UNIVERSIDADE FEDERAL DO AMAZONAS
INSTITUTO DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

TÉCNICAS DE CONTRAÇÃO DE DOMÍNIO DE VARIÁVEIS EM VERIFICAÇÃO FORMAL E DETECÇÃO DE VULNERABILIDADES DE SOFTWARE USANDO PROGRAMAÇÃO POR RESTRIÇÕES E ARITMÉTICA INTERVALAR

Jessé de Souza Deveza

Agosto de 2024

Manaus - AM

UNIVERSIDADE FEDERAL DO AMAZONAS
INSTITUTO DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

TÉCNICAS DE CONTRAÇÃO DE DOMÍNIO DE VARIÁVEIS EM VERIFICAÇÃO FORMAL E DETECÇÃO DE VULNERABILIDADES DE SOFTWARE USANDO PROGRAMAÇÃO POR RESTRIÇÕES E ARITMÉTICA INTERVALAR

Jessé de Souza Deveza

Dissertação de Mestrado apresentado ao Programa de Pós-Graduação em Informática do Instituto de Computação Universidade Federal do Amazonas, como parte dos requisitos necessários à obtenção do título de Mestre em Informática.

Orientador(a): Rosiane de Freitas Rodrigues, Ph.D.
Co-orientador(a): Lucas Carvalho Cordeiro, Ph.D.

Agosto de 2024
Manaus - AM

Ficha Catalográfica

Ficha catalográfica elaborada automaticamente de acordo com os dados fornecidos pelo(a) autor(a).

Deveza, Jessé de Souza
D491t Técnicas de contração de domínio de variáveis em verificação formal e detecção de vulnerabilidades de software usando programação por restrições e aritmética intervalar / Jessé de Souza Deveza . 2024
62 f.: il. color; 31 cm.

Orientadora: Rosiane de Freitas Rodrigues
Coorientador: Lucas Carvalho Cordeiro
Dissertação (Mestrado em Informática) - Universidade Federal do Amazonas.

1. Aritmética Intervalar. 2. ESBMC-Jimple. 3. Programação por Restrições. 4. Verificação Formal de Software. 5. Verificação de Modelo Limitado. I. Rodrigues, Rosiane de Freitas. II. Universidade Federal do Amazonas III. Título

Ministério da Educação
Universidade Federal do Amazonas
Coordenação do Programa de Pós-Graduação em Informática

FOLHA DE APROVAÇÃO

"TÉCNICAS DE CONTRAÇÃO DE DOMÍNIO DE VARIÁVEIS EM VERIFICAÇÃO FORMAL E DETECÇÃO DE VULNERABILIDADES DE SOFTWARE USANDO PROGRAMAÇÃO POR RESTRIÇÕES E ARITMÉTICA INTERVALAR"

JESSE DE SOUZA DEVEZA

DISSERTAÇÃO DE MESTRADO DEFENDIDA E APROVADA PELA BANCA EXAMINADORA CONSTITUÍDA PELOS PROFESSORES:

Prof. Dr. Rosiane de Freitas Rodrigues - PPGI/UFAM - PRESIDENTE

Prof. Dr. Edjard de Souza Mota - MEMBRO INTERNO

Prof. Dr. José Miguel Aroztegui Massera - MEMBRO EXTERNO

Dr. Eddie Batista de Lima Filho - MEMBRO EXTERNO

MANAUS, 27 de março de 2024.

Documento assinado eletronicamente por **Rosiane de Freitas Rodrigues**, **Professor do Magistério Superior**, em 22/08/2024, às 20:25, conforme horário oficial de Manaus, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Edjard de Souza Mota**, **Professor do Magistério Superior**, em 23/08/2024, às 21:23, conforme horário oficial de Manaus, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Eddie Batista de Lima Filho**, **Usuário Externo**, em 08/11/2024, às 15:12, conforme horário oficial de Manaus, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **José Miguel Aroztegui Massera**, **Usuário Externo**, em 09/11/2024, às 10:14, conforme horário oficial de Manaus, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Maria do Perpétuo Socorro Vasconcelos Palheta**, **Secretária**, em 11/11/2024, às 18:26, conforme horário oficial de Manaus, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site https://sei.ufam.edu.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **2203940** e o código CRC **1E8CE579**.

Avenida General Rodrigo Octávio, 6200 - Bairro Coroado I Campus Universitário Senador Arthur Virgílio Filho, Setor Norte - Telefone: (92) 3305-1181 / Ramal 1193
CEP 69080-900, Manaus/AM, coordenadorppgi@icomp.ufam.edu.br

Dedicatória

Dedico este trabalho a meu avô José Deveza e minha avó Marta da Rocha que sempre acreditaram em mim e hoje moram com Deus.

Agradecimentos

Primeiramente, agradeço a Deus por me proporcionar a vida e por sempre cuidar de mim. Agradeço também à minha orientadora, Dra. Rosiane de Freitas, por sua paciência e apoio contínuo ao longo deste processo. Seus conhecimentos foram inestimáveis para o desenvolvimento deste trabalho. Do mesmo modo, agradeço ao meu co-orientador Dr. Lucas Cordeiros, por seus valiosos ensinamentos e ideias que muito me ajudaram nesse processo.

Também gostaria de agradecer ao membro da banca examinadora, Dr. José Miguel Aroztegui Massera por fornecer valiosas sugestões e comentários junto ao seu acompanhamento da pesquisa. Bem como, agradeço ao doutorando Rafael de Sá Menezes por sua inestimável ajuda em todo o processo desta pesquisa.

Além disso, sou grato aos meus colegas de laboratório e amigos Thailson Clementino e Lia Martins e a todos meus estimados colegas do grupo de pesquisa Algox e SW Perfi, pela colaboração, discussões estimulantes e apoio mútuo ao longo deste processo.

Não poderia deixar de agradecer também à minha namorada Nelcilane Nogueira da Silva por seu apoio e companheirismo durante toda caminhada. Sempre me motivando nos momentos mais difíceis da minha vida pessoal.

Por fim, expresso minha gratidão a minha família. Meus pais, Vilane e Josemar. Meus irmãos, Jade e Lucas. Bem como, minha sogra Maria Inelcy e a todos meus queridos familiares e amigos que sempre acreditaram no meu potencial.

"Matemática, Ciências
História e o mistério
Que começou com o big bang."

The Big Bang Theory Theme.

TÉCNICAS DE CONTRAÇÃO DE DOMÍNIO DE VARIÁVEIS EM VERIFICAÇÃO FORMAL E DETECÇÃO DE VULNERABILIDADES DE SOFTWARE USANDO PROGRAMAÇÃO POR RESTRIÇÕES E ARITMÉTICA INTERVALAR

Jessé de Souza Deveza
Agosto/2024

Orientador(a): Rosiane de Freitas Rodrigues

Co-orientador(a): Lucas Carvalho Cordeiro

Nesta pesquisa são investigadas técnicas para resolução de problemas formulados para satisfazer um conjunto de restrições (*Constraint Satisfaction Problems - CSP*), modelados como expressões lógicas booleanas e cujas variáveis podem assumir valores reais, representando intervalos sobre os quais se aplica operações de aritmética intervalar e técnicas de programação por restrições (*Constraint Programming - CP*), para interseção, composição e contração de intervalos. Especificamente, o contexto de verificação formal de software é utilizado para a aplicação de redução do domínio de variáveis no pré-processamento de programas que usam verificação formal para provar por meio das teorias de módulo da satisfatibilidade (*Satisfiability Modulo Theories - SMT*) a satisfabilidade ou não da fórmula booleana que representa todos os estados do programa. Foi utilizado, portanto, o método formal de verificação de modelo limitado (*Bounded Model Checking - BMC*), que verifica em k passos se determinada propriedade é satisfatível ou, caso contrário, provê um contra-exemplo. Uma estratégia de pré-processamento BMC usando os programas modelados por CSP/CP e com soluções estimadas por uma técnica algorítmica de contração de intervalos é apresentada. Experimentos computacionais foram realizados usando benchmarks de programas em linguagem de programação C, Java e Kotlin, e utilizando o verificador ESBMC (e ESBMC-Jimple, desenvolvido no Projeto de PD&I SWPERFI UFAM-MOTOROLA, do qual esta pesquisa também faz parte) e outros. Os resultados indicam que, em muitos casos, há redução significativa do domínio das variáveis, influenciando na redução do espaço de busca exponencial da verificação de estados que, consequentemente, promove a diminuição do tempo de verificação dos programas, mostrando a adequabilidade da estratégia proposta. Além disso, o método também foi empregado na detecção de vulnerabilidades de softwares, especificamente naquelas que podem envolver operações com intervalos aritméticos, alcançando resultados satisfatórios para a maioria das categorias de vulnerabilidades utilizadas.

Palavras-chave: Aritmética Intervalar, ESBMC-Jimple, Programação por Restrições, Verificação Formal de Software, Verificação de Modelo Limitado, Vulnerabilidades.

Jessé de Souza Deveza

August/2024

Advisor: Rosiane de Freitas Rodrigues
Lucas Carvalho Cordeiro

This research investigates techniques for solving problems formulated to satisfy a set of constraints (Constraint Satisfaction Problems - CSP), modeled as Boolean logical expressions and whose variables can assume real values, representing intervals over which interval arithmetic operations and techniques of constraint programming (Constraint Programming - CP) are applied , for intersection, composition and contraction of intervals. Specifically, the context of formal software verification is used to get variable domain reduction in the preprocessing of programs by formal verification to prove, through Satisfiability Modulo Theories (SMT), the satisfiability or not from the Boolean formula that represents all states of the program. Therefore, it was used. the formal method of bounded model checking (Bounded Model Checking - BMC), which checks in k steps whether a given property is satisfiable or, if not, provides a counterexample. A BMC preprocessing strategy, using programs modeled by CSP/CP and with solutions estimated by an interval contraction algorithmic technique, is presented. Computational experiments were carried out using program benchmarks in C, Java and Kotlin programming languages, and using the ESBMC solver (and ESBMC-Jimple, developed in the SWPERFI UFAM-MOTOROLA PD&I Project, of which this research is also part) and others. The results indicate that, in many cases, there is a significant reduction in the domain of variables, influencing the reduction of the exponential search space for state verification which, consequently, promotes a reduction in program verification time, showing the suitability of the proposed strategy. Furthermore, the method was also used to detect software vulnerabilities, specifically those that may involve operations with arithmetic intervals, achieving satisfactory results for most of the vulnerability categories used.

