia o projeto, serve para fazer compilação, atualizar e manter dependências e tem integração com o *GitLab*. É mantido pela Apache e maiores informações sobre podem ser encontradas na página de apresentação (APACHE, 2018).

Para versionamento e gerenciamento de fontes foi utilizado o *Git* utilizando a versão 2.6.2 que é mantido por Linus Torvalds e Junio Hamano. Maiores informações podem ser encontradas na página de apresentação (TORVALDS, 2018).

Foi criado um projeto novo contendo somente o analisador estático de código para separar os projetos por responsabilidades. Desta forma o analisador estático de código pode ser incorporado em outro portal de algoritmos e também fica facilitada a evolução.

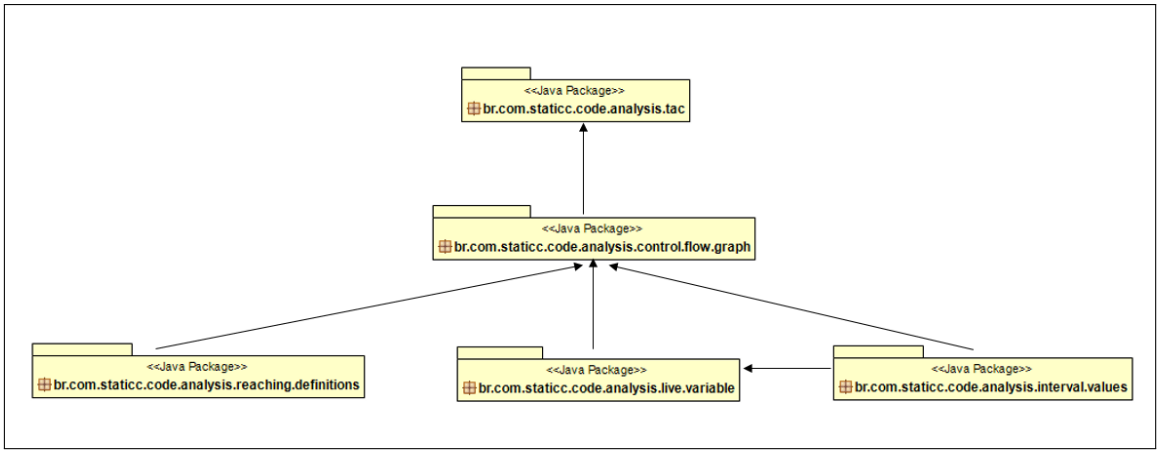
Durante o desenvolvimento deste trabalho, buscou-se implementar o analisador de forma que a evolução do mesmo seja garantida por testes unitários. Testes unitários foram desenvolvidos somente para classes com grande criticidade no analisador. Estas classes foram identificadas ao longo do desenvolvimento, pois necessitavam incremento de funcionalidades e precisavam manter características antes desenvolvidas, então foram criados testes unitários para cada comportamento que a classe necessitava manter. Desta forma a evolução do analisador foi produtiva e ao longo do desenvolvimento foram poucos momen-

tos que se identificou falhas das classes com testes unitários.

Todas as análises são *intra procedural*, ou seja, cada função, procedimento e código principal são processados separadamente.

O projeto é separado em 5 pacotes principais, cada um contém o nome que busca representar sua função na arquitetura. O pacote **TAC** (*br.com.staticc.code.analysis.tac*), responsável pela conversão para código de três endereços, **Control Flow Graph** (*br.com.staticc.code.analysis.control.flow.graph*), responsável por gerar o grafo de fluxo de controle, **Live Variable** (*br.com.staticc.code.analysis.live.variable*), responsável pela análise de vida de variáveis, **Reaching Definitions** (*br.com.staticc.code.analysis.live.reaching.definitions*), responsável pela análise de definições incidentes e **Interval Values** (*br.com.staticc.code.analysis.interval.values*), responsável pela análise de intervalo de valores. A Figura 22 apresenta o diagrama *UML* de pacotes que compõem o analisador, indicando também a relação de utilização entre eles.

Figura 22 — Pacotes principais



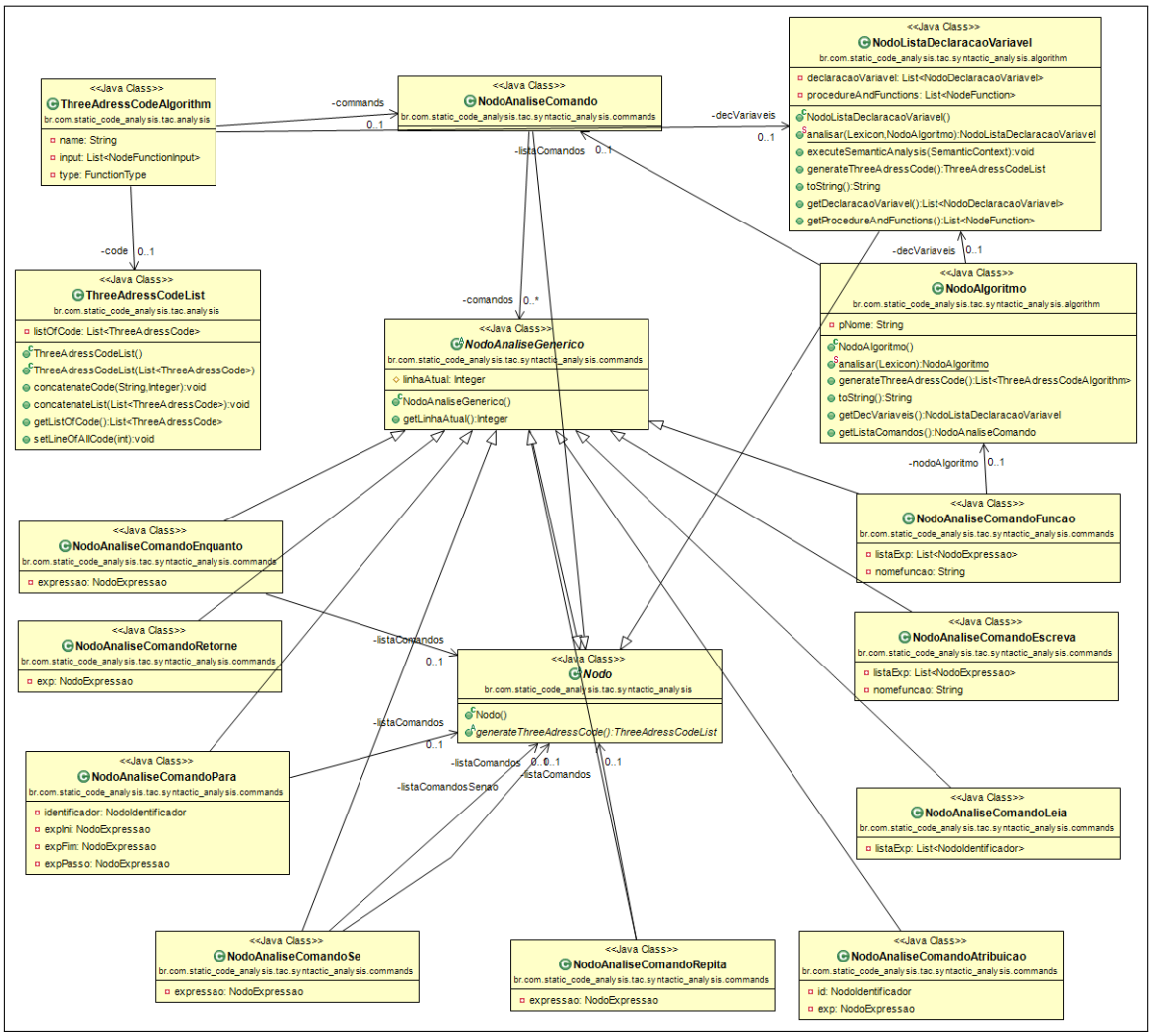
Fonte: Próprio autor

As próximas seções descrevem e ilustram cada um destes pacotes mostrando de forma sucinta as classes que os compõem. O código completo pode ser encontrado no *CD* em anexo a este trabalho.

4.1 CÓDIGO DE TRÊS ENDEREÇOS

Para o código ser analisado é necessário converte-lo para a representação intermediária chamada **Código de Três Endereços**. Este assunto foi abordado no Capítulo 2. A Figura 23 contém a composição das principais classes para conversão do código.

Figura 23 — Representação da conversão para Código de Três Endereços



Fonte: Próprio autor

Cada comando do algoritmo estende a classe abstrata **NodoAnalise-Generico** que estende a classe abstrata **Nodo** que contém a função para implementar a geração de código de três endereços.

O algoritmo é representado pela classe **NodoAlgoritmo** que contém o nome do algoritmo, declaração de variáveis e a lista de comandos. Esta classe contém a função *generateThreeAdressCode()* que é responsável por gerar uma lista com o objeto **ThreeAdressCodeAlgorithm**. Este objeto contém o nome do algoritmo, função ou procedimento, a declaração das variáveis, os parâmetros de entrada e o tipo do código, se é o principal, função ou procedimento. O Trecho de Código 11 apresenta como é gerado o código de três endereços.

Trecho de Código 11 — Geração do código de três endereços.

```
1 public List<ThreeAdressCodeAlgorithm> generateThreeAdressCode(){
2     List<ThreeAdressCodeAlgorithm> list = new ArrayList<>();
```

```
3
4     //get all functions/procedures
5     for (NodeFunction funcion : this.decVariaveis.getProcedureAndFunctions()) {
6         ThreeAdressCodeList generateThreeAdressCode = funcion.
              generateThreeAdressCode();
7
8         list.add(new ThreeAdressCodeAlgorithm(funcion.getName(), funcion.
              getDeclarations(), funcion.getInput() , funcion.getType(),
              generateThreeAdressCode, funcion.getCommands()));
9     }
10
11    //main code algorithm
12    ThreeAdressCodeAlgorithm algo = new ThreeAdressCodeAlgorithm(pNome,
          decVariaveis, FunctionType.MAIN, listaComandos.generateThreeAdressCode()
          , listaComandos);
13    algo.getCode().concatenateCode("STOP", null);
14    list.add(algo);
15
16    return list;
17    }
```

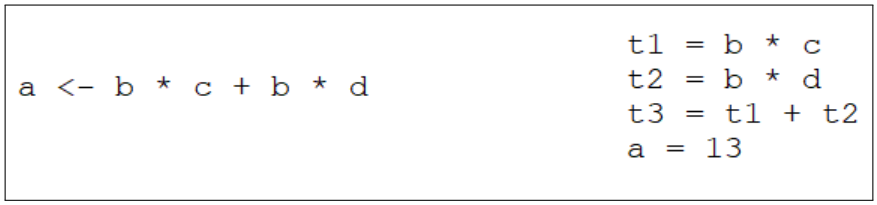
Fonte: Próprio autor

Note que as funções e procedimentos estão declaradas dentro da declaração de variáveis na linguagem *portugol*.

4.1.1 Exemplo de código gerado

No código de três endereços as operações se limitam no máximo a três operandos. É permitido uma instrução onde a variável de destino é atribuída por dois operandos e um operador aritmético entre eles. Quando uma operação envolve mais de três endereços são geradas variáveis temporárias com as operações quebradas em *n* operações, no final a variável temporária com resultado é atribuída à variável destino. A Figura 24 ilustra a conversão de uma instrução.

Figura 24 — Conversão de uma instrução



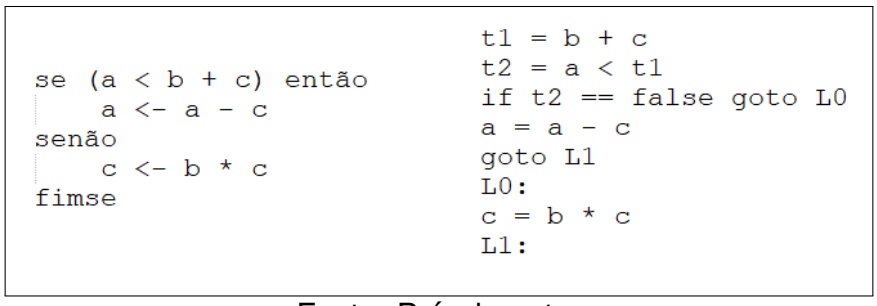
Fonte: Próprio autor

Neste trabalho foi feita uma otimização na conversão de instruções. Na Figura 24 antes do resultado chegar na variável **a** é atribuído o resultado na variável temporária **t3**, nesta otimização o resultado foi diretamente atribuído a

variável **a** sem passar pela variável temporária.

Como não existe laço de controle e condicional com *então* e *senão* no código de três endereços é possível alcançar o mesmo comportamento utilizando condicional com rótulos. O condicional contém o rótulo a ser acessado se a condição resultar em verdadeiro, se não for segue a próxima instrução. A Figura 25 ilustra a conversão de um condicional tradicional.