Keywords: Bounded Model Checking, Constraint Programming, ESBMC-Jimple, Formal Software Verification, Interval Arithmetic, Vulnerabilities.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Com a velocidade do avanço tecnológico, o desenvolvimento de sistemas de software para diferentes áreas exige cada vez mais mão de obra [Cordeiro et al. 2020]. Na indústria, ter a certificação de segurança e funcionalidade de tais sistemas torna-se uma demanda fundamental contra problemas de falhas de segurança, de integridade e vazamentos de memória [Clarke et al. 2018]. Nesse contexto, a área da ciência da Computação conhecida por **raciocínio automático**, que se preocupa em aplicar o raciocínio na forma de lógica a sistemas de computação, pode usar tais métodos nos processos de **verificação formal de software** [Alhawi et al. 2021]. Assim, dado um conjunto de suposições e um objetivo (ou propriedades a serem validadas), um sistema de raciocínio automatizado deve ser capaz de fazer inferências lógicas para esse objetivo automaticamente. Desse modo, ao fazer uso de tais procedimentos lógicos, modelos formais de verificação são construídos para a verificação de programas [Monteiro et al. 2022]. Neste trabalho, é usada a verificação conhecida como **Verificação de Modelo Limitado** *(Bounded Model Checking - BMC)*, a qual é uma estratégia para lidar com o problema da explosão de estados, o qual consiste no crescimento exponencial do tamanho do espaço de estado, na medida que o número de variáveis de estado no sistema aumenta [Clarke et al. 2012]. Para esta pesquisa, foi usado o modelo de verificação para aplicar uma técnica que, de certo modo, também colabora no tratamento do problema da explosão de estados, fazendo uma redução dos domínios das variáveis de entrada fazendo-se de técnicas de Programação por Restrições, aritmética de intervalos e algoritmos contratores.

Esta abordagem é uma extensão do trabalho realizado em programas C por Mohan-

nad Aldughaim [Aldughaim et al. 2020], o qual já tem resultados concretos de efetividade desta técnica em seus experimentos realizados em programas na linguagem C, incluindo sua automação no verificador ESBMC [Cordeiro et al. 2012, Gadelha et al. 2018]. Assim, realizaram-se outras aplicações com o auxílio deste método, incluindo programas em linguagem Kotlin, Java, além de C, para critério de comparação. Promoveu-se, também, uma abordagem voltada às vulnerabilidades comuns de software disponíveis na literatura, especificamente atuando nos exemplos disponíveis na enumeração de fraquezas comuns CWES (*Common Weakness Enumeration*) [Tihanyi et al. 2024, Tihanyi et al. 2023].

1.1. Contextualização

Entre os anos de 2018 e 2919 ocorreu a chamada ***Falha no Software do Boeing 737 MAX***, que foram acidentes fatais envolvendo o *Boeing 737 MAX*, resultando na morte de 346 pessoas. A Causa da falha no software do sistema que empurrava o nariz da aeronave para baixo com base em dados incorretos, levando à perda de controle [O Globo 2024].

Desse modo, há sempre a necessidade de uma verificação rigorosa e formal de software em sistemas críticos de aviação, especialmente aqueles que afetam diretamente a segurança. Esses incidentes sublinham a importância da verificação formal em sistemas críticos, onde erros de software podem ter consequências desastrosas. A verificação formal pode ajudar a identificar e corrigir erros antes que eles levem a falhas catastróficas, especialmente em sistemas que envolvem a vida humana ou infraestruturas críticas.

A elaboração de métodos em verificação de software é essencial para garantir o máximo possível de **confiabilidade**, ou seja, entregar todos os serviços especificados de forma segura, sem interrupções [CyBOK 2019].

No BMC, tem-se como principal abordagem a utilização de solucionadores SAT (*Boolean Satisfiability*) para checar as propriedades de um programa. Usando uma estratégia similar, podemos usar os solucionadores SMT (*Satisfiability Modulo Theories*) onde é feito uma série de procedimentos visando a geração de fórmula booleana que representa as propriedades e especificações de programas sequenciais e multi-tarefas para verificar sua satisfabilidade nos solucionadores SMT [Cordeiro et al. 2009, Cordeiro 2010].

Neste trabalho, empregou-se uma técnica que promove uma **contração do espaço de exploração** do modelo de verificação BMC promovendo redução do tempo (e consequentemente da memória) de processamento através da análise de intervalos das variáveis

do programa a ser verificado [Menezes et al. 2024b]. Com auxílio da **Programação por Restrições** *(Constraint Programming - CP)*, um CSP (*Constraint Satisfaction Problem*) [Rossi et al. 2006] pode ser modelado usando as propriedades do programa como restrições e os intervalos das variáveis como domínios, a fim de contrair esses intervalos e introduzi-los (novo domínio reduzido) junto às propriedades no contexto de verificação formal.

Dessa maneira, com algoritmos contratores aplicados em CSP's para fazer a contração dos intervalos das variáveis, a área de busca onde as restrições têm a certificação de serem violadas (propriedades do programa) foram contraídas. Usando a mesma estratégia, com o auxílio dos mesmos algoritmos, também contraiu-se a região de intervalo das variáveis em uma área onde as propriedades têm segurança de serem mantidas. Restando apenas uma região de valores limites, que não afeta o fornecimento dos contra-exemplos na verificação BMC, como veremos na metodologia de aplicação da técnica.

1.2. Questões de Pesquisa e Objetivos

- Questões de Pesquisa: Com base na análise do contexto técnico científico vigente, duas questões de pesquisa foram propostas nesta investigação.

(1) É possível contribuir com o aprimoramento do processo de verificação formal de software utilizando técnicas de programação por restrições?
(2) É possível identificar vulnerabilidades de software usando técnicas de programação por restrições no pré-processamento da verificação BMC e obter um melhor tempo de processamento?

Os objetivos, geral e específicos, visam responder às questões de pesquisa formuladas.

- Objetivo Geral:

- Propor uma estratégia de pré-processamento na verificação formal de software, aplicando técnicas de programação por restrições e aritmética intervalar de modo a reduzir o domínio de variáveis de interesse dos programas em análise.

- Objetivos Específicos:

- Analisar e definir no processo de verificação formal de software via BMC, qual seria um melhor ponto de aplicação de técnicas de CP;

- Analisar o impacto da aplicação de técnicas de CP em linguagens Kotlin, Java e C;
- Verificar a adequabilidade de aplicação de técnicas de CP para detectar vulnerabilidades clássicas de software mais rapidamente;
- Obter elementos para posterior demonstração formal, completude e corretude, sobre a eficiência da aplicação de técnicas de CP na redução de domínio de variáveis.

1.3. Metodologia de Pesquisa

A proposta de pesquisa em questão é de **natureza aplicada**, baseada em experimentos empíricos realizados onde já está sendo investigada a influência de técnicas de programação por restrições no processo de verificação formal de software com BMC.

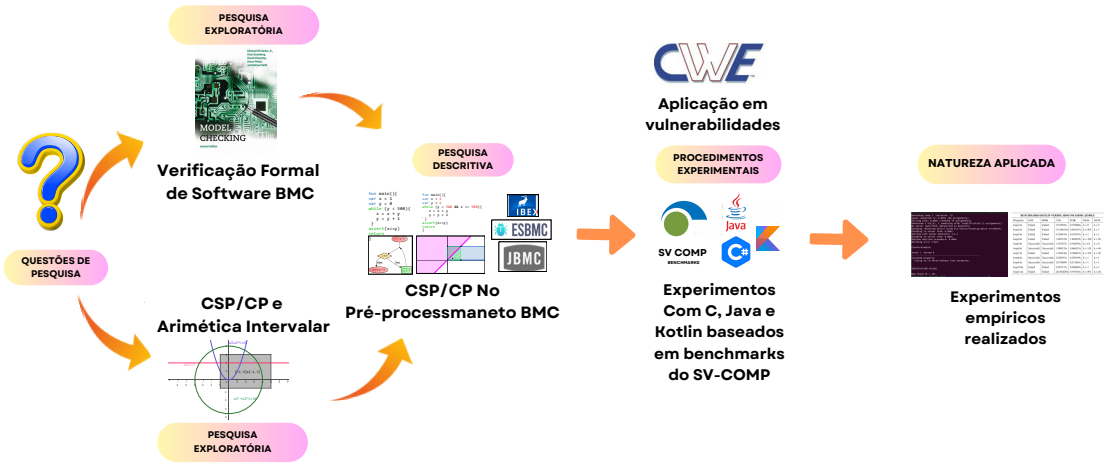


Figura 1.1. metodologia de pesquisa: Inicialmente uma pesquisa exploratória com o embasamento teórico em verificação formal, CP e arimética intervalar, posteriormente, pesquisa descritiva apresentando a estratégia de CSP/CP com BMC, depois, a realização dos experimentos com especificações e vulnerabilidade aplicados em programas Kotlin, Java e C e por fim uma análise dos resultados.