Figura 25 — Conversão de uma instrução

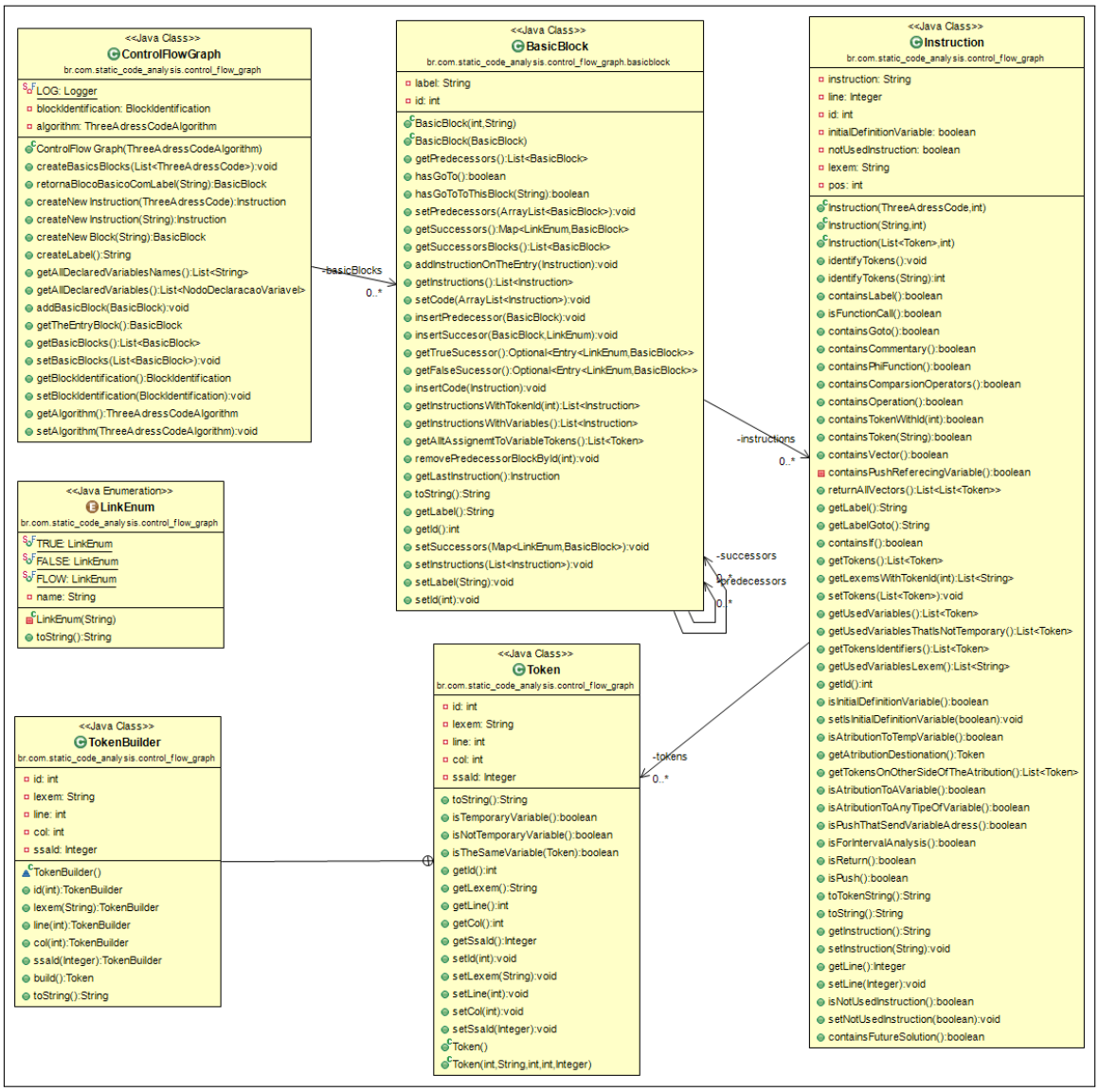


Fonte: Próprio autor

4.2 GRAFO DE FLUXO DE CONTROLE

Para facilitar análises estáticas é necessário construir um grafo de fluxo de controle. Todas as análises seguintes apresentadas utilizam o grafo de fluxo de controle para se orientar no algoritmo. A Figura 26 contém a composição das principais classes.

Figura 26 — Representação do Grafo de Fluxo de Controle



Fonte: Próprio autor

A principal classe é a **ControlFlowGraph**, ela faz o processamento e monta os blocos básicos e o grafo de fluxo de controle, o parâmetro de entrada é o **ThreeAdressCodeAlgorithm** gerado no conversor para código de três endereços. O bloco básico é representado pela classe **BasicBlock**. Neste objeto existe a lista de blocos predecessores e sucessores a ele. Na lista de blocos sucessores existe um enumerador chamado **LinkEnum** que define se aquele sucessor é resultante de um condicional verdadeiro, falso ou se é somente um fluxo normal sem condicional. A Figura 12 contém um Trecho de Código que descreve a geração de blocos básicos em condicionais.

Trecho de Código 12 — Geração de blocos básicos de condicionais

```
1 if (containsIf){
2     //insert the if
3     if (!containsLabel)
4     lastBlock.insertCode(inst);
5
6     //creates a new empty block that is the sucessor of the if
7     BasicBlock newBlock = createNewBlock();
8     newBlock.insertPredecessor(lastBlock);
9
10    addBasicBlock(newBlock);
11
12    //bind the if block with the sucessor false
13    lastBlock.insertSuccesor(newBlock, LinkEnum.FALSE);
14
15    //creates a new block with the destination label
16    BasicBlock labelDestino = retornaBlocoBasicoComLabel(inst.getLabelGoto());
17
18    //bind the if block with the sucessor true
19    lastBlock.insertSuccesor(labelDestino, LinkEnum.TRUE);
20    //bind the sucessor with the predecessor if
21    labelDestino.insertPredecessor(lastBlock);
22
23    if (lastBlock.getId() != labelDestino.getId())
24    addBasicBlock(labelDestino);
25    lastBlock = newBlock;
26 }
```

Fonte: Próprio autor

Cada bloco básico contém um campo que guarda o nome do rótulo (se o mesmo tiver), um identificador sequencial e uma lista de instruções, a instrução é representada pela classe **Instruction**. Dentro da classe **Instruction** há uma lista de símbolos, o símbolo é representado pela classe **Token**. Cada símbolo contém um identificador que lhe é atribuído no momento do processamento da instrução. A classe **TokenDictionary** contém o dicionário de símbolos, cada um com seu respectivo identificador, este identificador é sequencial. A Figura 13 contém um Trecho de Código que descreve a geração de identificador para cada tipo de símbolo.

Trecho de Código 13 — Geração de identificador para os símbolos

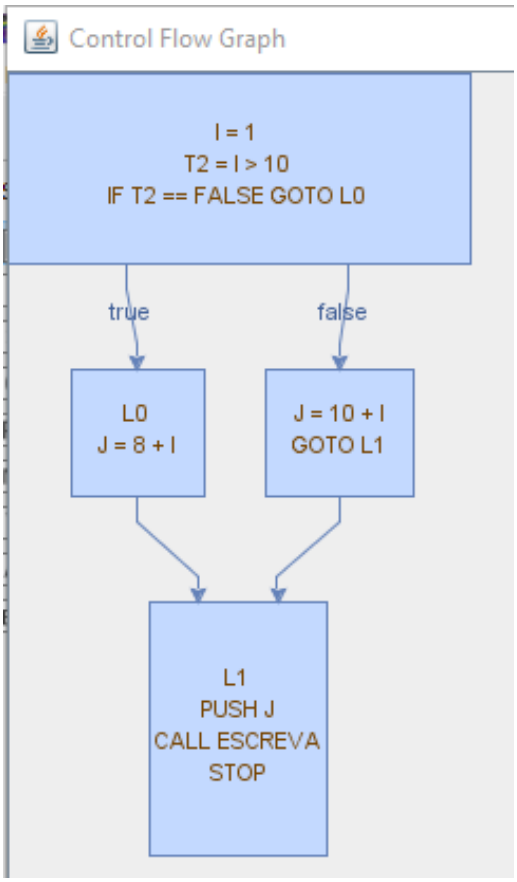
```
1 tokens = new HashMap<String, Integer>();
2 tokens.put(TK_IF, id++);
3 tokens.put(TK_CALL, id++);
4 tokens.put(TK_GOTO, id++);
5 tokens.put(TK_PUSH, id++);
6 tokens.put(TK_POP, id++);
7 tokens.put(TK_ATRIBUTION, id++);
```

Fonte: Próprio autor

Para representar graficamente o grafo de fluxo de controle foi criada uma classe chamada **Rendering**. Esta classe gera um **JFrame** com o grafo, a Figura 27 contém o resultado da geração de grafo de um código simples.

Figura 27 — Grafo de Fluxo de Controle gerado

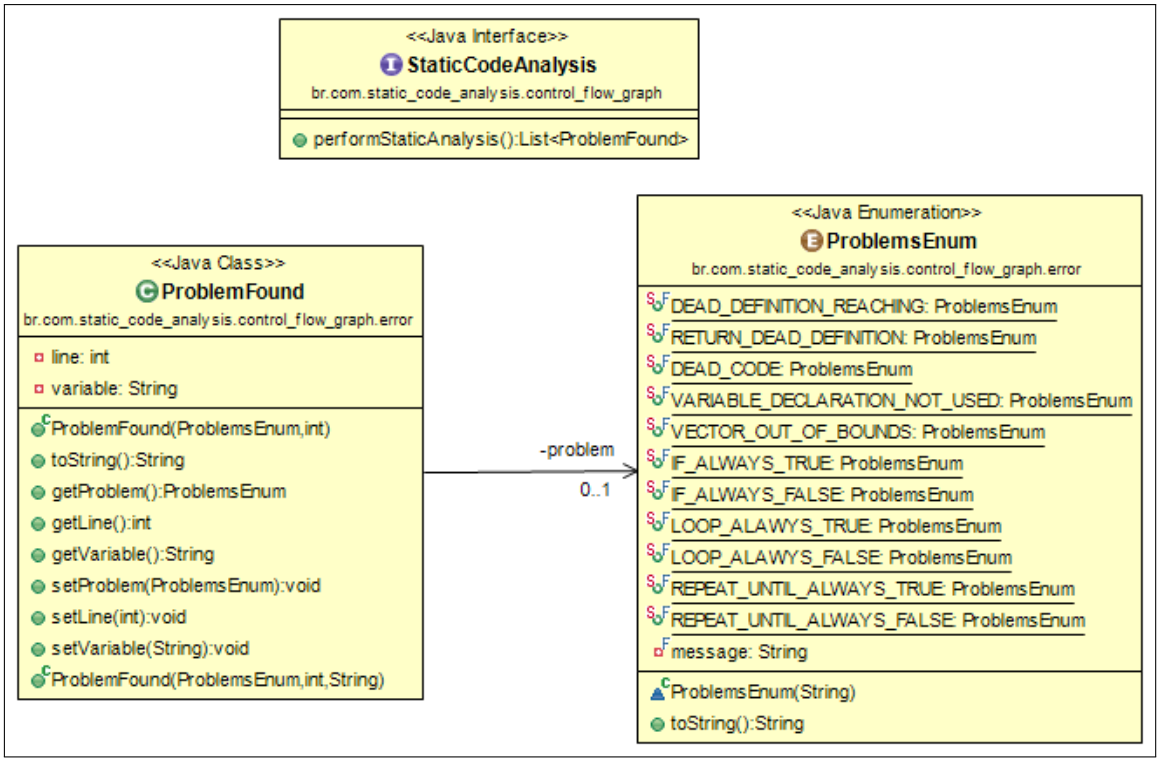
```
algoritmo "algo"
var
    n, i, j : inteiro
inicio
    i ← 1
    se i > 10 entao
        j ← 10 + i
    senao
        j ← 8 + i
    fimse
    escreva(j)
fimalgoritmo
```



Fonte: Próprio autor

4.2.1 Estrutura para análise estática

Figura 28 — Representação das classes para análise estática



Fonte: Próprio autor

A Figura 28 representa a interface **StaticCodeAnalysis** que contém um método chamado **performStaticAnalysis()**. Este método retorna uma lista de problemas encontrados, o problema é representado pela classe **ProblemFound** que contém um enumerador chamado **ProblemsEnum** onde todos os tipos de erros estão mapeados. A classe **ProblemFound** tem um método **toString()** que preenche os dados do erro na mensagem, como por exemplo a linha onde o erro ocorreu e a variável envolvida. O Trecho de Código 14 contém os enumeradores definidos.

Trecho de Código 14 — Enumerador com os problemas mapeados

```
1 DEAD_DEFINITION_REACHING("Variável #variable# na linha #line# pode ser usada sem
       ter sido inicializada."),
2 RETURN_DEAD_DEFINITION("Retorno da função na linha #line# pode retornar uma
      variável não definida."),
3 DEAD_CODE("Código morto na linha #line#."),
4 VARIABLE_DECLARATION_NOT_USED("Variável #variable# não está sendo utilizada."),
5 VECTOR_OUT_OF_BOUNDS("Vetor #variable# está sendo acessado fora de seus limites
      na linha #line#."),
6 IF_ALWAYS_TRUE("Condicional na linha #line# sempre resultara em verdadeiro."),
7 IF_ALWAYS_FALSE("Condicional na linha #line# sempre resultara em falso."),
8 LOOP_ALAWYS_TRUE("Laço na linha #line# entrará em loop infinito."),
9 LOOP_ALAWYS_FALSE("Laço na linha #line# nunca será executado."),
```

```
10 REPEAT_UNTIL_ALWAYS_TRUE("Repita na linha #line# será executado somente uma vez.
       "),
11 REPEAT_UNTIL_ALWAYS_FALSE("Repita na linha #line# entrada em loop infinito.");
```

Fonte: Próprio autor

4.3 ANÁLISE DE DEFINIÇÕES INCIDENTES

Esta análise utiliza o grafo de fluxo de controle para percorrer o código e encontrar quais definições incidem em um determinado ponto no código. A Figura 29 contém a composição das principais classes para a análise de definições incidentes.

Figura 29 — Representação da Análise de Definições Incidentes

Fonte: Próprio autor

A classe **ReachingDefinitionsAnalysis** implementa a interface **StaticCodeAnalysis**, deste modo a classe implementa o método **performStaticAnalysis()** que é responsável por executar a análise. No construtor recebe o objeto **ControlFlowGraph** que é o grafo de fluxo de controle. Para guardar as definições que incidem na saída e entrada de um bloco existe a classe **BlockDefinition** que estende a classe **BasicBlock**.

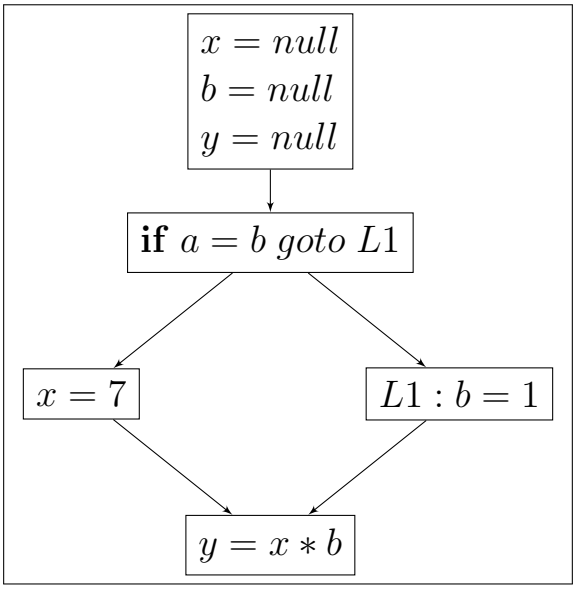
4.3.1 Identificação de erros

A análise de definições incidentes é utilizada para identificar variáveis não inicializadas e funções que não definem valor de retorno. A descrição deste

erro está na Seção 3.1.

Para encontrar se uma variável está sendo utilizada sem ter sido inicializada é necessário inserir um bloco básico na entrada do algoritmo e inserir instruções como se fosse a declaração das variáveis em estado nulo. Se alguma destas instruções incidirem em uso, foi encontrado um erro. A Figura 30 contém um bloco básico na entrada com as variáveis em estado nulo, a variável **a** e **b** são parâmetros de entrada então não entram neste bloco.

Figura 30 — Exemplo de um bloco básico de entrada



Fonte: Próprio autor

Após adicionar este bloco com as inicializações o algoritmo de definições incidentes é executado. Quando o algoritmo termina cada bloco contém as definições que incidem nele mesmo.

Para encontrar erros é executado um laço em cada bloco básico verificando em cada instrução quais definições incidem em cada variável utilizada. O Trecho de Código 15 contém a análise feita em cada bloco para encontrar variáveis não inicializadas ou funções que não definem valor de retorno.

Trecho de Código 15 — Procurando por variáveis não inicializadas ou funções que não definem valor de retorno

```
1 List<ProblemFound> problems = new ArrayList<>();
2 for (BlockDefinition block : blockDefinitions) {
3     List<Instruction> in = block.getIn();
4     for (Instruction inst : block.getInstructions()) {
5         List<Token> usedVariables = inst.getUsedVariables();
6
7         if (!usedVariables.isEmpty()) {
```

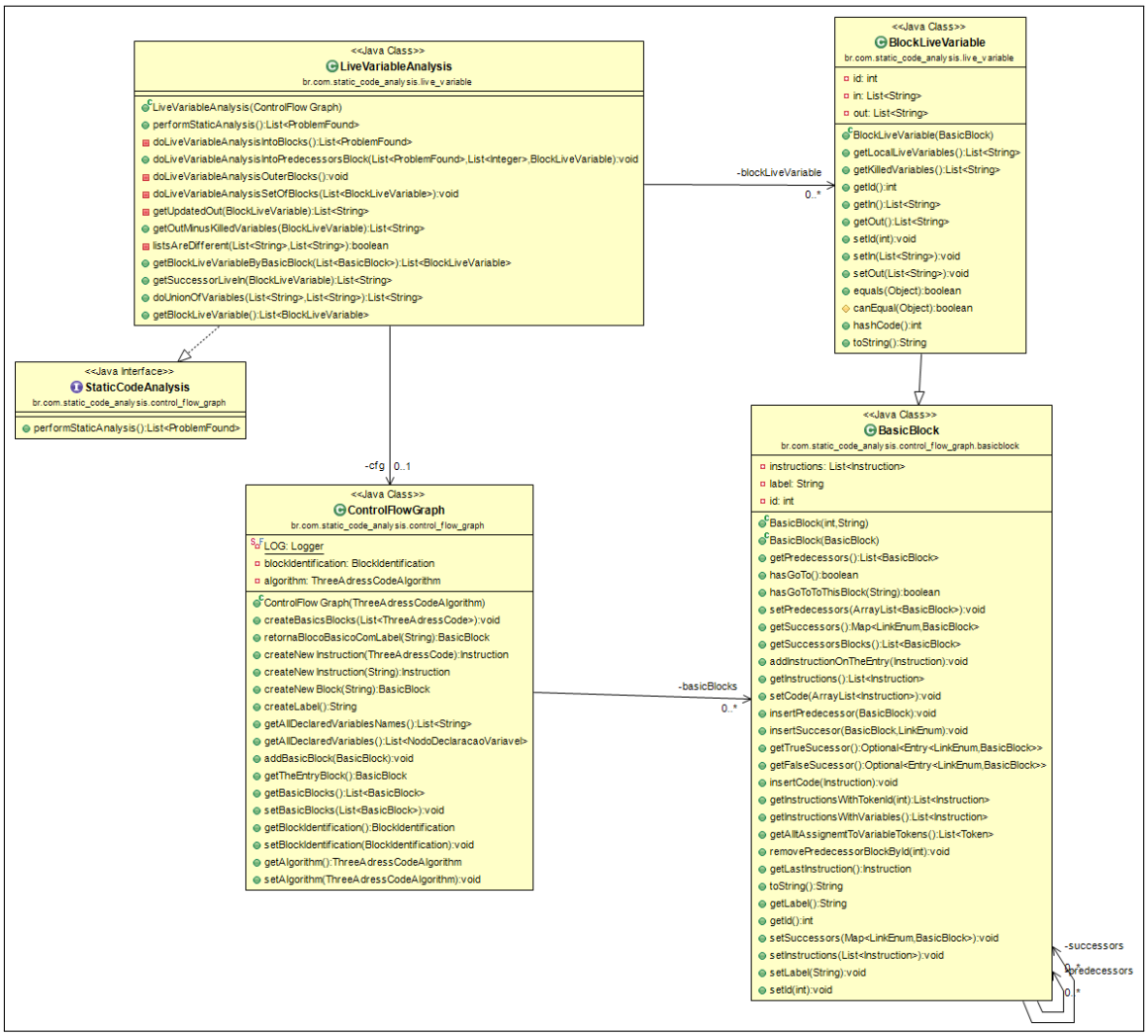
```
 8              for (Token used : usedVariables) {
 9                  if (in.stream().anyMatch(f -> f.getTokens().get(0).getLexem().
                        equals(used.getLexem())
10                  && f.isInitialDefinitionVariable()))
11                  problems.add(new ProblemFound(block.isInstructionUsedOnReturn(
                        inst) ? ProblemsEnum.RETURN_DEAD_DEFINITION : ProblemsEnum.
                        DEAD_DEFINITION_REACHING, inst.getLine(), used.getLexem()));
12              }
13          }
14
15          if(inst.isAtributionToAnyTipeOfVariable()) {
16              Token generatedVariable = inst.getAtributionDestionation();
17              for (Iterator<Instruction> iterator = in.iterator(); iterator.
                    hasNext();) {
18                  Instruction ins = iterator.next();
19                  if (ins.getTokens().get(0).getLexem().equals(generatedVariable.
                        getLexem()))
20                  iterator.remove();
21              }
22              in.add(inst);
23          }
24      }
25  }
26  return problems;
```

Fonte: Próprio autor

4.4 ANÁLISE DE VIDA DE VARIÁVEIS

Na análise de vida de variáveis os blocos básicos são percorridos pelo grafo de fluxo de controle para encontrar quais variáveis estão vivas na entrada e na saída de cada bloco básico. A Figura 31 contém a composição das principais classes.

Figura 31 — Representação da Análise de Vida de Variáveis



Fonte: Próprio autor

A classe **LiveVariableAnalysis** implementa a interface **StaticCodeAnalysis**, deste modo a classe implementa o método **performStaticAnalysis()** que é responsável por executar a análise. No construtor recebe o objeto **ControlFlowGraph** que é o grafo de fluxo de controle. Para guardar as variáveis que estão vivas na saída e entrada de um bloco existe a classe **BlockLiveVariable** que estende a classe **BasicBlock**.

4.4.1 Identificação de erros

A análise de definições incidentes é utilizada para identificar código morto. Uma instrução ou mais é marcada como código morto se sua definição não for usada ao longo do código. A descrição deste erro está na Seção 3.2.

A execução do algoritmo de análise de vida de variáveis faz com que cada bloco contenha as variáveis vivas na entrada e na saída. Para encontrar código morto é feita uma análise de trás para frente tanto no grafo de fluxo de controle quanto nas instruções no bloco básico. Então se inicia pelo último bloco básico e se executa pelos predecessores iterativamente. Quando uma variável é definida e não está viva é encontrado o código morto, e esta instrução é marcada como não utilizada. Após a análise em um bloco é propagado aos outros blocos o resultado das variáveis vivas. O Trecho de Código 15 contém a análise feita em cada bloco para encontrar código morto.

Trecho de Código 16 — Procurando por código morto

```
1 //if the block was visited don't need to do the analisys again
2 if(visitedBlocks.stream().anyMatch(a -> a == block.getId()))
3     return;
4 visitedBlocks.add(block.getId());
5
6 List<String> usedVariables = new ArrayList<>();
7 usedVariables.addAll(block.getOut());
8 List<Instruction> instructions = block.getInstructions();
9 for(int i = instructions.size() - 1; i >= 0; i--) {
10     Instruction ins = instructions.get(i);
11     boolean variableNotUsed = false;
12     if (ins.isAtributionToAnyTipeOfVariable()) {
13         String lexem = ins.getAtributionDestionation().getLexem();
14
15         if(!usedVariables.stream().anyMatch(var -> var.equals(lexem))) {
16             if(!ins.isAtributionToTempVariable())
17                 problems.add(new ProblemFound(ProblemsEnum.DEAD_CODE, ins.
                       getLine(), lexem));
18         variableNotUsed = true;
19         ins.setNotUsedInstruction(true);
20     }
21 }
22
23 //add all used variables
24 if(!variableNotUsed)
25     usedVariables.addAll(ins.getUsedVariablesLexem());
26 }
27
28 //update in live variable
29 block.setIn(this.getUpdatedOut(block));
30
31
32 //update sucessors live variable
33 List<BlockLiveVariable> blockDefinitionByBasicBlock = this.
       getBlockLiveVariableByBasicBlock(block.getPredecessors());
34 this.doLiveVariableAnalysisSetOfBlocks(blockDefinitionByBasicBlock);
35
36 //do in predecessors blocks
37 for (BlockLiveVariable predecessor : blockDefinitionByBasicBlock)
38     doLiveVariableAnalysisIntoPredecessorsBlock(problems, visitedBlocks,
           predecessor);
```

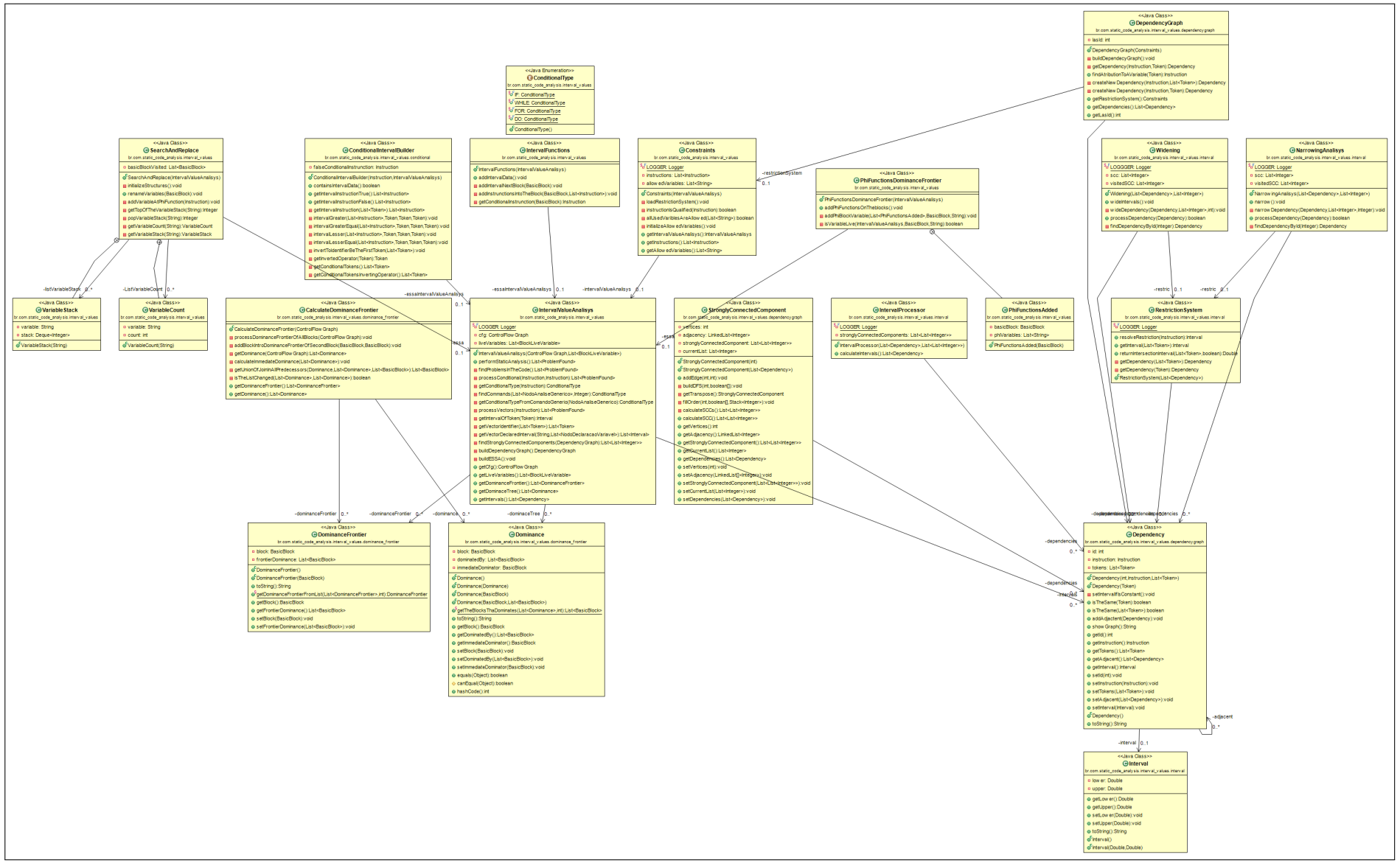
Fonte: Próprio autor

4.5 ANÁLISE DE INTERVALO DE VALORES

A análise de intervalo de valores identifica os intervalos que as variáveis assumem em um algoritmo. Este trabalho implementa a versão *e-SSA*, portanto cada atribuição de uma variável é única, assim é possível descobrir o intervalo em cada ponto do código. A Figura 32 representa as principais classes da análise de intervalo de valores.

A classe de entrada é a **IntervalValueAnalisys** que implementa a interface **StaticCodeAnalysis**. Ela recebe no construtor o grafo de fluxo de controle e a lista com as variáveis vivas em cada bloco. Existem cinco etapas na análise de intervalo de valores: a conversão do código para a representação *e-SSA*, montagem do grafo de dependências, encontrar os componentes fortemente conexos, processar os intervalos em cada componente fortemente conexo e por fim a identificação de erros com os intervalos encontrados. O Trecho de Código 17 contém o método que executa as cinco etapas citadas anteriormente.