Inicialmente, foi realizada uma **pesquisa exploratória**, visando um levantamento bibliográfico, onde foram estudados conteúdos relacionados à verificação formal de software e otimização com a programação por restrições. Elaborou-se uma **análise quali-quantitativa** para se verificar a influência da aplicação das técnicas de programação por restrições no processo de verificação formal por meio de **experimentos computacionais**

envolvendo as linguagens de programação C, Java e Kotlin. Além de uma **análise descritiva** sobre a aplicação da técnica de contração de intervalos com o uso das definições de CSP para reduzir os domínios das variáveis envolvidas no programa a ser verificado. Os procedimentos envolvem o pré-processamento da verificação com BMC, onde se sucedeu à análise de intervalo das variáveis de entrada envolvidas no programa e sua atuação sobre as propriedades envolvidas. Além disso, a técnica foi aplicada em uma vulnerabilidade comum de software, tendo como base exemplos em C disponível no CWE (Common Weakness Enumeration), onde cada fraqueza foi considerada como uma restrição a ser analisada.

Tomando como base a aplicação do método em programas C, Java e Kotlin, bem como a aplicação em vulnerabilidades, comprovou-se a efetividade da técnica e suas implicações no processo de verificação BMC. A **Figura 1.1** mostra o esquema da metodologia de pesquisa deste trabalho.

1.4. Contribuições da Pesquisa

A pesquisa contribui significativamente para o campo de Verificação Formal, aplicando a programação por restrições ao integrar técnicas de resolução de problemas CSP/CP, oferecemos uma nova perspectiva sobre como lidar com a complexidade dos sistemas de software, especialmente em termos de tempo e profundidade de exploração do espaço de busca. Essa abordagem amplia o escopo da aplicação de técnicas CSP/CP e aprimora a precisão e eficiência dos métodos de verificação formal, contribuindo para avanços significativos na garantia de qualidade de software.

As contribuições desta pesquisa incluem uma estratégia eficaz de pré-processamento para o BMC, que demonstrou a capacidade de reduzir significativamente o tempo de verificação de programas e a complexidade do espaço de busca exponencial. Além disso, o método apresentado oferece uma maneira para lidar com o problema da explosão de estados, visto que a contração dos intervalos de entrada promove uma menor quantidade de estados alcançáveis. Experimentos computacionais envolvendo programas em linguagens C, Java e Kotlin revelaram resultados promissores, indicando uma redução substancial no domínio das variáveis e a aplicabilidade prática da nossa abordagem. Essas contribuições não apenas avançam o conhecimento teórico no campo da verificação formal, mas também têm implicações práticas significativas para o desenvolvimento de software seguro e

confiável.

Organização da Continuidade de Dissertação: O restante deste trabalho está estruturado da seguinte maneira: No Capítulo 2, é apresentado a fundamentação teórica deste trabalho, com as definições para o devido entendimento do mesmo. Assuntos como Programação por Restrições, Problemas de Satisfação por Restrições, Aritmética Intervalar e Verificação de Modelo Limitado são tratados nesse capítulo. No Capítulo 3 é explanada a técnica de contração de domínios das variáveis com CSP/CP via BMC com os devidos passos a serem seguidos, apresentando um exemplo prático para o melhor entendimento do método. Além da descrição de trabalhos relacionados, pesquisas que contêm técnicas de programação por restrições com algoritmos de filtragem (contratores) e CSP para redução de domínios, verificação formal de programas e o uso da verificação formal de software com essas técnicas. No capítulo 4 é apresentado o projeto de experimentos usado na pesquisa, destacando a seleção de técnicas e ferramentas, as linguagens de programação utilizadas e os tipos de programas empregados para avaliar o desempenho e a eficácia do método proposto. No Capítulo 5, os resultados da aplicação do método em programas escritos nas linguagens Kotlin, Java e C que contêm propriedades de verificação mostram a efetividade da técnica nos casos abordados. Além disso, é explorada a aplicação do método em vulnerabilidades comuns de software, onde foram usadas como restrições as próprias fraquezas que o programa vulnerável contém. Finalmente, no Capítulo 6, as considerações finais com as expectativas de continuidade e desenvolvimento desta pesquisa são apresentadas.

Capítulo 2

Fundamentação Teórica

Neste capítulo, as definições e conceitos importantes para a abordagem deste trabalho são apresentadas, bem como, os Problemas e Técnicas de Satisfação por Restrições, com as definições de CSP/CP [Rossi et al. 2006], além de introduzir os conceitos de CSP's em números reais com a matemática intervalar [Mustafa et al. 2018] e algoritmos contratores [Jaulin et al. 2001a]. Do mesmo modo, são apresentados os principais conceitos da Teoria e Técnicas para Verificação Formal de Software, bem como a Verificação de Modelos Limitado [Clarke et al. 2012] e as definições e descrições das categorias de Vulnerabilidades de Software [CWE] presente na pesquisa.

2.1. Problemas e Técnicas de Satisfação de Restrições

Programação por Restrições (*Constraint Programming* - CP), é uma técnica de modelagem e programação matemática para a representação de problemas de otimização combinatória [Rossi et al. 2006]. A mesma é formada pela união de dois paradigmas computacionais, a programação lógica e a resolução de restrições. O modelo de Programação por Restrições pode ser representado por um conjunto de variáveis onde cada uma delas tem um domínio finito, um conjunto de restrições e uma função objetivo.

O uso da Programação por Restrições pode ser entendido por sua utilidade em resolver problemas representados por um conjunto de restrições a serem satisfeitas, os chamados **Problemas de Satisfação por Restrições** (*Constraint Satisfaction Problem* - CSP), um CSP pode ser definido como:

Definição 1. *Um **Problema de Satisfação por Restrições CSP** P pode ser escrito como uma trupla $P = \langle \mathcal{X}, \mathcal{D}, \mathcal{F} \rangle$, onde $\mathcal{X} = \{x_1, ..., x_n\}$ é um conjunto com n variáveis, $\mathcal{D} = \{D_1, ..., D_n\}$ é um conjunto com n elementos que correspondem aos domínios de cada variável em $\mathcal{X}$, $\mathcal{F} = \{f_1, ..., f_t\}$ são t-restrições f_i que um CSP pode ter. Uma restrição $f_i \in \mathcal{F}$ pode ser definida como $f_i : D_1 \times ... \times D_n \rightarrow \{V(verdadeiro), F(Falso)\}$, onde $i \in [1, .., t]$, ou seja, uma condição que pode ou não ser validada usando os valores de domínios das variáveis envolvidas. Deste modo, uma solução $\mathcal{S}$ de um CSP deve ter os valores $\{d_1, ..., d_n \mid d_1 \in D_1, ..., d_n \in D_n\}$ de cada domínio em $\mathcal{D}$ que **satisfazem** todas as t-restrições:*

$$\mathcal{S} = \{d_1, ..., d_n \mid f_i : D_1 \times ... \times D_n \rightarrow V(verdadeiro), i \in [1, .., t]\} \tag{2.1}$$

Como exemplo, considere um CSP P no contexto de análise de regiões intervalares, nessa situação, as variáveis podem assumir qualquer valor dentro de seu intervalo real:

$$\begin{aligned} &\mathcal{X} : x_1, x_2, \\ &\mathcal{D} : D_1 = [-100, 100], D_2 = [-100, 100], \\ &\mathcal{F} : f_1 : x_1 \leq x_2, f_2 : x_2 \leq 30, f_3 : x_1 \geq 0. \end{aligned} \tag{2.2}$$

Para alcançar a solução do CSP os valores de cada domínio em $\mathcal{D}$ deve ser validado pelas restrições em $\mathcal{F}$, retornando assim:

$$\mathcal{S} = \{d_1 \in [-100, 100], d_2 \in [-100, 100] \mid d_1 \leq d_2, d_2 \leq 30, d_1 \geq 0\}. \tag{2.3}$$

Deste modo, percebe-se que obter uma solução em um determinado CSP significa encontrar os valores de entrada (dos domínios das variáveis) que fazem com que as restrições sejam validadas. No caso do exemplo, podemos ter uma noção na **Figura** 2.1 dos valores que satisfazem as restrições, o qual é indicado pela região de coloração rosa. Observe que os valores nesta região satisfazem as três restrições do exemplo.

Existem na literatura diferentes alternativas para se alcançar as soluções de um CSP. A Programação por Restrições utiliza técnicas como *Backtracking*, *Forward Checking*, Busca Local, *Constraint Propagation Search*, entre outros. Nesta pesquisa, foi

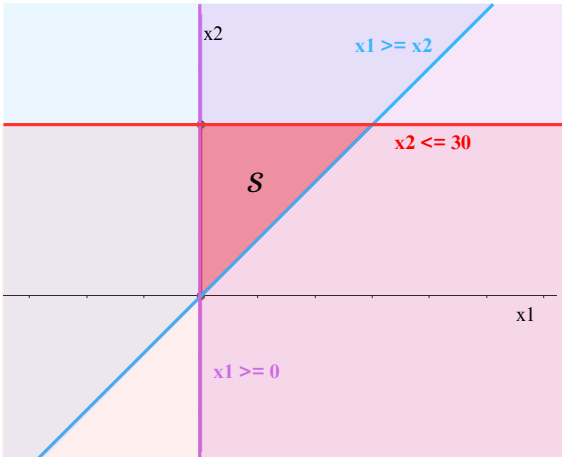


Figura 2.1. Exemplo CSP: As retas em roxo, vermelho e azul indicam as regiões limite de cada uma das três restrições e na região em rosa, triângulo formado pela interseção das regiões, os valores que satisfazem as restrições indicado por $\mathcal{S}$.

usado um método conhecido como "contractor programming", que faz uso de algoritmos contratores, os quais estimam a solução para um CSP contraindo (reduzindo) os domínios iniciais do CSP.