Figura 32 — Representação da Análise de Intervalo de Valores



Fonte: Próprio autor

Trecho de Código 17 — Método principal da classe **IntervalValueAnalisys**

```
1 public List<ProblemFound> performStaticAnalysis() {
2     //build the extended static single assigment
3     buildESSA();
4
5     //build the dependency graph with the constraints
6     DependencyGraph dependencyGraph = buildDependencyGraph();
7
8     //find the strongly connected components
9     List<List<Integer>> findStronglyConnectedComponents =
          findStronglyConnectedComponents(dependencyGraph);
10
11    //Process intervals
12    IntervalProcessor intervalProcessor = new IntervalProcessor(dependencyGraph.
          getDependencies(), findStronglyConnectedComponents);
13    intervals = intervalProcessor.calculateIntervals();
14
15    return findProblemsInTheCode();
16 }
```

Fonte: Próprio autor

As próximas Seções descrevem estas cinco etapas e a identificação de erros.

4.5.1 Conversão do código para *e-SSA*

A primeira etapa é calcular a fronteira de dominância. A classe **CalculateDominanceFrontier** recebe no construtor o grafo de fluxo de controle e o método **processDominanceFrontierOfAllBlocks()** calcula a fronteira de dominância para cada bloco básico. O Trecho de Código 18 contém o método.

Trecho de Código 18 — Calculando a fronteira de dominância

```
1 private void processDominanceFrontierOfAllBlocks(ControlFlowGraph cfg) {
2     this.dominance = this.getDominance(cfg);
3     for (BasicBlock b : cfg.getBasicBlocks()) {
4         Dominance domB = dominance.stream().filter(d -> d.getBlock().getId() ==
              b.getId()).findFirst().get();
5         if (b.getPredecessors().size() >= 2) {
6             for (BasicBlock p : b.getPredecessors()) {
7                 BasicBlock runner = p;
8                 while (runner.getId() != domB.getImmediateDominator().getId()) {
9                     // add b to runner?s dominance frontier set
10                    this.addBlockIntroDominanceFrontierOfSecondBlock(b, runner);
11
12                    //get the next runner
13                    int idRunner = runner.getId();
14                    runner = dominance.stream().filter(d -> d.getBlock().getId()
                          == idRunner).findFirst().get()
15                    .getImmediateDominator();
16                }
```

```
17              }
18          }
19      }
20  }
```

Fonte: Próprio autor

A próxima etapa é inserir as funções *phi* na fronteira de dominância de cada bloco básico, mas somente onde as variáveis estão vivas na entrada do bloco. É utilizada a *Análise de Vida de Variáveis* para criar a versão do *e-SSA* mínima. A classe **PhiFunctionsDominanceFrontier** é responsável por adicionar as funções *phi*, recebendo no construtor o objeto da análise de intervalo de valores. O Trecho de Código 19 contém o método **addPhiFunctionsOnTheblocks()** da classe citada.

Trecho de Código 19 — Inserindo as funções phi na fronteira de dominância de cada bloco básico

```
 1 public void addPhiFunctionsOnTheblocks() {
 2     ControlFlowGraph cfg = essa.getCfg();
 3     List<DominanceFrontier> listDominanceFrontier = essa.getDominanceFrontier();

 4
 5     List<BasicBlock> basicBlocks = cfg.getBasicBlocks();
 6     List<PhiFunctionsAdded> phiAdded = new ArrayList<>();
 7     //each block get all
 8     for (BasicBlock basicBlock : basicBlocks) {
 9         //find dominance frontier
10         Optional<DominanceFrontier> findFirst = listDominanceFrontier.stream().
               filter(d -> d.getBlock().getId() == basicBlock.getId()).findFirst();
11         if(findFirst.isPresent()) {
12             DominanceFrontier dominanceFrontier = findFirst.get();
13             List<Token> alltAssignemtToVariableTokens = basicBlock.
                   getAlltAssignemtToVariableTokens();
14             for (Token token : alltAssignemtToVariableTokens)
15                 for(BasicBlock blockFrontier : dominanceFrontier.
                       getFrontierDominance())
16                     if(!phiAdded.stream().anyMatch(ph -> ph.basicBlock.getId()
                           == blockFrontier.getId() && ph.phiVariables.stream().
                           anyMatch(var -> var.equals(token.getLexem()))))
17                         && isVariableLive(essa ,blockFrontier, token.getLexem())
                               )
18                         addPhiBlockVariable(phiAdded, blockFrontier, token.
                               getLexem());
19
20         }
21     }
22 }
```

Fonte: Próprio autor

Com as funções *phi* inseridas é necessário adicionar os dados extraídos

de cada condicional do código. A classe **IntervalFunctions** recebe no construtor o objeto da análise de intervalo de valores e processa cada bloco básico procurando por condicionais simples (dois operandos e um operador de comparação). Quando encontra um condicional cria uma instrução com o intervalo dado tanto para o caminho verdadeiro quanto para o falso. O Trecho de Código 20 contém os métodos responsáveis pela adição dos dados extraídos.

Trecho de Código 20 — Inserindo intervalo encontrado no condicional

```
 1 public void addIntervalData() {
 2     ControlFlowGraph cfg = essaIntervalValueAnalisys.getCfg();
 3
 4     for(BasicBlock block : cfg.getBasicBlocks()) {
 5         Instruction lastInstruction = block.getLastInstruction();
 6         if(lastInstruction.containsIf())
 7             this.addIntervalNextBlock(block);
 8     }
 9 }
10
11 private void addIntervalNextBlock(BasicBlock block) {
12     ConditionalIntervalBuilder conditionalIntervalBuilder = new
           ConditionalIntervalBuilder(getConditionalInstrunction(block),
           essaIntervalValueAnalisys);
13
14     if(conditionalIntervalBuilder.containsIntervalData()) {
15         Optional<Entry<LinkEnum, BasicBlock>> trueSucessor = block.
               getTrueSucessor();
16         if(trueSucessor.isPresent())
17         addInstrunctionsIntoTheBlock(trueSucessor.get().getValue(),
               conditionalIntervalBuilder.getIntervalInstructionFalse());
18
19         Optional<Entry<LinkEnum, BasicBlock>> falseSucessor = block.
               getFalseSucessor();
20         if(falseSucessor.isPresent())
21         addInstrunctionsIntoTheBlock(falseSucessor.get().getValue(),
               conditionalIntervalBuilder.getIntervalInstructionFalse());
22     }
23 }
```

Fonte: Próprio autor

A última etapa consiste em renomear as variáveis, para que cada uma seja atribuída somente uma vez no algoritmo. Variáveis temporárias não são renomeadas pois possuem já a característica de atribuição única. Para criar a numeração das variáveis foi criado um campo inteiro na classe **Token** chamado **ssaId**. A classe **SearchAndReplace** recebe no construtor o objeto de análise de intervalo de valores e o método **renameVariables** recebe o bloco de entrada do algoritmo. Recursivamente os blocos vão sendo processados resultando em suas variáveis renomeadas.

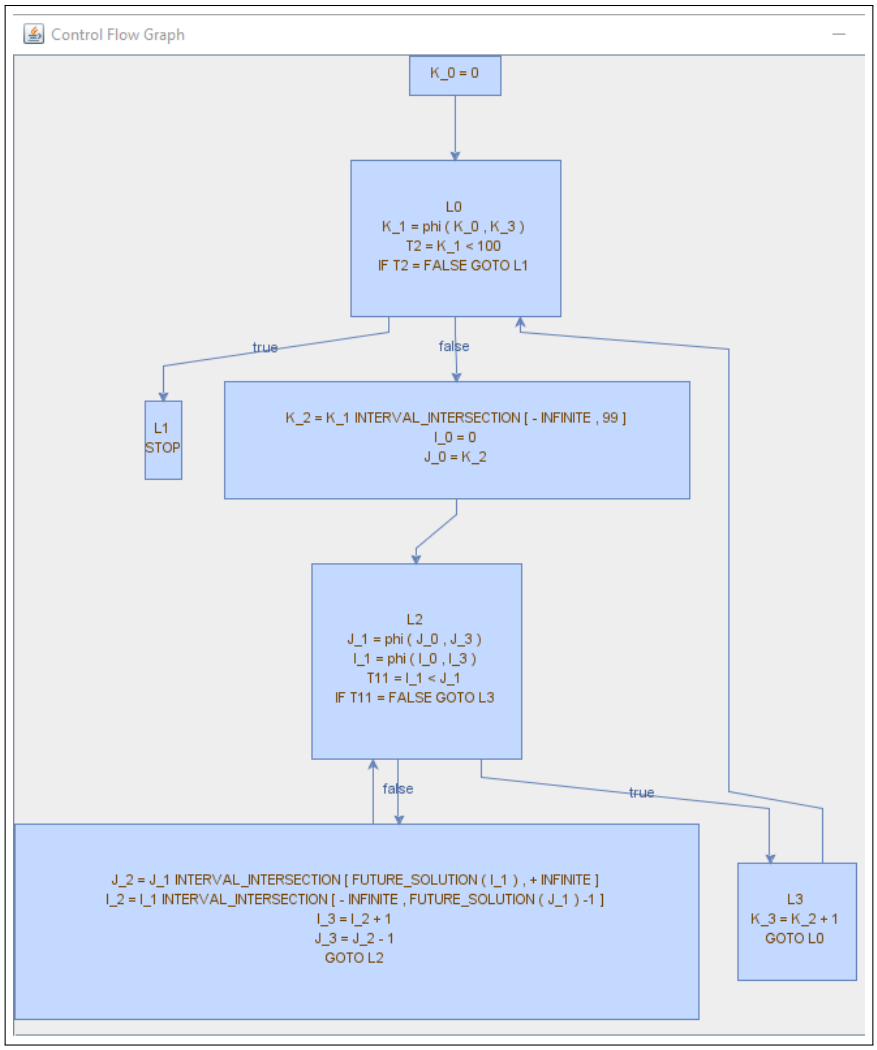
Trecho de Código 21 — Uma parte do método **renameVariables(BasicBlock basicBlock)** que preenche o identificador *SSA*

```
1 //onlyh if is a atribution to a variable
2 if(instruction.isAtributionToAVariable()) {
3     Token atribDest = instruction.getAtributionDestionation();
4     VariableCount variableCount = getVariableCount(atribDest.getLexem());
5     atribDest.setSsaId(variableCount.count);
6
7     VariableStack variableStack = getVariableStack(atribDest.getLexem());
8     variableStack.stack.push(variableCount.count);
9     variableCount.count++;
10 }
```

Fonte: Próprio autor

Com esta etapa finalizada o algoritmo está convertido para a representação *e-SSA*. A Figura 33 contem um exemplo.

Figura 33 — Representação da Análise de Intervalo de Valores



Fonte: Próprio autor

4.5.2 Montando Grafo de Dependências

Para montar o grafo de dependências é necessário inicialmente extrair as restrições do algoritmo. A classe **Constraints** extrai somente restrições pertinentes a análise de intervalo de valores, somente operações que resultem em um resultado de tipo *inteiro*. Com a lista de restrições a classe **DependencyGraph** monta o grafo de dependência resultando em uma lista com objeto **Dependency**, esse objeto contém a lista de dependências adjacentes. O Trecho de Código 22 contém o algoritmo para montar o grafo de dependências.

Trecho de Código 22 — Montando grafo de dependências

```
1 List<List<Token>> linkFutureSolution = new ArrayList<>();
2 for (Instruction instruction :  restrictionSystem.getInstructions()) {
3     Token atributionDestionation = instruction.getAtributionDestionation();
4     List<Token> otherTokens = instruction.getTokensOnOtherSideOfTheAtribution();
```