2.1.1. Aritmética Intervalar e CSP/CP

A **Aritmética Intervalar** é uma técnica que estuda as propriedades de funções definidas em intervalos, ao invés de números reais [Moore and Cloud 2009]. Na aritmética intervalar, um intervalo é definido como segue.

Definição 2. *[Sainz et al. 2014] Um **intervalo** é um subconjunto conectado e fechado de $\mathbb{R}$ denotado por {x}. Ele tem um limite inferior $\underline{x}$ e um limite superior $\overline{x}$, respectivamente, onde $\underline{x}, \overline{x} \in \mathbb{R}$. Um intervalo, então, pode ser definido como $[x] = [\underline{x}, \overline{x}] \in \mathbb{IR}$ onde $\mathbb{IR}$ é o conjunto de todos os intervalos de $\mathbb{R}$.*

As operações usuais obedecem às definições do método de intervalo, a adição e multiplicação em um nível básico. Divisão e subtração são descritas como funções de intervalo. Sejam S_{x_1} e S_{x_2} conjuntos e seja $\diamond$ um operador binário, a aritmética desses conjuntos pode ser definida como: $S_{x_1} \diamond S_{x_2} = \{x_1 \diamond x_2 | x_1 \in S_{x_1}, x_2 \in S_{x_2}\}$ [Jaulin et al. 2001b].

Para operações binárias envolvendo conjuntos vazios, o resultado também é um conjunto vazio. Da mesma forma, essas operações podem ser aplicadas a intervalos, ou

seja, a todos $[x], [y] \in \mathbb{IR}$ não vazios. O **Quadro 2.1** contém as definições das operações de adição, subtração, produto e divisão sobre os intervalos $[x] = [\underline{x}, \overline{x}]$ e $[y] = [\underline{y}, \overline{y}]$.

Definição de Operações Binárias com Aritmética Intervalar		
Operação	**Intervalos**	**Resultado**
Adição	$[x] + [y]$	$[\underline{x}, \overline{x}] + [\underline{y}, \overline{y}] = [\underline{x} + \underline{y}, \overline{x} + \overline{y}]$
Subtração	$[x] - [y]$	$[\underline{x}, \overline{x}] - [\underline{y}, \overline{y}] = [\underline{x} - \overline{y}, \overline{x} - \underline{x}]$
Produto	$[x] \times [y]$	$[\min\{\underline{x} \cdot \underline{y}, \underline{x} \cdot \overline{y}, \overline{x} \cdot \underline{y}, \overline{x} \cdot \overline{y}\}, \max\{\underline{x} \cdot \underline{y}, \underline{x} \cdot \overline{y}, \overline{x} \cdot \underline{y}, \overline{x} \cdot \overline{y}\}]$
Divisão	$[x] \div [y]$	$[\min\{\frac{\underline{x}}{\underline{y}}, \frac{\underline{x}}{\overline{y}}, \frac{\overline{x}}{\underline{y}}, \frac{\overline{x}}{\overline{y}}\}, \max\{\frac{\underline{x}}{\underline{y}}, \frac{\underline{x}}{\overline{y}}, \frac{\overline{x}}{\underline{y}}, \frac{\overline{x}}{\overline{y}}\}],\ 0 \notin [\underline{y}, \overline{y}]$

Tabela 2.1. Quadro de operações binárias sobre intervalos.

A operação de divisão sobre intervalos, no caso geral, é definido como:

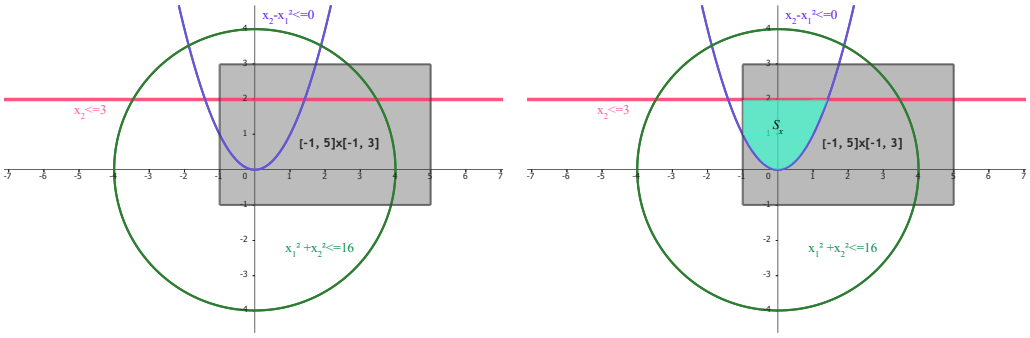
$$[x] \div [y] = [x] \cdot \frac{1}{[y]} = [\underline{x}, \overline{x}] \cdot \frac{1}{[\underline{y}, \overline{y}]}. \tag{2.4}$$

$onde$:

$$\frac{1}{[y]} = \begin{cases} \varnothing & \text{se } [y] = [0, 0], \\ [\frac{1}{\overline{y}}, \frac{1}{\underline{y}}] & \text{se } 0 \notin [\underline{y}, \overline{y}], \\ [\frac{1}{\overline{y}}, +\infty[& \text{se } \underline{y} = 0 \; e \; \overline{y} > 0, \\ ]-\infty, \frac{1}{\underline{y}}] & \text{se } \overline{y} = 0 \; e \; \underline{y} < 0, \\ ]-\infty, +\infty[& \text{se } \underline{y} < 0 < \overline{y}. \end{cases} \tag{2.5}$$

Desse modo, com o entendimento sobre intervalos e de operações básicas junto à Aritmética Intervalar, podemos introduzir os conceitos de CSP/CP em tal domínio. Assim, um CSP explorado no contexto dos números reais, ou seja, na aritmética intervalar, pode ser definido como:

Definição 3. *Um **problema de satisfação por restrição (CSP)**, no contexto da **análise de intervalo**, é definido como o triplo $\langle x, [x], F \rangle$, onde $x = \{x_1, ..., x_n\}$ é um conjunto com n variáveis, $[x] = [x_1] \times ... \times [x_n]$ é a caixa que representa o produto cartesiano entre os domínios (intervalos) de cada variável de x e F representa as restrições do CSP tais*



(a) (b)

Figura 2.2. Na figura (a): Gráfico com as regiões limitadas pelas restrições e caixa em cinza. Na figura (b): Gráfico contendo a região de solução em cor ciano da caixa.

que $F = \{f_1(x) \leq 0, ..., f_t(x) \leq 0\}$*, ou seja, t-restrições que um CSP pode ter. Quando todas as restrições são atendidas, se tem um conjunto de soluções* S_x *do CSP, definido como:* $S_x = \{(x_1, ..., x_n) \in [x] \mid F\}$.

É importante destacar que as restrições são escritas em termos de desigualdade e, no caso de uma igualdade, a restrição pode ser reescrita como duas desigualdades. Por exemplo, a igualdade $a = b$ pode ser reescrita como $a \leq b$ e $b \leq a$.

Como exemplo, sejam x_1 e x_2 duas variáveis com seus respectivos domínios intervalares $[x_1] = [-1, 5]$ e $[x_2] = [-1, 3]$. As restrições deste problema são: $F = \{f_1(x) : x_1^2 + x_2^2 \leq 16, \quad f_2(x) : x_2 - x_1^2 \leq 0, \quad f_3(x) : x_2 \leq 3\}$. Como no exemplo anterior, a solução deve obedecer a todas as restrições, neste caso, todas as restrições de f. Isso significa ter somente os valores dos domínios que satisfazem f_1, f_2 e f_3. Assim, temos como conjunto solução:

$$S_x = \{(x_1, x_2) \in [-1, 5] \times [-1, 3] \mid f_1, f_2, f_3\}. \tag{2.6}$$

A **Figura** 2.2 apresenta o esquema do problema de satisfação por restrições com o gráfico (a) contendo os limites de regiões das restrições indicadas com verde para f_1, roxo para f_2 e rosa para f_3. Os intervalos (caixa), que são os domínios das variáveis indicadas na cor cinza. No gráfico (b), uma região a mais, na cor ciano, representando as soluções de S_x, ou seja, a região da caixa onde atende todas as restrições do problema.

2.1.2. Contração de Intervalos em CSP

Os métodos de contração de intervalos são frutos da aplicação combinada de técnicas de programação por restrições e aritmética intervalar, provendo boas estimativas para as soluções de um CSP, ou seja, realiza métodos de aproximações das soluções diminuindo o conjunto de entrada sem violar as restrições [Chabert and Jaulin 2009]. Os métodos são conhecidos na literatura por programação por contração (*contractor programming* [Chabert and Jaulin 2009]).

Definição 4. *[Jaulin et al. 2001a] Um* ***contrator*** *é um método de intervalo que estima a solução de um determinado CSP com o mapa* $C : \mathbb{IR}^n \mapsto \mathbb{IR}^n$*. Seja* S_x *o conjunto da solução e* C *um contrator, que é aplicado à caixa* $[x]$ *então* $C([x]) \subseteq [x]$ *e* $C([x]) \cap S_x = [x] \cap S_x$*. O primeiro satisfaz a condição de contração e o último satisfaz a condição de correção.*

Em outras palavras, um contrator aplicado a um CSP é um operador usado para reduzir seu domínio sem violar a restrição. No exemplo da **Figura 2.3** um contrator C (em cinza escuro) é aplicado em uma caixa inicial $[x]$ (em cinza claro), note que a região onde a restrição se mantém S_x (em azul) continua fazendo a mesma interseção com os domínios reduzidos com o contrator após a contração.

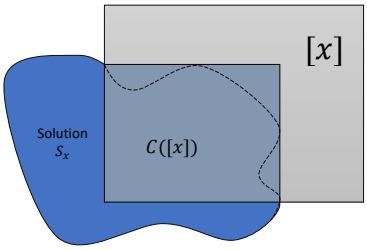


Figura 2.3. Exemplo do desempenho de um contrator.

Na pesquisa, foi usado o algoritmo *Forward-Backward contractor* que é um contrator baseado na propagação de restrição. Esse contrator possibilita contrair os domínios do CSP levando em consideração qualquer uma das restrições **isoladamente**, ou seja, usando apenas uma restrição do conjunto de restrições por vez, realizando propagação para frente e propagação para trás. Nessa técnica há dois tipos de contratores, o interno C_{in} e o externo C_{out}, que são complementos um do outro em termos de restrições. No

contrator externo, usam-se os valores que não violam a restrição e no interno, os valores que violam a restrição, restando uma região limite entre valores que violam e que não violam a propriedade.

O **Algoritmo 1** descreve os passos desse tipo de contrator, nele a função C_{FB} é composta pelo conjunto de domínio das variáveis [x], uma restrição f e $[I]$ que são valores de intervalo que podem satisfazer a restrição. $[y]$ são os valores após serem avaliados pelo conjunto de restrições, isto é, valores que satisfazem cada restrição, esse passo resultante é dito: *Forward* (linha 2). Em seguida, cada novo domínio $[x_i]$ do conjunto de domínio [x] é determinado realizando um passo inverso sobre a restrição fazendo ${f_{xi}}^{-1}$ (*passo Backward*) , em que se obtém apenas os valores que fazem correspondência com $[y]$ e finaliza até fazer esta ação com todos os conjuntos $[x_i]$ (linhas 3,4,5) . Finalmente, o novo valor de [x] é retornado e finaliza a Função (linhas 6 e 7).

Algoritmo 1: Forward-Backward Contractor C_{FB}

```
1 Função C_FB( [x] , f, [I]) faça
2     [y] = [I] ∩ f([x])
3     para todo [x_i] ∈ [x] faça
4         [x_i] = [x_i] ∩ f_{x_i}^{-1}([x], [y])
5     fim para
6     retorna [x]
7 fim Função
```

Como exemplo da atuação de um contrator que usa os passos do algoritmo descrito, considere um CSP com duas variáveis x_1 e x_2 e domínios $[x_1] = [-100, 100]$ e $[x_2] = [-50, 200]$, respectivamente. Como restrição temos $f1 : x_1 + x_2 \leq 0$, assim o valor de $I = (\infty, 0]$, pois é o intervalo que satisfaz a restrição. Usando os passos do algoritmo:

Passo ***Forward*** $[y] = [I] \cap f([x])$:

$$[y] = [I] \cap ([x_1] + [x_2]) \tag{2.7}$$

$$[y] = [-\infty, 0] \cap ([-100, 100] + [-50, 200])$$
$$[y] = [-150, 0].$$

Passo ***Backward*** $[x_i] = [x_i] \cap {f_{x_i}}^{-1}([x],[y])$ para x_1 e x_2:

$$\begin{array}{cc} [x_1]' = [x_1] \cap ([y] - [x_2]) & [x_2]' = [x_2] \cap ([y] - [x_1]) \\ [\mathbf{x_1}]' = [\mathbf{-100, 50}] & [\mathbf{x_2}]' = [\mathbf{-50, 100}] \end{array} \tag{2.8}$$

Assim, o algoritmo retorna uma nova caixa menor que a caixa inicial (domínios iniciais), ou seja, de $[x] = [-100, 100] \times [-50, 200]$ para uma contraída (domínio final) $[x]' = [-100, 50] \times [-50, 100]$.

O contrator externo C_{out} **Algoritmo 2** realiza o cálculos dos intervalos considerando as restrições como $f([x]) \leq 0$, ou seja, admite que os valores intervalares que satisfazem a restrição estão no intervalo $(-\infty, 0]$. Assim, o intervalo $[I] = (-\infty, 0]$, é assumido para todas as restrições nesse contrator .

Algoritmo 2: Contrator Externo C_{out}

```
1 Função C_out( [x] , F, [I]) faça
2     Para cada f_i ∈ F
3         [x] ← C_FB([x], f_i, (−∞, 0])
4     Fim
5     Retorna [x]
6 Fim Função
```

No contrator interno C_{in} **Algoritmo 3**, as restrições são utilizadas usando o seu complemento, ou seja, $f([x]) > 0$ e assim o intervalo onde é satisfeito essa condição corresponde a $I = [0, \infty)$. Nesse contexto, a região estimada com esse contrator corresponde aos valores que não satisfazem a restrição .

Algoritmo 3: Contrator Interno C_{in}

```
1 Função C_in( [x] , F, [I]) faça
2     Para cada f_i ∈ F
3         [x] ← C_FB([x], f_i, [0, ∞))
4     Fim
5     Retorna [x]
6 Fim Função
```

O acesso ao uso de tais contratores e seus algoritmos pode ser feito usando a biblioteca de programas C/C++ ibex-lib [Chabert and Ninin 2016], nessa biblioteca são

usadas as definições da aritmética intervalar e serve para realizar programação por restrições. Deste modo, foi utilizado o contrator *forward-backward* disponível nessa biblioteca para realizar a contração dos domínios de variáveis nos experimentos nesse trabalho.

2.2. Verificação Formal de Software e a Verificação de Modelo

A **verificação formal** é uma técnica rigorosa para garantir a corretude de software [Monteiro et al. 2018]. Ela utiliza métodos matemáticos para verificar se um programa atende a todas as especificações e propriedades desejadas [Morse et al. 2011]. Ao contrário de métodos de teste convencionais, a verificação formal oferece garantias sólidas de corretude, uma vez que os resultados das verificações são baseados em provas matemáticas. Um método formal automático para verificar sistemas concorrentes de estados finitos chama-se **verificação de modelo** (*Model Checking*). O método tem sido usado com sucesso na prática para verificar projetos complexos de circuitos sequenciais e protocolos de comunicação. O principal desafio na verificação de modelos é lidar com o problema da explosão de estados. Esse problema ocorre em sistemas com muitos componentes que podem interagir entre si ou em sistemas que possuem estruturas de dados que podem assumir muitos valores diferentes (por exemplo, o caminho de dados de um circuito).

Na verificação de modelo, o sistema a ser checado é modelado em um sistema de transição de estados conhecido como estrutura Kripke (Apêndice A). Em uma estrutura Kripke M, um estado é representado por s e uma propriedade do sistema é representado por ϕ. Um algoritmo de verificação de modelo é um procedimento de decisão para M, $s \models \phi$ (s satisfaz ϕ), que é decidir se no estado s a propriedade ϕ é satisfeita ou não [Clarke et al. 2012]. Assim, a estrutura Kripke é generalizada em uma fórmula booleana que usam os operadores da Lógica Temporal (Apêndice B) e para assim obter o problema de decisão, ou seja, verificar se a fórmula é satisfatível [Emerson and Clarke 1982].

Ao automatizar a verificação da fórmula nos conhecidos verificadores formais, significa estar usando um método de verificação de modelo. No entanto, em sistemas complexos, o número de estados alcançáveis cresce exponencialmente conforme aumenta o número de variáveis de estado, tornando inviável a modelagem de todos os estados do sistema. Para contornar esse problema, os verificadores formais utilizam o método de verificação por violação, buscando contra-exemplos que violem a propriedade até um estado limite s_k, como veremos na próxima seção.

2.2.1. Verificação de Modelo Limitado (BMC)

Na verificação de modelo, uma maneira de lidar com problema de explosão de estados se dá através da verificação de modelo limitado (*Bounded Model Checking* - BMC). No BMC procura-se por contra-exemplos em execuções cujo comprimento seja limitado por algum inteiro k. Caso não encontre, incrementa-se k até que o problema se torne intratável ou se tenha atingido o limiar de completude.

Assim, dado um sistema de transição de estados M, um estado k e uma propriedade ϕ que assume o quantificador temporal global, ou seja, $\mathbf{G}\phi$. Uma condição de verificação $\psi(k)$ é um caminho do sistema de transição de estados até S_k e definida pela fórmula booleana:

Definição 5. *Seja ψ uma fórmula booleana que representa um sistema de transição de estados de s_0 até um estado s_k:*

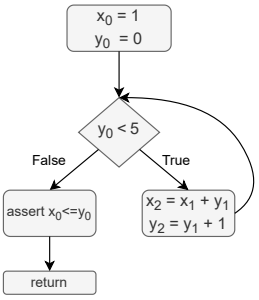
$$\psi(k) = I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \neg\phi(s_k) \tag{2.9}$$

- *$I(s_0)$ garante que o estado s_0 seja um dos estados iniciais.*
- *O conjunto $\neg\phi(s_k)$ afirma que o estado s_k satisfaz $\neg\phi$.*
- *$\bigwedge_{i=0}^{k-1} T(s_i, s_{i+1})$ garante uma transição T de s_i para s_{i+1}, onde $1 \leq i < k$.*

Na verificação de programas usando BMC, a fórmula representa todos os estados que o programa pode alcançar junto com a especificação de verificação, que é a propriedade a ser checada. Quando a fórmula é satisfeita, isso significa que a propriedade foi violada em algum estado alcançável do programa, pois a negação da propriedade foi satisfeita. Como exemplo de construção de fórmula booleana para a verificação formal BMC, observe o trecho de programa escrito em Kotlin da **Figura 2.4(a)**.