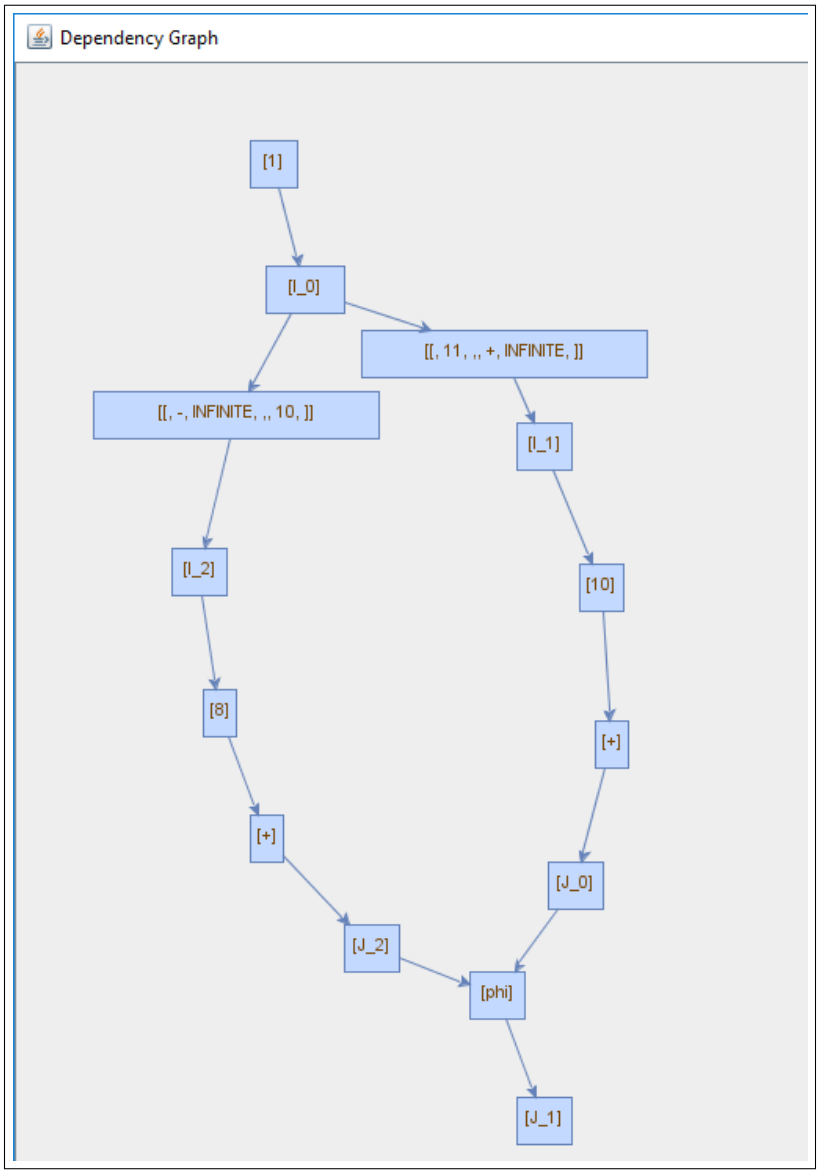
```
 5
 6     Dependency atribution = this.getDependency(instruction,
           atributionDestionation);
 7     if(instruction.isForIntervalAnalysis()) {
 8         // i_2 = i_1 intersection [ft(j+1), + infinite]
 9         Token token = otherTokens.get(0);
10
11         Dependency interval = createNewDependency(instruction, otherTokens.
               subList(2, otherTokens.size()));
12         interval.addAdjactent(atribution);
13
14         linkFutureSolution.add(interval.getTokens());
15
16         Dependency dependency = getDependency(instruction, token);
17         dependency.addAdjactent(interval);
18
19     }
20     else if(instruction.containsPhiFunction()) {
21         //i_5 = phi (i_3, i_4)
22         Token phi = instruction.getTokens().stream().filter(tk -> tk.getId() ==
               TokenDictionary.getIdToken(TokenDictionary.TK_PHI)).findFirst().get
               ();
23         Dependency phiDep = createNewDependency(instruction, phi);
24         phiDep.addAdjactent(atribution);
25
26         for (Token token :  instruction.getUsedVariables()) {
27             Dependency usedVariables = getDependency(instruction, token);
28             usedVariables.addAdjactent(phiDep);
29         }
30     }
31     else
32     {
33         //i_1 = j_1 + 1 // i_1 = 1
34         Dependency depSecond = atribution;
35         if(otherTokens.size() > 1) {
36             Dependency depNew = createNewDependency(instruction, otherTokens.get
                   (1));
37             depNew.addAdjactent(atribution);
38
39             depSecond = getDependency(instruction, otherTokens.get(2));
40             depSecond.addAdjactent(depNew);
41         }
42
43
44         Dependency dep = getDependency(instruction, otherTokens.get(0));
45         dep.addAdjactent(depSecond);
46     }
47 }
48
49
50
51 //link future solution to the variables
52 for (List<Token> ft : linkFutureSolution) {
53     Instruction instruction = new Instruction(ft, 1);
54     if(instruction.containsFutureSolution()) {
55         Token token = instruction.getTokensIdentifiers().get(0);
56         Dependency variable = getDependency(instruction, token);
57         Dependency interval = this.dependencies.stream().filter(dep -> dep.
               isTheSame(ft)).findFirst().get();
```

```
58
59          variable.addAdjactent(interval);
60      }
61  }
```

Fonte: Próprio autor

Para representar graficamente o grafo de dependências foi criada uma classe chamada **RenderingDependencies**. Esta classe gera um **JFrame** com o grafo, a Figura 34 contém o resultado da geração do grafo de um código simples.

Figura 34 — Grafo de Dependências



Fonte: Próprio autor

4.5.3 Encontrando Componentes Fortemente Conexos

Os componentes fortemente conexos são construídos a partir do grafo de dependências pela classe **StronglyConnectedComponent**. O algoritmo retorna uma lista dentro de outra lista de identificadores, cada lista representa um componente fortemente conexo e os identificadores se referem às dependências. O Trecho de Código 23 contém a função que calcula os componentes.

Trecho de Código 23 — Encontrando os componentes fortemente conexos

```
1 //process the dependencies
2 public List<List<Integer>> calculateSCC(){
3     Integer[][] edges = new Integer[dependencies.size()][dependencies.size()];
4
5     for (Dependency dep : dependencies) {
6         for(Dependency adj: dep.getAdjacent()) {
7             Integer[] row = edges[dep.getId()];
8             if(row[adj.getId()] == null) {
9                 addEdge(dep.getId(), adj.getId());
10                row[adj.getId()] = 1;
11            }
12        }
13    }
14
15    return calculateSCCs();
16 }
```

Fonte: Próprio autor

4.5.4 Processando intervalos

A classe **Interval** representa um intervalo, nela é utilizada a classe **Double** do java para representar o intervalo pois ela permite armazenar o valor infinito. Para resolver o sistema de restrições foi criada a classe **ConstraintSystem** que recebe no construtor o grafo de dependências e o método **resolveConstraint** recebe uma instrução e retorna um intervalo. O Trecho de Código 24 contém uma parte do algoritmo que calcula um sistema de restrição.

Trecho de Código 24 — Resolução de um sistema de restrição

```
1 if(ins.isForIntervalAnalysis()) {
2     Interval dep1 = getDependency(otherTokens.get(0)).getInterval();
3     Interval interSectionInterval = getInterval(otherTokens);
4
5
6     Double a1 = dep1.getLower();
7     Double a2 = dep1.getUpper();
8
9     Double b1 = interSectionInterval.getLower();
10    Double b2 = interSectionInterval.getUpper();
```

```
11
12      try {
13          if(((a1 <= b1) && (b1 <= a2)) || ((b1 <= a1) && (a1 <= b2))) {
14              interval.setLower(Double.max(a1, b1));
15              interval.setUpper(Double.min(a2, b2));
16          }
17      }catch(Exception e) {
18          interval.setLower(null);
19          interval.setUpper(null);
20      }
21  }
```

Fonte: Próprio autor

A classe **IntervalProcessor** é responsável por executar as etapas da análise de intervalo de valores em cada componente fortemente conexo. As etapas consistem em alargamento do intervalo das variáveis, correção das intersecções e estreitamento do intervalo das variáveis. O Trecho de Código 25 contém uma parte do algoritmo que processa os intervalos.

Trecho de Código 25 — Processando intervalos

```
1 for (List<Integer> scc : stronglyConnectedComponents) {
2
3     Widening intervalWidening = new Widening(dependencies, scc);
4     Narrowing inntervalNarrowing = new NarrowingAnalisys(dependencies, scc);
5     for (int i = 0; i < TIMES_TO_CONVERGE; i++) {
6         //widening intervals and resolve future resolution
7         intervalWidening.wideIntervals();
8
9         //Narrowing Analysis
10        inntervalNarrowing.narrowIntervals();
11    }
12 }
```

Fonte: Próprio autor

A classe **Widening** é responsável por alargar os intervalos e corrigir intersecções. Ela recebe no construtor o grafo de dependências e o componente fortemente conexo. O método **wideIntervals()** percorre o componente fortemente conexo calculando o intervalo das dependências. O Trecho de Código 26 contém o cálculo do intervalo.

Trecho de Código 26 — Alargamento do intervalo e corrigindo intersecções de uma dependência

```
1 Interval interval = dep.getInterval();
2 Interval resolveRestriction = constraintSystem.resolveConstraint(dep.
      getInstruction());
3
4 //I[Y] = [ null, null]
```

```
 5 if(interval.getLower() == null && interval.getUpper() == null) {
 6     dep.setInterval(resolveRestriction);
 7 }else if(resolveRestriction.getLower() < interval.getLower() &&
       resolveRestriction.getUpper() > interval.getUpper()) { //E(Y)lower < I[Y]
       lower && E(Y)upper > I[Y]upper
 8     dep.getInterval().setLower(Double.NEGATIVE_INFINITY);
 9     dep.getInterval().setUpper(Double.POSITIVE_INFINITY);
10 }else if(resolveRestriction.getLower() < interval.getLower()) {//E(Y)lower < I[Y
       ] lower
11     dep.getInterval().setLower(Double.NEGATIVE_INFINITY);
12 }else if (resolveRestriction.getUpper() > interval.getUpper()) {//E(Y)upper > I[
       Y]upper
13     dep.getInterval().setUpper(Double.POSITIVE_INFINITY);
14 }
```

Fonte: Próprio autor

A classe **Narrowing** é responsável por estreitar os intervalos. Ela recebe no construtor o grafo de dependências e o componente fortemente conexo. O método **narrowIntervals()** percorre o componente fortemente conexo calculando o intervalo das dependências. O Trecho de Código 27 contém o cálculo do intervalo.

Trecho de Código 27 — Estreitamento do intervalo de uma dependência

```
 1 Interval interval = dep.getInterval();
 2 Interval evaluation = constraintSystem.resolveConstraint(dep.getInstruction());
 3
 4 try {
 5     if(interval.getLower() == Double.NEGATIVE_INFINITY && evaluation.getLower()
           > Double.NEGATIVE_INFINITY) {
 6         interval.setLower(evaluation.getLower());
 7     }else if(interval.getLower() > evaluation.getLower()) {
 8         interval.setLower(evaluation.getLower());
 9     }else if(interval.getUpper() == Double.POSITIVE_INFINITY && evaluation.
           getUpper() < Double.POSITIVE_INFINITY) {
10         interval.setUpper(evaluation.getUpper());
11     }else if(interval.getUpper() < evaluation.getUpper()) {
12         interval.setUpper(evaluation.getUpper());
13     }
14 }catch(Exception e) {
15     //if one of the objects are null don't need to narrow
16 }
```

Fonte: Próprio autor

4.5.5 Identificando erros

A análise de intervalo de valores é utilizada para identificar expressões constantes, laços infinitos e erros de índices de vetores. As descrições destes

erros podem ser encontradas nas seções 3.3 e 3.4.

No código de três endereços tanto um condicional como um laço são representados por um condicional, portanto a identificação é feita da mesma forma para ambas as situações. Para encontrar uma expressão constante em um condicional é necessário testar a expressão com todas as combinações possíveis dos intervalos. Geralmente são quatro combinações, pois temos os limites inferior e superior, somente no caso do operador *igual* é comparado somente duas vezes. Se esses testes resultarem sempre em verdadeiro ou falso temos uma expressão constante e após é descoberto o que o condicional representa no algoritmo (enquanto, repita, se ou para) assim é atribuída a mensagem de erro. O Trecho de Código 28 contém o algoritmo que executa essas análises.

Trecho de Código 28 — Testando todas combinações possíveis nos condicionais

```
 1 if(operator.getId() == TokenDictionary.getIdToken(TokenDictionary.TK_GREATER)) {
 2     results.add(op1Interval.getLower() > op2Interval.getLower());
 3     results.add(op1Interval.getLower() > op2Interval.getUpper());
 4     results.add(op1Interval.getUpper() > op2Interval.getLower());
 5     results.add(op1Interval.getUpper() > op2Interval.getUpper());
 6 }
 7 else
 8 if(operator.getId() == TokenDictionary.getIdToken(TokenDictionary.TK_LESSER)) {
 9     results.add(op1Interval.getLower() < op2Interval.getLower());
10     results.add(op1Interval.getLower() < op2Interval.getUpper());
11     results.add(op1Interval.getUpper() < op2Interval.getLower());
12     results.add(op1Interval.getUpper() < op2Interval.getUpper());
13 }
14 else
15 if(operator.getId() == TokenDictionary.getIdToken(TokenDictionary.
       TK_GREATER_EQUAL)) {
16     results.add(op1Interval.getLower() >= op2Interval.getLower());
17     results.add(op1Interval.getLower() >= op2Interval.getUpper());
18     results.add(op1Interval.getUpper() >= op2Interval.getLower());
19     results.add(op1Interval.getUpper() >= op2Interval.getUpper());
20 }
21 else
22 if(operator.getId() == TokenDictionary.getIdToken(TokenDictionary.
       TK_LESSER_EQUAL)) {
23     results.add(op1Interval.getLower() <= op2Interval.getLower());
24     results.add(op1Interval.getLower() <= op2Interval.getUpper());
25     results.add(op1Interval.getUpper() <= op2Interval.getLower());
26     results.add(op1Interval.getUpper() <= op2Interval.getUpper());
27 }
28 else
29 if(operator.getId() == TokenDictionary.getIdToken(TokenDictionary.TK_EQUAL)) {
30     results.add(op1Interval.getLower().doubleValue() == op2Interval.getLower().
           doubleValue());
31     results.add(op1Interval.getLower().doubleValue() == op2Interval.getUpper().
           doubleValue());
32 }
33
34
35 if(!results.isEmpty()) {
```

```
36      long ttrue = results.stream().filter(a -> a.booleanValue()).count();
37      long ffalse = results.stream().filter(a -> !a.booleanValue()).count();
38
39      ConditionalType conditionalType = getConditionalType(conditional);
40
41      if(ttrue == 0 || ffalse == 0) {
42          if(conditionalType == ConditionalType.IF) {
43              problems.add(new ProblemFound( ffalse == 0 ? ProblemsEnum.
                    IF_ALWAYS_TRUE : ProblemsEnum.IF_ALWAYS_FALSE, exp.getLine()));
44          }else if(conditionalType == ConditionalType.FOR || conditionalType ==
                ConditionalType.WHILE) {
45              problems.add(new ProblemFound( ffalse == 0 ? ProblemsEnum.
                    LOOP_ALAWYS_TRUE : ProblemsEnum.LOOP_ALAWYS_FALSE, exp.getLine()
                    ));
46          }else if(conditionalType == ConditionalType.DO) {
47              problems.add(new ProblemFound( ffalse == 0 ? ProblemsEnum.
                    REPEAT_UNTIL_ALWAYS_TRUE : ProblemsEnum.
                    REPEAT_UNTIL_ALWAYS_FALSE, exp.getLine()));
48          }
49      }
```

Fonte: Próprio autor

Para identificar o erro de índice de vetores é percorrido todo o código de três endereços a procura de acesso em vetores. Esta análise testa o intervalo acessado no vetor em todas dimensões do mesmo. A comparação é feita utilizando a declaração das dimensões do vetor no algoritmo e utilizando o intervalo descoberto para verificar se o vetor vai ser acessado fora dos seus limites. O Trecho de Código 29 contém o algoritmo que procura acesso indevido no vetor.