No BMC, o programa deve ser modelado como um sistema de transição de estados, derivado de seu grafo de fluxo de controle (*Control-flow Graph* - CFG) [Allen 1970] e depois empregar na forma de atribuição única estática (*Static Single Assignment* - SSA) onde exige que cada variável seja atribuída exatamente uma vez, e cada variável seja definida antes de ser usada [Rastello and Tichadou 2022]. Para o programa do exemplo, sua forma SSA fica conforme a **Figura2.4(b)** e a fórmula booleana:

```
1 fun main(){
2   var x = 1
3   var y = 0
4   while (y < 5){
5      x = x + y
6      y = y + 1
7   }
8   assert(x<=y)
9   return
10 }
```



(a) (b)

Figura 2.4. Na figura (a) temos um programa Koltin a ser verificado via BMC. Na figura (b) temos o CFG do programa na forma SSA.

$$
\begin{aligned}
&(x_0 = 1) \wedge (y_0 = 0) \wedge (x_1 = x_0 + y_0) \wedge (y_1 = y_0 + 1) \\
&\wedge (x_2 = x_1 + y_1) \wedge (y_2 = y_1 + 1) \\
&\wedge (x_3 = x_2 + y_2) \wedge (y_3 = y_2 + 1) \\
&\wedge (x_4 = x_3 + y_3) \wedge (y_4 = y_3 + 1) \\
&\wedge (y_0 < 5) \wedge (y_1 < 5) \wedge (y_2 < 5) \wedge (y_3 < 5) \wedge (y_4 < 5) \\
&\wedge \neg[(x_0 \leq y_0) \wedge (x_1 \leq y_1) \wedge (x_2 \leq y_2) \wedge (x_3 \leq y_3) \wedge (x_4 \leq y_4)].
\end{aligned}
$$

A verificação de modelo limitado BMC baseada em Satisfabilidade Booleana (SAT) foi introduzida como uma técnica complementar aos Diagramas de Decisão Binária (BDDs) para aliviar o problema de explosão de estados [Biere et al. 2009]. No entanto, outra abordagem possível, é usar as Teorias de Módulo de Satisfabilidade (*Satisfiability Modulo Theories - SMT*). Assim, no BMC baseado em SMT, ψ é uma fórmula sem quantificador em um subconjunto decidível da lógica de primeira ordem que deve ser verificada quanto à satisfabilidade por um solver SMT. Uma ferramenta que realiza tal tarefa chama-se ESBMC (Efficient SMT-Based Bounded Model Checker), essa que faz uma tradução precisa de programas escritos em linguagem C para fórmulas sem quantificador usando a lógica SMT-LIB e assim explora as teorias e solucionadores SMT [Cordeiro et al. 2009]. Esse verificador que incluiu a funcionalidade de verificação de códigos em Kotlin, o qual foi usado na demonstração do método usado nesta pesquisa.

2.3. Vulnerabilidades de Software

Uma **vulnerabilidade de software** é uma fraqueza em um sistema de software que pode ser explorada por um invasor para comprometer a segurança do sistema. Essas vulnerabilidades podem ser introduzidas durante o desenvolvimento do software devido a erros

de programação, falhas de design ou problemas de configuração. As vulnerabilidades de software são um alvo comum para ataques cibernéticos e podem levar a uma variedade de consequências negativas.

2.3.1. Tipos de Vulnerabilidades

Na literatura, existem diversos relatos de vulnerabilidades de software, no entanto, o CWE é uma lista abrangente e padronizada de vulnerabilidades de segurança de software [CWE]. É mantida pelo *MITRE Corporation* `www.mitre.org`, uma organização sem fins lucrativos dos Estados Unidos, e é usada para fornecer uma linguagem comum para descrever e categorizar vulnerabilidades de software. A CWE é uma parte essencial do *Common Vulnerabilities and Exposures* (CVE) [CVE], que é um dicionário público de informações de segurança cibernética.

A categoria de **gestão de memória** corresponde as linguagens de programação (C e C++, por exemplo) que não possuem a responsabilidade de atribuir, aceder e desalocar corretamente a memória nas mãos do programador e dizem que o comportamento dos programas que acedem ou gerem incorretamente a memória é indefinido. Estas linguagens são por vezes designadas por linguagens não seguras em termos de memória, e os erros relacionados com a gestão da memória (vulnerabilidades da gestão da memória) são uma fonte notória de erros de segurança nestas linguagens [CyBOK 2019].

Outra categoria vulnerabilidade é a de **geração de resultados estruturados**, os programas têm muitas vezes de construir dinamicamente resultados estruturados que serão depois consumidos por outro programa. Os exemplos incluem: a construção de consultas SQL para serem consumidas por uma base de dados ou a construção de páginas HTML para serem consumidas por um navegador Web. Pode-se pensar no código que gera a saída estruturada como um subcomponente. A estrutura pretendida da saída e a forma como a entrada para o subcomponente deve ser utilizada na saída podem ser consideradas como um contrato ao qual esse subcomponente deve aderir. Uma prática comum de programação insegura consiste em construir esse resultado estruturado através da manipulação de cadeias de caracteres. A saída é construída como uma concatenação de cadeias em que algumas destas cadeias são derivadas (direta ou indiretamente) da entrada do programa. Esta prática é perigosa, porque deixa implícita a estrutura pretendida da cadeia de saída, e os valores maliciosamente escolhidos para as cadeias de entrada podem

fazer com que o programa gere uma saída não pretendida [CyBOK 2019].

A categoria de **condição de corrida** trata da simultaneidade em programas que acessam recursos compartilhados, como memória, arquivos ou bases de dados, exige que o programa faça suposições sobre as ações de outros atores concorrentes (como outras threads ou processos). Essas suposições podem ser vistas como um contrato entre o programa e seu ambiente, especificando como o ambiente deve interagir com os recursos do programa. Por exemplo, um programa pode depender de acesso exclusivo a certos recursos durante um período específico de execução. Violações dessas suposições resultam em erros de simultaneidade, conhecidos como condições de corrida, onde o comportamento do programa depende de qual ator acessa o recurso [Cyb].

Dentro da área de verificação formal, essas são as principais categorias de vulnerabilidades que são tratadas, valendo apena recorrer as categorias disponíveis no CWE [Branescu et al. 2024] ou mesmo pelo cybook, que é uma iniciativa destinada a fornecer um quadro de referência abrangente para o conhecimento fundamental em cibersegurança [CyBOK 2019].

Capítulo 3

BMC usando o Método de Contração de Intervalos Com CSP/CP

A técnica de contração de intervalos faz uso das definições de CSP para reduzir os domínios das variáveis envolvidas no programa a ser verificado. A **Figura 3.1** mostra o ponto da atuação dessa técnica, no pré-processamento da verificação BMC.

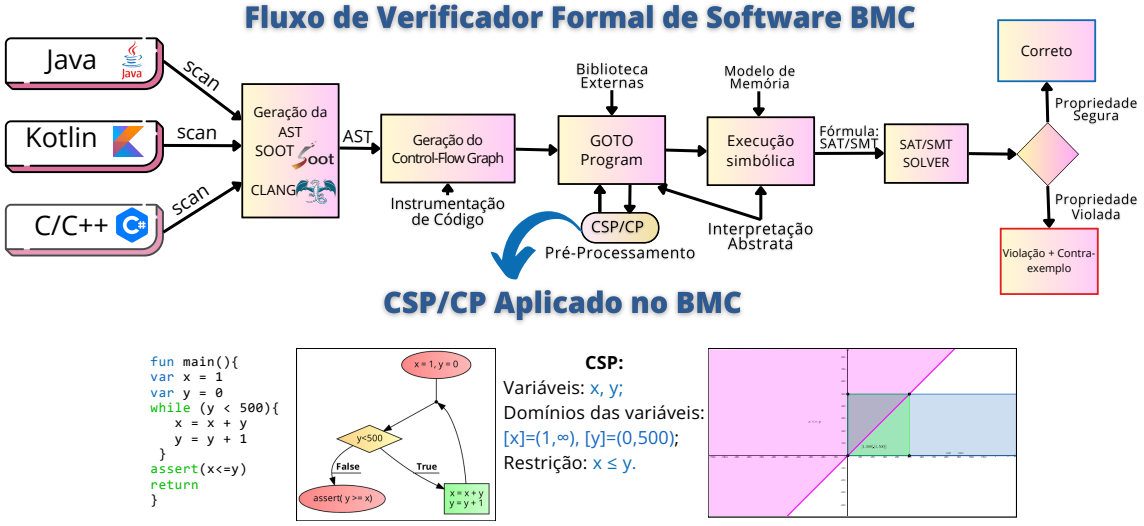


Figura 3.1. Processo básico da verificação de programas usando um verificado automático BMC - AST: Árvore de Sintaxe Abstrata (*Abstract Syntax Tree*). CFG: Grafo de Fluxo de Controle (*Control-flow Graph*). GOTO Program: Converte o programa ANSI-C em um programa GOTO (por exemplo, substituição de switch e while por instruções if e goto).

O método consiste em modelar as restrições e propriedades de um programa em um CSP e, deste modo, estimar a solução por meio de um algoritmo de contração de domínios intervalares. Os domínios do CSP são construídos analisando as declarações de variáveis independentes e atribuições de variáveis. As funções *assert* que representam as propriedades testadas no programa são as restrições e também indicam quais variáveis terão seus domínios envolvidos e gerados para a aplicação dos algoritmos contratores. Nesse trabalho, todo o processo de análise e conversão para CSP dos programas foram realizadas de forma manual, ou seja, sem a automação dos processos descritos, apenas analisando as propriedades e restrições do programa direto de seu código-fonte, foram usados benchmarks em linguagem Kotlin e C para a validação desse método.

Posteriormente, com a biblioteca IBEX [Chabert and Ninin 2016], foram aplicados os algoritmos contratores *forward-backward*: externo e interno a fim de estimar a solução do CSP e reduzir consequentemente os intervalos das variáveis envolvidas. A **Figura 3.2** contém o fluxograma dos passos para alcançar a redução dos domínios das variáveis, até a inclusão dos novos domínios estimados pelos algoritmos contratores no programa.

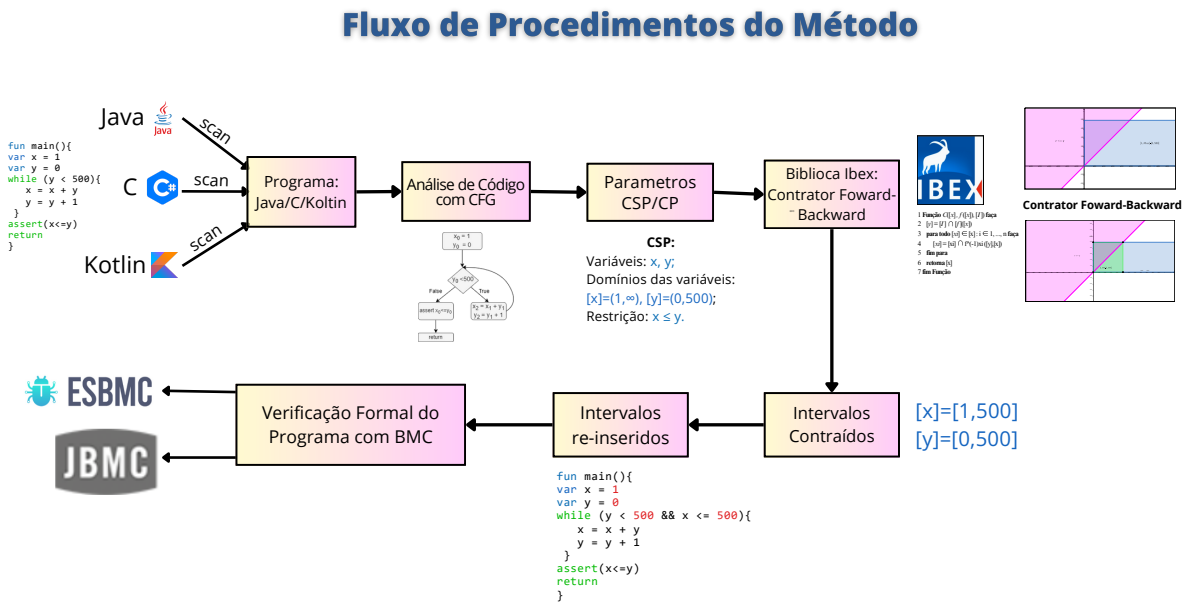


Figura 3.2. Procedimentos do método de CSP/CP para a verificação formal de software BMC.

3.1. Exemplo de Aplicação do Método

Um exemplo de aplicação da técnica pode ser vista para o programa da **Figura 3.3**, o mesmo contém sua versão nas linguagens Koltin, Java e C, esse programa contém declarações de duas variáveis x e y, em seguida, há um loop onde é realizada operações envolvendo as variáveis, ao fim de todas as iterações do loop há uma função assertiva que compara os valores finais das variáveis.

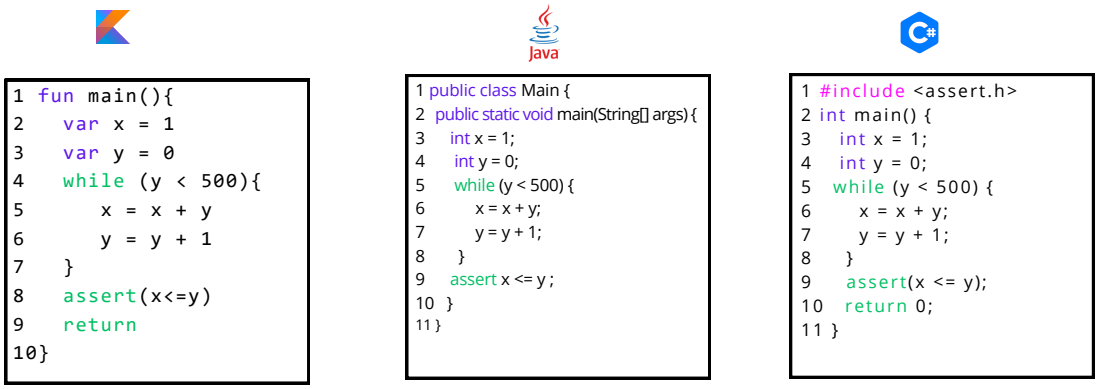


Figura 3.3. Programa com intervalos iniciais x = [1, Max_{int}] e y = [0, 500].

Como primeiro passo, um CSP baseado nas declarações de variáveis envolvidas e nas condições de propriedades de verificação (assert) é construído. Note que as variáveis envolvidas nesse caso, são as que estão relacionadas com a função $assert$, a qual também define a restrição $(x <= y)$. Com relação aos domínios das variáveis, os valores inicialmente possíveis de cada variável são analisados, note que para o valor de y o intervalo inicial é $[y] = [0, 500]$, para saber o valor de x é preciso realizar as devidas iterações no loop, mas isto não é viável no contexto da verificação formal, pois tal procedimento será realizado durante a verificação BMC, assim, o maior valor possível é definido, ou seja, $[x] = [1, Max_{int}]$.

Em seguida, com o auxílio da biblioteca IBEX, o contrator *forward-backward* que teve como entrada o CSP do programa é aplicado. Inicialmente, o contrator externo é aplicado usando a restrição $x <= y$ onde se obtém os intervalos: $[x] = [1, 500]$ e $[y] = [1, 500]$, contendo apenas valores intervalares que satisfazem a restrição. Depois, o contrator interno é aplicado usando a restrição como $x > y$ resultando em $[x] = [1, Max_{int}]$ e $[y] = [0, 500]$. Desse modo, duas caixas definidas pelos contratores interno e externo

são geradas: $[1, 500] \times [1, 500]$ e $[1, Max_{int}] \times [0, 500]$, respectivamente. Dessa forma, para obter a região limite com valores que satisfazem e que não satisfazem a restrição é usado a interseção: $[1, 500] \times [1, 500] \cap [1, max_{int}] \times [0, 500] = [\mathbf{1, 500}] \times [\mathbf{1, 500}]$, que será o valor a ser analisado para o contexto de verificação. Finalmente, para a re-inserção dos novos valores intervalares, o programa é analisado e os intervalos inseridos mantendo a semântica do programa. Para este exemplo, não o valor inferior da variável y ($y = [\mathbf{1}, 500]$) do programa não foi o valor usado, pois as variáveis serão usadas no desenrolar do loop e isto modificaria seu contexto. Com isto, observe na Figura 3.4 que apenas foi inserido uma restrição $x <= 500$, mantendo a conformidade do programa, ou seja, com intervalos finais $[x] = [\mathbf{1, 500}]$ e $[y] = [\mathbf{0, 500}]$.

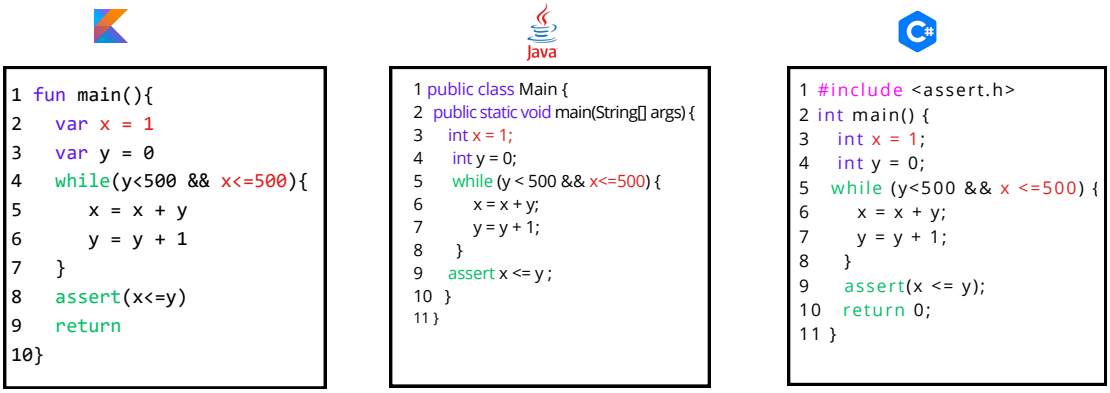
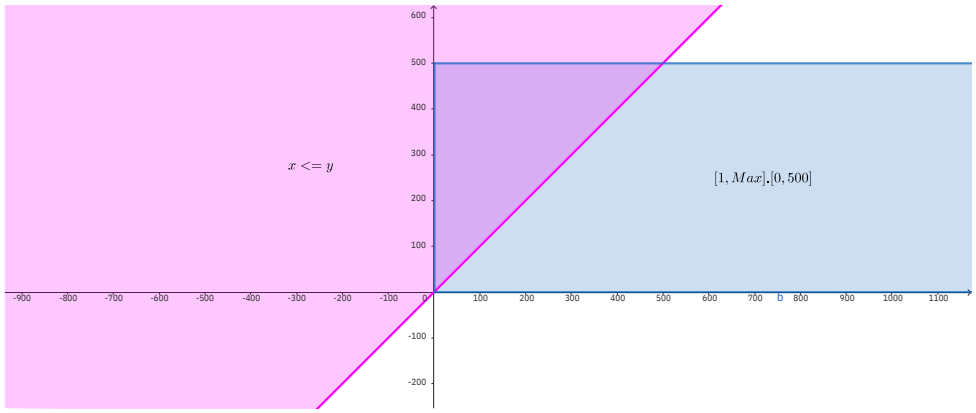
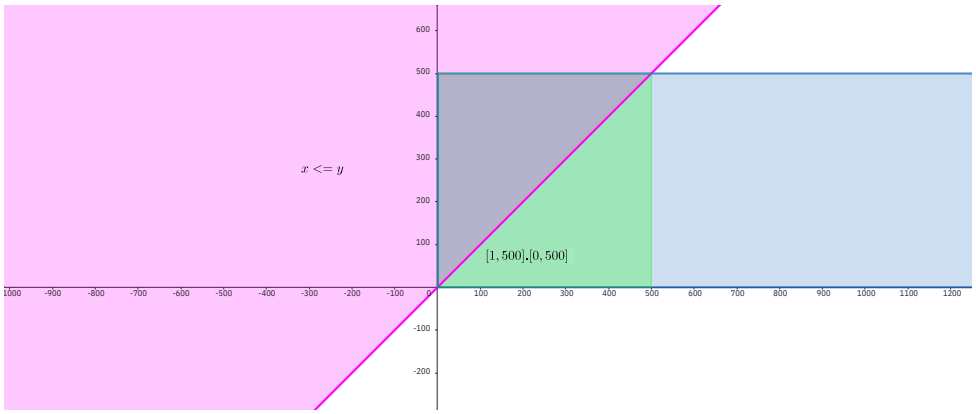


Figura 3.4. Programa com intervalos finais x = [1, 500] e y = [0, 500].