Trecho de Código 29 — Testando o intervalo nas dimensões dos vetores

```
1  for (List<Token> vector : returnAllVectors) {
2      Token variable = vector.get(0);
3      List<Interval> vectorDeclaredInterval = getVectorDeclaredInterval(variable.
           getLexem(), allDeclaredVariables);
4      List<Token> vectorIdentifier = getVectorIdentifier(vector);
5
6      for (int i = 0; i < vectorIdentifier.size(); i++) {
7          Token usedVariable = vectorIdentifier.get(i);
8
9          Interval intervalDeclared = vectorDeclaredInterval.get(i);
10         Interval variableInterval = getIntervalOfToken(usedVariable);
11
12         //test with the lower interval
13         if(variableInterval.getLower() != null) {
14             if(variableInterval.getLower() < intervalDeclared.getLower() ||
                   variableInterval.getLower() > intervalDeclared.getUpper()) {
15                 problems.add(new ProblemFound(ProblemsEnum.VECTOR_OUT_OF_BOUNDS,
                       inst.getLine(), variable.getLexem()));
16                 break;
17             }
18         }
19
20         //test with the upper interval
```

```
21          if(variableInterval.getUpper() != null) {
22              if(variableInterval.getUpper() > intervalDeclared.getUpper() ||
                    variableInterval.getUpper() < intervalDeclared.getLower()) {
23                  problems.add(new ProblemFound(ProblemsEnum.VECTOR_OUT_OF_BOUNDS,
                        inst.getLine(), variable.getLexem()));
24                  break;
25              }
26          }
27      }
28  }
```

Fonte: Próprio autor

4.6 INTEGRAÇÃO COM O PORTAL DE ALGORITMOS

O portal de algoritmos da *UCS* é composto por uma interface *web*, compilador, banco de dados e um sistema de validação de correção dos exercícios. A interface web tem como propósito interagir com o usuário (aluno ou professor), disponibilizando uma ferramenta para desenvolvimento de algoritmos e solução de problemas cadastrados pelos professores. O compilador analisa, interpreta e executa o algoritmo. E o banco de dados contém os problemas salvos e as soluções desenvolvidas pelos alunos e dados de teste para verificar a correção dos exercícios. É desenvolvido em Java da mesma forma que o analisador desenvolvido por este trabalho.

A integração é feita via importação do *JAR* (*Java ARchive*) compilado do analisador estático. A classe **AlgoPanel** é a interface gráfica do WebAlgo, o método **executar** executa a compilação do algoritmo e se não houver nenhum problema é chamada a análise estática. O Trecho de Código 30 contém o algoritmo no WebAlgo que chama a análise estática e adiciona no painel de compilação se encontra algum erro.

Trecho de Código 30 — Chamando a análise estática

```
1 public static void executarAnaliseEstaticaCodigo(AlgoPanel algo, String codigo)
      {
2     StaticAnalysis staticAnalysis = new StaticAnalysis();
3     try {
4         staticAnalysis.doStaticAnalysis(codigo).stream().forEach(algo::
              adicionaAviso);
5     } catch (Exception e) {
6         e.printStackTrace();
7     }
8 }
```

Fonte: Próprio autor

A classe **StaticAnalysis** do analisador estático recebe por parâmetro o código e executa todas as análises implementadas neste trabalho. Retorna uma lista de problemas encontrados se existirem senão retorna vazia. O Trecho de Código 31 contém o algoritmo que executa as análises.

Trecho de Código 31 — Executando todas as análises

```
1 public List<ProblemFound> doStaticAnalysis(Code code) throws Exception {
2     List<ProblemFound> problems = new ArrayList<>();
3
4     Lexicon lexico = Lexicon.instanciaLexio(code);
5     NodoAlgoritmo nodeAlgorithm = NodoAlgoritmo.analisar(lexico);
6
7     List<ThreeAdressCodeAlgorithm> algorithm = nodeAlgorithm.
          generateThreeAdressCode();
8
9     for (ThreeAdressCodeAlgorithm algorithmThreeAdressCode : algorithm) {
10        ControlFlowGraph cod = new ControlFlowGraph(algorithmThreeAdressCode);
11        new Rendering(cod.getTheEntryBlock());
12
13        //Variable not used
14        VariableNotUsedAnalysis variableNoteUsedAnalysis = new
              VariableNotUsedAnalysis(cod);
15        problems.addAll(variableNoteUsedAnalysis.performStaticAnalysis());
16
17        //Live variable analysis
18        LiveVariableAnalysis liveVariableAnalysis = new LiveVariableAnalysis(cod
              );
19        problems.addAll(liveVariableAnalysis.performStaticAnalysis());
20
21        //Reaching Definitions
22        ReachingDefinitionsAnalysis reachingDefinitions = new
              ReachingDefinitionsAnalysis(cod);
23        problems.addAll(reachingDefinitions.performStaticAnalysis());
24
25        //ESSA Interval
26        IntervalValueAnalisys essaIntervalValueAnalisys = new
              IntervalValueAnalisys(cod, liveVariableAnalysis.getBlockLiveVariable
              ());
27        problems.addAll(essaIntervalValueAnalisys.performStaticAnalysis());

28    }
29
30
31    return problems;
32 }
```

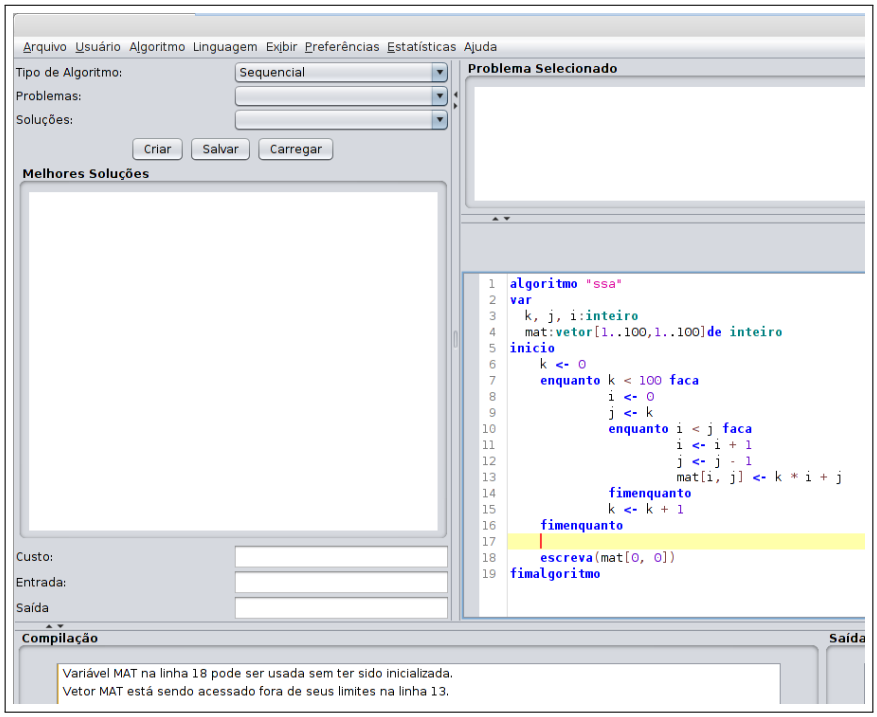
Fonte: Próprio autor

5 TESTES REALIZADOS

Aqui são descritos alguns dos testes para a validação do analisador estático de código desenvolvido. Alguns dos testes foram realizados com códigos providenciados pelo orientador. Foram montados algoritmos com erros para testar a detecção e também algoritmos sem erros para verificar se o analisador não produzia falsos positivos.

Para provocar o erro de índices de vetores foi utilizado um algoritmo com dois laços de controles. A Figura 35 contém o algoritmo no *WebAlgo* com o aviso de que o vetor **mat** pode ser acesso fora dos seus limites na linha 13.

Figura 35 — Algoritmo com erro de índice de um vetor

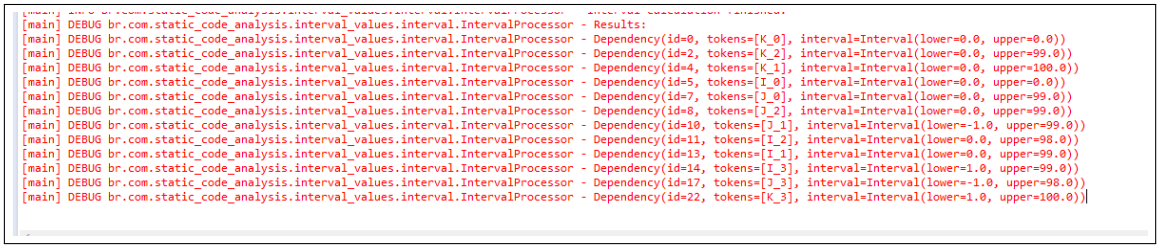


Fonte: Próprio autor

O que faz com que isso aconteça é a variável **j** que no momento de

atribuição do vetor pode chegar ao valor **-1**. A Figura 36 contém o *log* que representa todos os intervalos assumidos pelas variáveis.

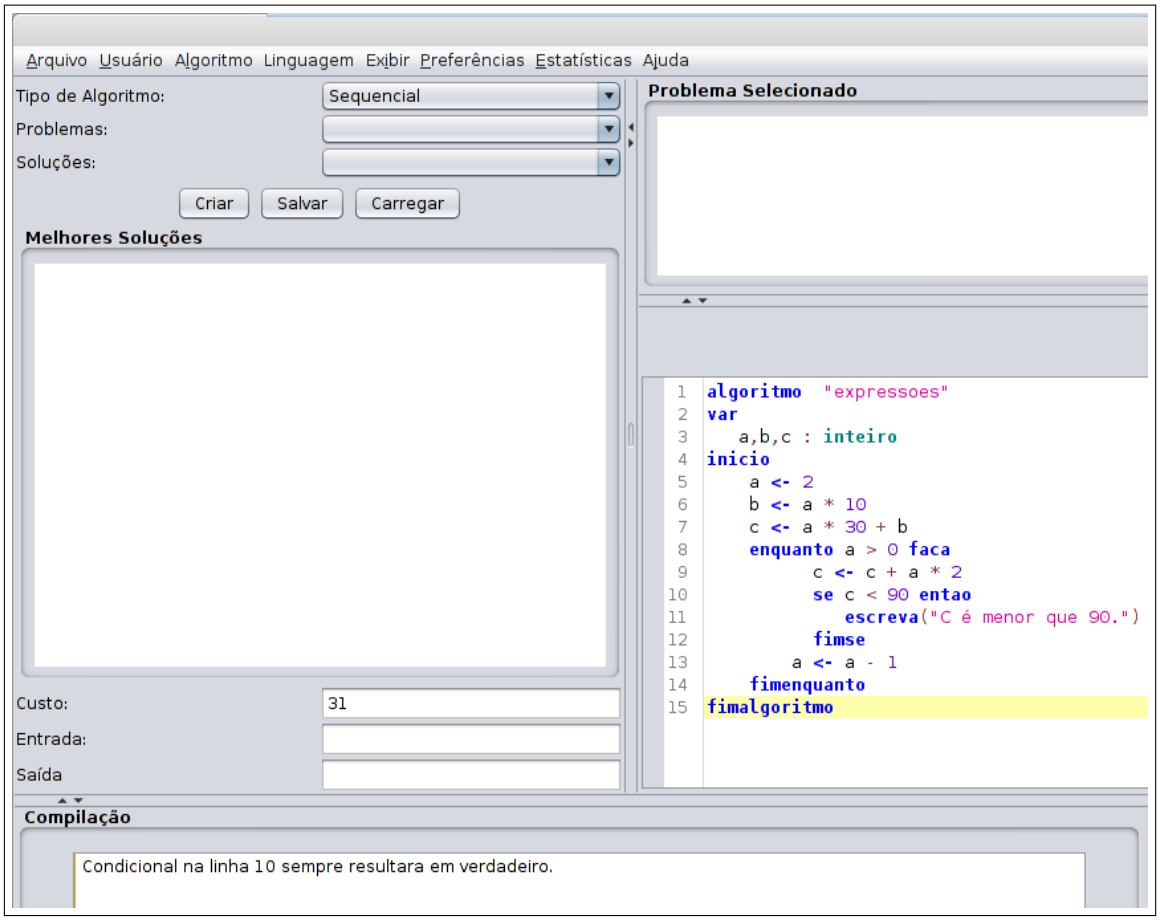
Figura 36 — *Log* com intervalos assumidos pelas variáveis



Fonte: Próprio autor

Para provocar uma expressão constante foi utilizado um algoritmo que no laço de controle é verificado se a variável **c** é menor que **90**. Só que a variável **c** nunca vai ser maior que **90** então a mensagem de condicional sempre verdadeiro é apresentada. A Figura 37 contém o algoritmo no *WebAlgo* com o aviso de que o condicional na linha 10 sempre resultará em verdadeiro.

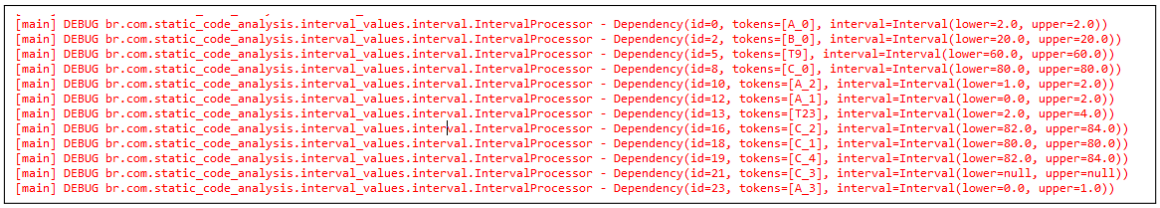
Figura 37 — Algoritmo com expressão constante



Fonte: Próprio autor

Isto acontece pois ao longo do laço a variável c incrementa **2** vezes o valor de **a** e chega no valor máximo de 84. A Figura 38 contém o *log* que representa todos os intervalos assumidos pelas variáveis.

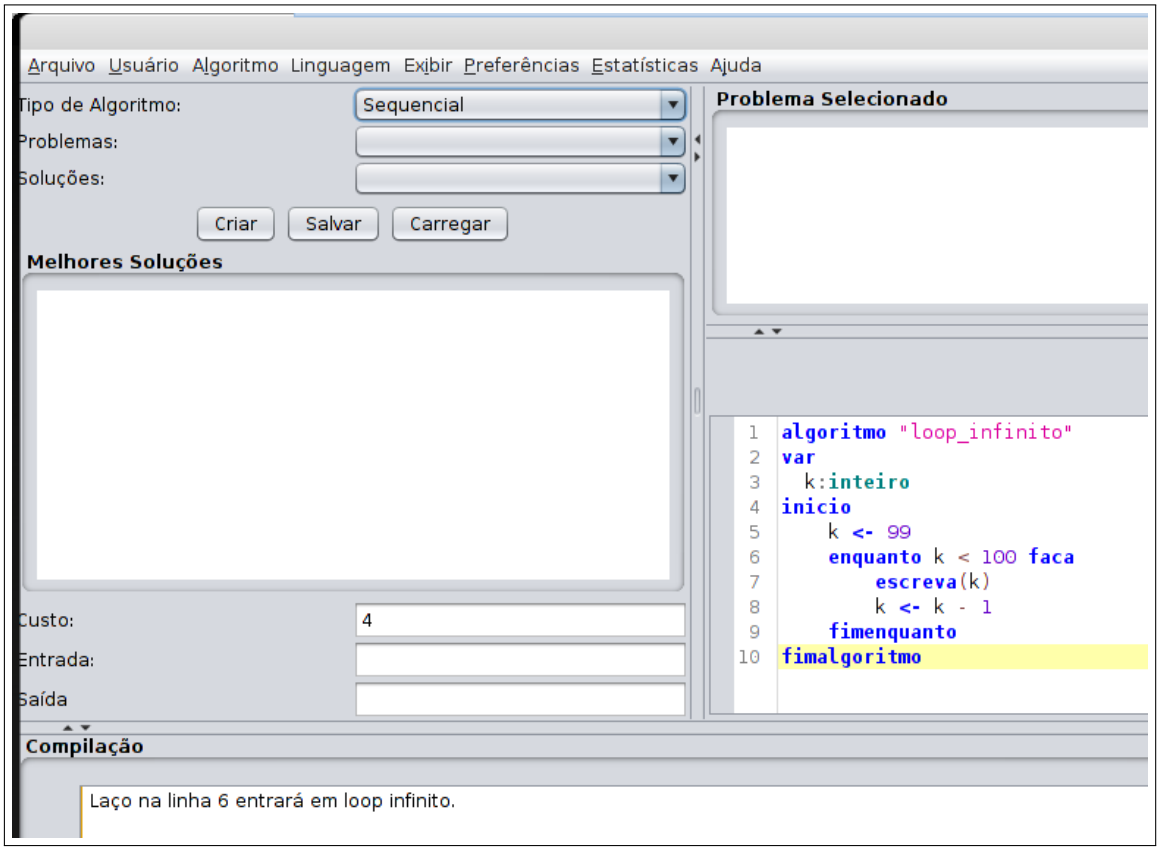
Figura 38 — *Log* com intervalos assumidos pelas variáveis



Fonte: Próprio autor

Para provocar um *loop* infinito foi utilizado um algoritmo em que a variável **k** inicia com o valor **99** e o laço executa enquanto o valor desta variável for menor que **100**. Dentro do laço a variável **k** é decrementada em uma unidade a cada iteração, resultando que o valor dela nunca chegue a **100**. A Figura 39 contém o algoritmo no *WebAlgo* com o aviso de que o laço na linha 6 entrará em loop infinito.

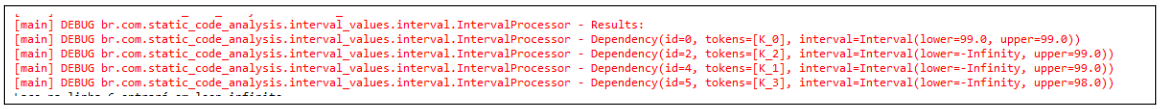
Figura 39 — Algoritmo com loop infinito



Fonte: Próprio autor

Portanto a expressão é constante verdadeira ocasionando em um *loop* infinito. A Figura 40 contém o *log* que representa todos os intervalos assumidos pelas variáveis.

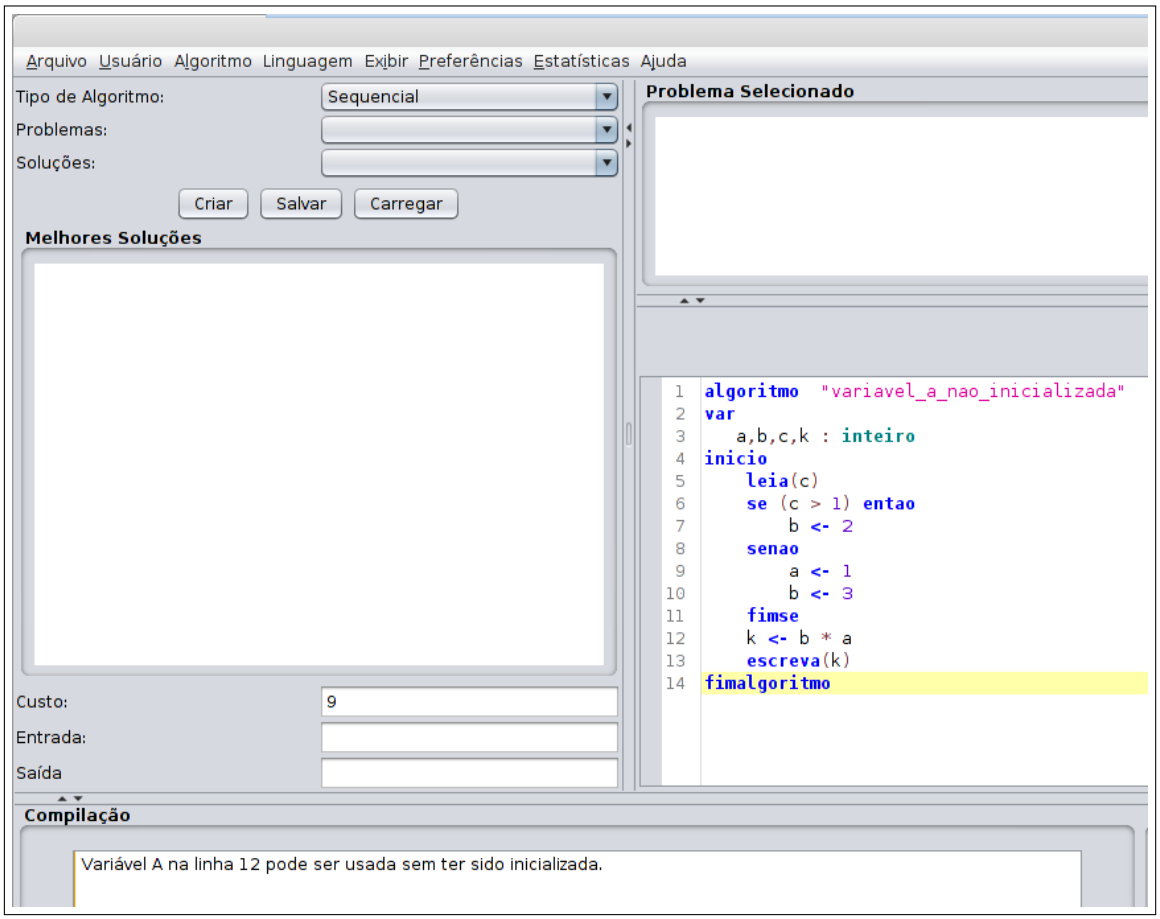
Figura 40 — *Log* com intervalos assumidos pelas variáveis



Fonte: Próprio autor

Para provocar o erro de variável não inicializada foi utilizado um algoritmo que em um condicional a variável **a** é definida somente no *senão*, e no final do algoritmo ela é utilizada na atribuição da variável **k**. Por causa disso existe um caminho no algoritmo em que a variável não foi definida. A Figura 41 contém o algoritmo no *WebAlgo* com o aviso do erro.

Figura 41 — Algoritmo com variável não inicializada

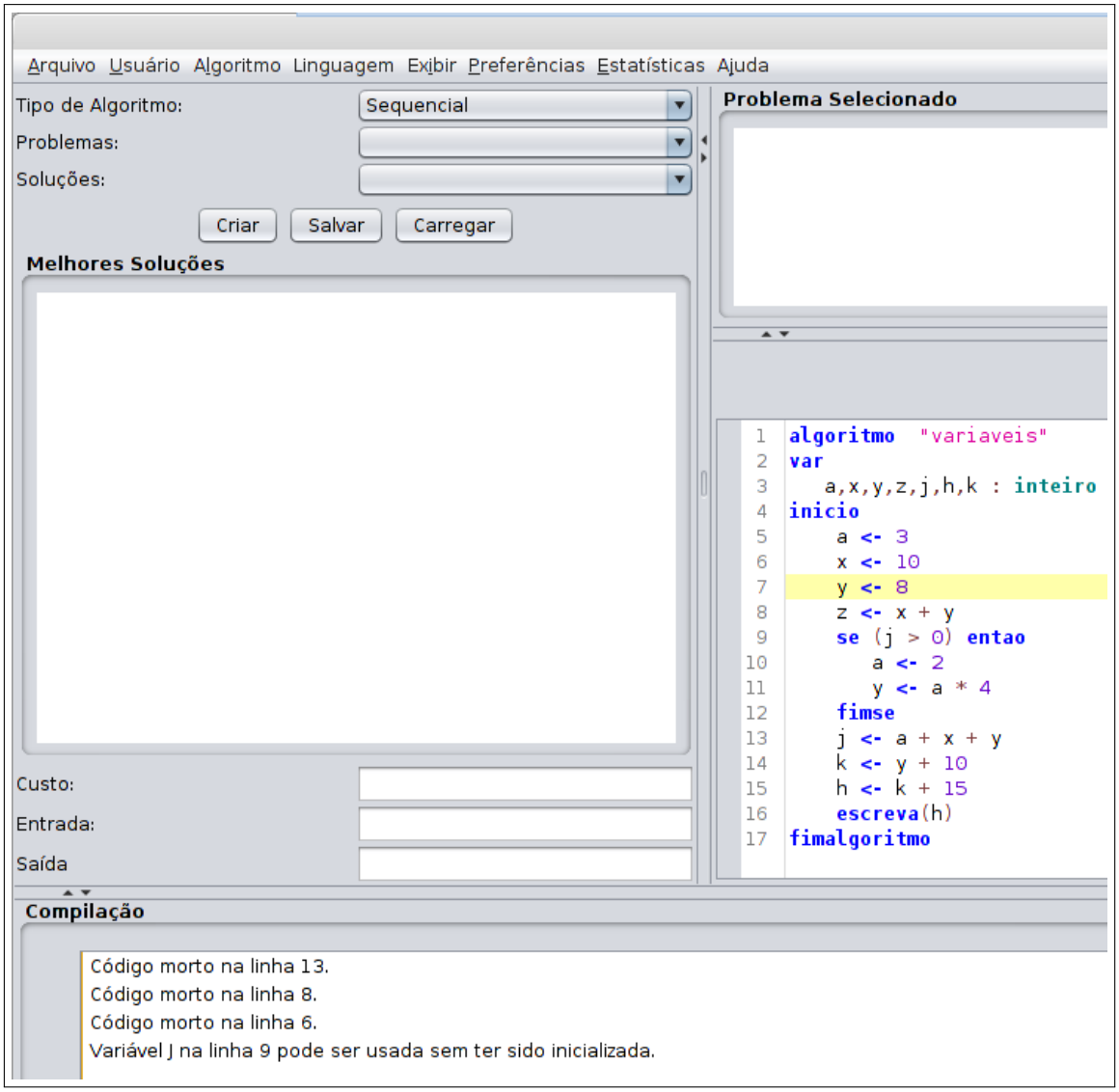


Fonte: Próprio autor

Para provocar o erro de código morto foi utilizado um algoritmo em que algumas instruções são executadas mas no final do algoritmo os resultados delas

não são usados. A Figura 42 contém o algoritmo no *WebAlgo* com o aviso do erro.

Figura 42 — Algoritmo com código morto



Fonte: Próprio autor

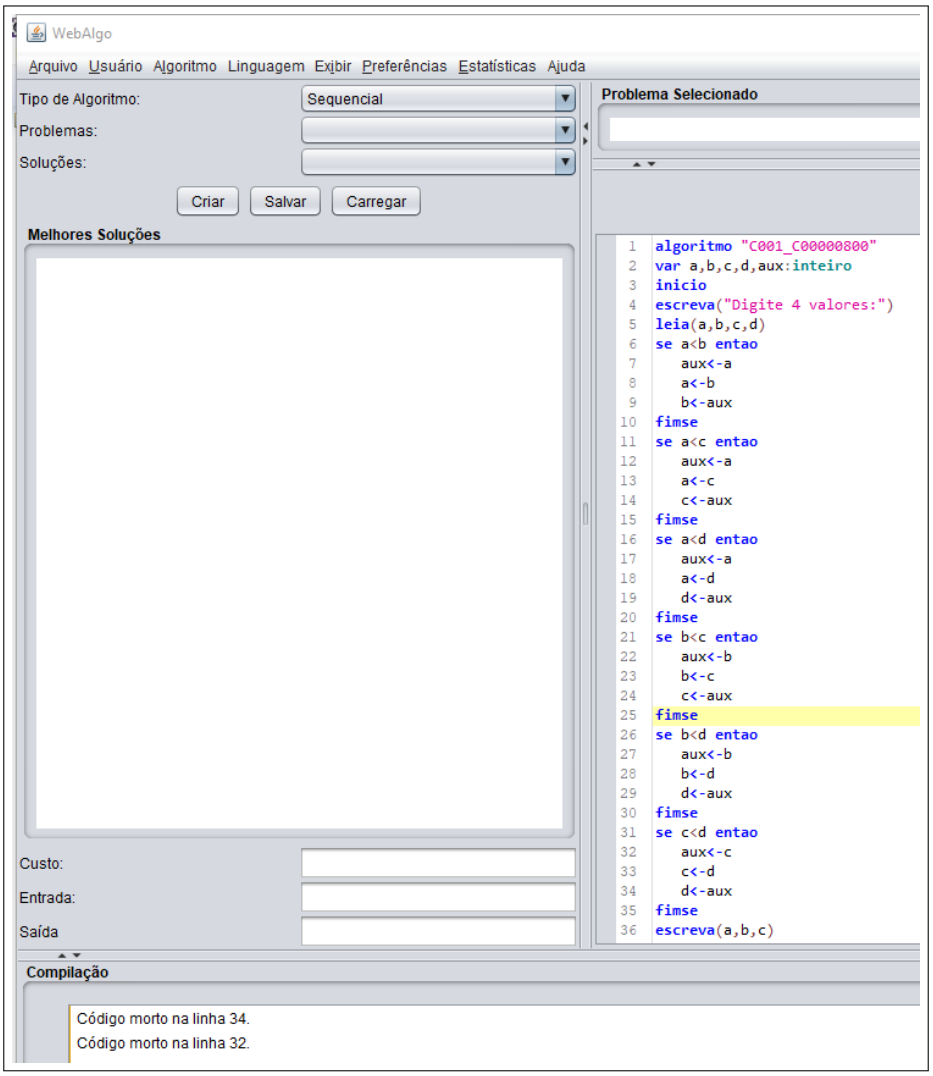
Neste exemplo é marcado código morto da linha 13 e 8 por estas atribuições não estarem sendo utilizadas ao longo do algoritmo, a atribuição da linha 6 foi utilizada mas em uma instrução que resultou em código morto, portanto é um código morto também.

Foram executados testes também na base de dados de soluções do meu orientador. Os próximos testes descritos são sobre esses algoritmos, no total foram 339 analisados.

No algoritmo da Figura 43 a variável **d** não foi utilizada após sua atribuição

e atribuição final da variável **aux** também não, desta maneira foi originado código morto na linha 34 e 32.

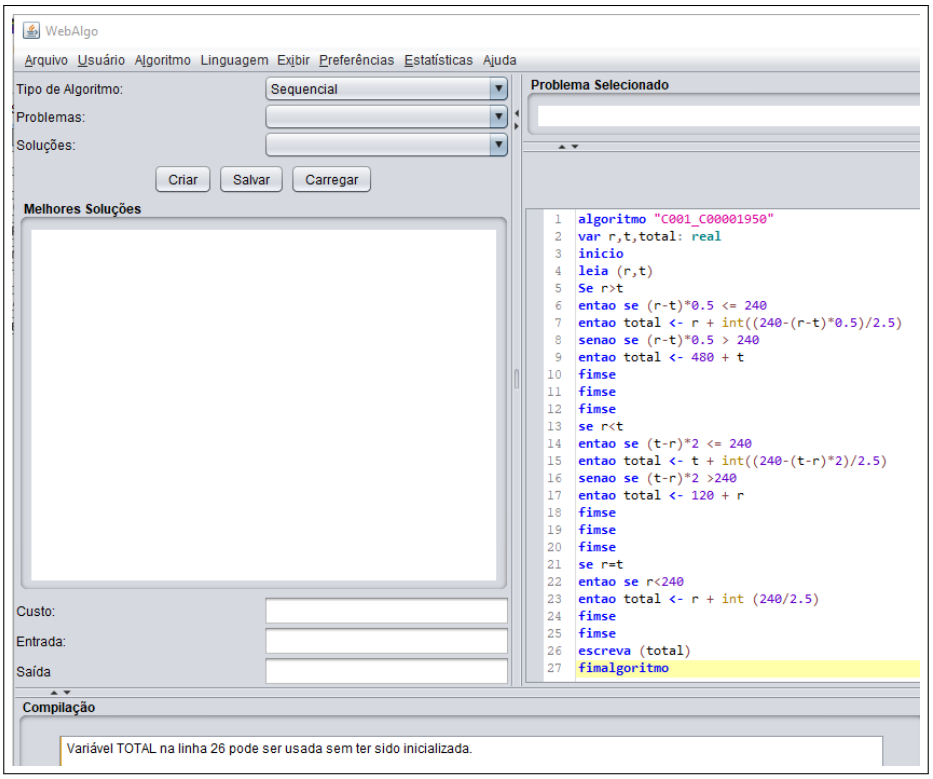
Figura 43 — Algoritmo da base de dados com código morto



Fonte: Próprio autor

No algoritmo da Figura 44 a variável **total** pode ser usada sem ter sido inicializada por causa da possibilidade de não entrar nos condicionais onde a mesma é definida.

Figura 44 — Algoritmo da base de dados com código morto



Fonte: Próprio autor

Estes exemplos acima tiveram ocorrência em mais 97 algoritmos, não houve incidência de erro de índice de vetores, expressão constantes ou laços infinitos.

6 CONCLUSÕES

Neste trabalho foi desenvolvido um analisador estático de código para o portal de algoritmos da *UCS*. O objetivo principal foi introduzir uma ferramenta para encontrar erros em códigos no qual o portal de algoritmos não tinha até o momento.

Todas análises levantadas no referencial teórico foram utilizadas e se mostraram eficazes na detecção de erros. O desenvolvimento da análise de intervalo de valores se mostrou menos complexa e mais trabalhosa do que o previsto, enquanto a análise de vida de variáveis e definições incidentes se mostraram menos trabalhosas e com complexidade esperada.

A integração com o portal de algoritmos foi uma etapa que consumiu um tempo consideravelmente menor do que o previsto. O código do portal foi fácil de se compreender e alterações foram de baixo nível de complexidade.

Os testes realizados com a base de dados de soluções do orientador foi de grande importância tanto para estabilizar o analisador estático quanto para efetuar testes. Os erros encontrados pelos testes efetuados foram facilmente resolvidos.

Algumas melhorias foram identificadas ao longo do desenvolvimento, na próxima Seção elas são descritas.

6.1 MELHORIAS FUTURAS

As análises desenvolvidas nesse trabalho foram análises locais, executadas dentro de funções, procedimentos e código principal. Um trabalho futuro poderia focar as análises no algoritmo como um todo, desta forma possivelmente seria encontrado mais erros pois o código inteiro do algoritmo seria tratado junto.

A análise de intervalo de valores foi desenvolvida em variáveis inteiras pelo que define o referencial teórico. A estrutura atual permite ativar a análise para variáveis numéricas, porem é necessário uma bateria de testes e inúmeros cenários para testar a funcionalidade. Um trabalho futuro poderia ativar a análise para variáveis numéricas e avaliar o quanto é benéfica para a detecção de erros.

O analisador foi desenvolvido somente para o português estruturado. Um trabalho futuro poderia integrar as análises de código com o portal de algoritmo em código *C*. É necessário desenvolver a conversão do código *C* para linguagem de três endereços descrita neste trabalho e possivelmente fazer alguns ajustes no projeto do analisador estático.

REFERÊNCIAS

AHO, A. V.; ULLMAN, J. D.; SETHI, R. **COMPILADORES**: principios, tecnicas e ferramentas. 2.ed. [S.l.]: LTC, 2008. p. 257-313.

ALLEN, F. E.; COCKE, J. **A Program Data Flow Analysis Procedure**. [S.l.]: Communications of the ACM, 1976. v.19, n.3. p. 137.

APACHE. **Apache Maven Project**. [S.l.: s.n.], 2018. <Disponível em: `https://maven.apache.org/`>. Acesso em: 29 de Março de 2018.

ATKINSON, D. C.; GRISWOLD, W. G. **Implementation Techniques for Efficient Data-Flow Analysis of Large Programs**. IEEE International Conference: Software Maintenance, 2001. p. 52-61.

BIGGIN, T. **Static Code Analysis**. Madison, WI 53706 - United States: University of Wisconsin, 2012.

BIRCH, J.; ENGELEN, R. van; GALLIVAN, K. **Value Range Analysis of Conditionally Updated Variables and Pointers**. Florida: Department of Computer Science and School of Computational Science and Information Technology, 2004.

BLUME, W.; EIGENMANN, R. **Symbolic Range Propagation**. Urbana, Illinois 61801-2307: Center for supercomputing Research and Develipment, 1994.

BRIGGS, P. et al. **Practical improvements to the construction and destruction of static single assignment forms**. [S.l.]: Reverse Engineering, 1998.

CHOI, J.-D.; CYTRON, R.; FERRANTE, J. **Automatic Construction of Sparse Data Flow Evaluation Graphs**. Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages: ACM, 1991.

COOPER, K. D.; HARVEY, T. J.; KENNEDY, K. **Iterative Data-flow Analysis, Revisited**. Beijing, China: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, 2004. p. 45.

COUSOT, P.; COUSOT, R. **Abstract interpretation**: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. [S.l.]: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, 1977.

COVERITY. **Coverity Code Advisor**. [S.l.: s.n.], 2015. <Disponível em: https://www.coverity.com/products/code-advisor/>. Acesso em: 4 de Outubro de 2015.

CPPCHECK. **CppCheck Features**. [S.l.: s.n.], 2015. <Disponível em: http://www.cppcheck.sourceforge.net/#features>. Acesso em: 4 de Outubro de 2015.

CYTRON, R. et al. **Efficiently Computing Static Single Assignment Form and the Control Dependence Graph**. [S.l.]: ACM TRANSACTIONS ON PROGRAMMING LANGUAGES AND SYSTEMS, 1991. DOI: 10.1145/115372.115320.

DORNELES, R. V.; JR, D. P.; ADAMI, A. G. **ALGOWEB**: a web-based environment for learning introductory programming. Advanced Learning Technologies (ICALT), IEEE 10th International Conference on: IEEE, 2010. p. 83-85.

FOSDICK, L. D.; OSTERWEIL, L. J. **Data Flow Analysis In Software Reliability***. University of Colorado, Boulder, Colorado 80809: ACM Computing Surveys (CSUR),, 1976. v.8, n.3. p. 305-330.

GERMAN, A. **Software Static Code Analysis Lessons Learned**. [S.l.]: QinetiQ, 2003.

GIMPEL. **PC-Lint**. [S.l.: s.n.], 2015. <Disponível em: http://www.gimpel.com/html/lintfaq.htm>. Acesso em: 4 de Outubro de 2015.

GOMES, I. et al. **An overview on the Static Code Analysis approach in Software Development**. Rua Dr. Roberto Frias 4200-465, Porto, Portugal: Faculdade de Engenharia da Universidade do Porto, 2009. ei05021, ei05051, ei05080, pro08007@fe.up.pt.

GOUGH, J.; KLAEREN, H. **Eliminating Range Checks Using Static Single Assignment Form**. Queensland University of Technology: Proc. of the 19th Australian Computer Science Conf., 1996.

HARRISON, W. H. **Compiler analysis of the value ranges for variables**. [S.l.]: Software Engineering, IEEE Transactions on 3, 1977. p 243-250.

HAVLAK, P. H. **Interprocedural symbolic analysis**. Houston, Texas: UMI Microform, 1995.

JOHNSON, S. C. **Lint, a C Program Checker**. Murray Hill, New Jersey 07974: Bell Laboratories, 1978.

KENNEDY, K. **A survey of data flow analysis techniques**. Steven S. Muchnick and Neil D. Jones (eds.): Program Flow Analysis: Theory and Applications, 1981.

KESLEY, R. **A correspondence between continuation passing style and static single assignment form**. [S.l.]: ACM SIGPLAN Notices, 1995. v.30, n.3. p. 13-22.

KNOOP, J.; KOSHCHUTZKI, D.; STEFFEN, B. **Basic-block Graphs**:living dinosaurs. University of Passau: [s.n.], 1997. knoop,koschuet,steffen@fmi.uni-passau.de.

KOSCHKE, R.; GIRAD, J.-F.; WURTHNER, M. **An Intermediate Representation for Integrating Reverse Engineering Analyses**. University of Stuttgart, Germany: Fraunhofer Inst. for Experimental Software Eng., 1998.

LEUPERS, R. **LANCE**: a c compiler platform for embedded processors. Computer Science 12.ed. 44221 Dortmund, Germany: University of Dortmund, 2001.

LINT. **Unix & Linux Forums - Lint**. [S.l.: s.n.], 2015. <Disponível em: `http://www.unix.com/man-page/FreeBSD/1/lint`>. Acesso em: 4 de Outubro de 2015.

MOHNEN, M. **A Graph?Free Approach to Data?Flow Analysis**. Springer Berlin Heidelberg: Compiler Construction, 2002. p. 46-61.

ORACLE. **Java SE 6 Documentation**. [S.l.: s.n.], 2017. <Disponível em: `http://docs.oracle.com/javase/8/docs/`>. Acesso em: 29 de Março de 2018.

ORACLE. **Java SE Overview**. [S.l.: s.n.], 2018. <Disponível em: `http://www.oracle.com/javase/`>. Acesso em: 29 de Março de 2018.

RODRIGUES, R. E.; CAMPOS, V. H. S.; PEREIRA, F. M. Q. **A Fast and Low-Overhead Technique to Secure Programs Against Integer Overflows**. Brasil: Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). IEEE Computer Society, 2013.

SOUZA, C. M. de. **VisuAlg- Ferramenta de Apoio ao Ensino de Programação**. [S.l.]: Revista TECCEN, 2009. v.2, n.2. ISSN 1984-0993.

SREEDHAR, V. C. et al. **Translating Out of Static Single Assignment Form**. Hewlett-Packard Company,11000 Wolfe Road, Cupertino, CA 95014, USA: Performance Delivery Laboratory, 1999.

STAIGER, S. et al. **Interprocedural Static Single Assignment Form**. Reverse Engineering: WCRE 2007. 14th Working Conference, 2007. p. 1-10.

TAYLOR, R. N.; OSTERWEIL, L. J. **Anomaly Detection in Concurrent Software by Static Data Flow Analysis**. [S.l.]: IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, 1980. v.SE-6, n.3. 0098-5589/80/05004-265 00.75 0 1980 IEEE.

TORVALDS, L. **Git**. [S.l.: s.n.], 2018. <Disponível em: `https://git-scm.com/`>. Acesso em: 10 de Fevereiro de 2018.

WEGMAN, M. N.; ZADECK, F. K. **Constant Propagation with Conditional Branches**. 1101 Route 134 Kitchawan Rd, Yorktown Heights, NY 10598, Estados Unidos: IBM T. J. Watson Research Center, 1995.

XU, Z. **SAFETY-CHECKING OF MACHINE CODE**. Madison, WI 53706, Estados Unidos: University of Wisconsin, 2001.

ZADECK, F. K. **An Efficient Algorithm for Incremental Data-Flow Analysis**. Brown University: [s.n.], 1988.