Uma noção visual da redução do espaço de busca da técnica é apresentado na **Figura 3.5**, onde no gráfico da figura (a) se tem uma região inicial na cor azul indicando o espaço de busca a ser explorado na verificação sem o auxílio deste método, observe que o esquema é moldado em torno da relação entre as variáveis envolvidas, ou seja, $x <= y$ onde a tonalidade rosa indica os valores onde essa restrição se mantém. No gráfico da figura (b), contém uma área menor de coloração esverdeada, que é uma estimação feita pelo algoritmo contrator para os domínios das variáveis, observe que temos valores que satisfazem e que não satisfazem a restrição, sendo esta a nova região do espaço de busca, mostrando assim, a influência desta técnica na verificação.



(a)

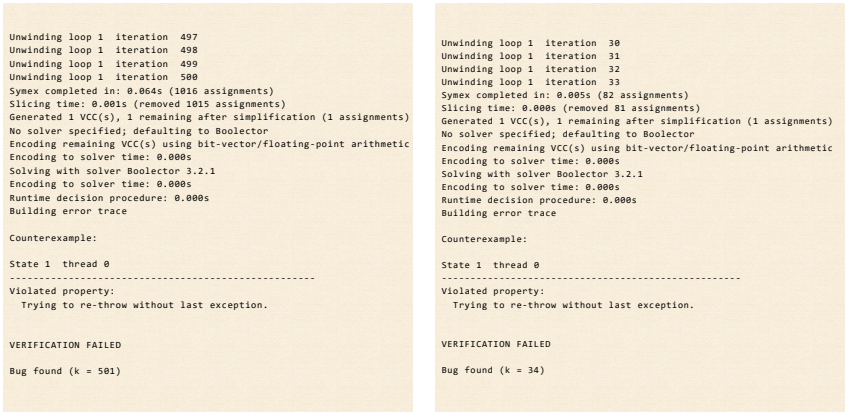


(b)

Figura 3.5. Na figura (a): Região inicial da exploração na cor azul. Na figura (b): Região de exploração com os intervalos contraídos na cor verde.

O objetivo do método é realizar uma contração dos domínios intervalares iniciais sem se preocupar com as atribuições que ocorreram ao longo do loop. Por isso, as estimativas dos intervalos sempre deixam uma margem de valores que podem tanto "satisfazer" quanto "não satisfazer" a propriedade que será verificada pelo método formal BMC.

Uma comparação dos resultados de verificação com e sem o auxílio do método, foi realizada usando uma ferramenta de verificação BMC (ESBMC-Jimple) para os programas Kotlin, o JBMC para os programas em Java e o ESBMC para os programas em C. Deste modo, para o programa exemplo, o verificador retornou o mesmo resultado em ambos os casos (com e sem intervalos contraídos). Diferenciando no tempo e profundidade de busca k, conforme a Figura 3.6.



(a) (b)

Figura 3.6. Na figura (a):Resultado de verificação sem a aplicação do método, k=501 e tempo=592.673s. Na figura (b): Resultado de verificação com a aplicação do método, k=34 e tempo=9.828s.

Os resultados de todos os experimentos realizados são discutidos na seção 5 onde as respostas foram as mesmas em ambos os casos em que foi encontrado o *bug*. A diferença ocorreu na quantidade de profundidade de busca k do modelo BMC que o verificador realizou, influenciando deste modo no tempo da verificação.

3.2. Trabalhos Relacionados

Nesta seção são relatados alguns trabalhos que apresentam afinidade com a redução dos domínios de variáveis, em particular, trabalhos que contém a aplicação de contratores como parte do processo e trabalhos que envolvam a análise e redução de domínios de variáveis em verificação de software.

3.2.1. Trabalhos com Redução do Domínio de Variáveis - (com contratores)

No trabalho "*A minimal contractor for the polar equation: Application to robot localization*" [B. Desrochers 2016], é abordado um teorema que pode ser usado para construir contratores mínimos consistentes com equações e outro teorema para derivar um separador ótimo de um contrator mínimo. Como aplicação, é enfatizado a restrição polar de canalização associada à mudança entre coordenadas cartesianas e coordenadas polares. O método é executado no problema de localização de um robô subaquático real, onde são coletadas medidas de alcance e goniométricas de pontos de referência.

No trabalho intitulado *"Robust TDOA passive location using interval analysis and*

contractor programming" [Olivier Reynet 2009] é apresentada uma nova metodologia para resolver problemas de localização passiva não linear. Ela é baseada em uma linguagem de modelagem de intervalo de alto nível chamada "*Quimper*". Considerando que as resoluções clássicas de localização passiva não fornecem nenhuma garantia de convergência para uma solução, análise de intervalo, propagação de restrição e contratores permitem evitar qualquer aproximação e qualquer linearização. Além disso, o Quimper fornece naturalmente garantias de localização e erro limitado.

No trabalho intitulado "*Improving a Constraint Programming Approach for Parameter Estimation*" é abordado o problema de encontrar todas as instâncias de um modelo paramétrico que possam explicar pelo menos q observações em uma tolerância. O trabalho usa um algoritmo combinatório [Neveu et al. 2015] que percorre exaustivamente todo o espaço de vetores de parâmetros para extrair as instâncias válidas do modelo. Esse algoritmo é baseado em métodos de programação de restrição de intervalo e no chamado operador de q-interseção, um operador de interseção relaxado que assume que pelo menos q dados observados são mentirosos. Esse artigo propõe várias melhorias no algoritmo. A maioria deles são genéricos e alguns outros são dedicados ao problema de detecção de forma usado para validar nossa abordagem. Comparado ao algoritmo de [Neveu et al. 2015], esse algoritmo pode garantir um número de observações ajustadas nas instâncias do modelo produzido. Além disso, os primeiros experimentos em reconhecimento de planos e círculos destacam acelerações de duas ordens de magnitude.

O **Quadro 3.1** resume a seleção representativa dos trabalhos relacionados com aplicação de contratores e redução de domínios de variáveis, ou seja, trabalhos onde se tem o uso de programação por contratores (*Contractor Programmming*) aplicados em uma determinada área de conhecimento e como é aplicada. Cada trabalho é apresentado com seu título, área de aplicação, autores, ano e a revista/conferência de publicação.

3.2.2. Trabalhos Com Análise de Intervalos e Redução do Espaço de Busca na Verificação Formal

No trabalho intitulado "*Using range analysis for software verification*" [Zaks et al. 2006], foi usada técnicas de abstração leves, eficientes e sólidas para determinar estaticamente possíveis intervalos para valores de variáveis de programa. Essas informações de intervalo para cada variável podem ser usadas para melhorar a eficiência do software de verificação

Trabalhos Relacionados: Aplicação de contratores e redução de domínios de variáveis				
Título do trabalho	**Área de Aplicação**	**Autores**	Ano	**Revista/Confêrencia**
A minimal contractor for the polar equation: Application to robot localization	Robótica: localização de um robô subaquático.	B. Desrochers e L. Jaulin,	2016	Engineering Applications of Artificial Intelligence
Robust TDOA passive location using interval analysis and contractor programming	Localização passiva não linear.	O. Reynet, L. Jaulin e G. Chabert.	2009	International Radar Conference ”Surveillance for a Safer World”(RADAR 2009)
Improving a Constraint Programming Approach for Parameter Estimation	Modelo paramétrico: O problema de estimativa de parâmetros.	B. Neveu, M. de la Gorce e G. Trombettoni	2015	27th International Conference on Tools with Artificial Intelligence (ICTAI

Tabela 3.1. Quadro de trabalhos relacionados: Aplicação de contratores.

de modelo, fornecendo um espaço de estado menor a ser considerado pelos mecanismos de verificação de *back-end*, como aqueles baseados em BDDs ou SAT-solvers. O principal método é baseado no *framework* que formula cada problema de análise como um sistema de restrições de desigualdade entre polinômios simbólicos vinculados. Em seguida, reduz o sistema de restrição a um programa linear, que pode ser analisado pelos poderosos mecanismos de computação disponíveis direcionados para programas lineares.

No trabalho *"LocFaults: A new flow-driven and constraint-based error localization approach*" [Bekkouche 2015] uma abordagem chamada LocFaults é apresentada como uma técnica de localização de erros baseada em fluxo e restrições. Essa abordagem inovadora utiliza o grafo de controle de fluxo do programa e um sistema de restrições para identificar conjuntos mínimos de correção para caminhos específicos, tornando a localização de erros mais precisa e eficiente. Experimentos preliminares com linguagem JAVA demonstraram a eficácia da técnica, especialmente na detecção e correção de erros em programas com declarações numéricas.

O Trabalho sobre redução de espaço de busca com análise estática e interpretação abstrata de Rafael Menezes [Menezes et al. 2024a] (Membro do grupo de estudo/projeto de qual faço parte), analisa o comportamento de programas em C de forma mais abstrata, permitindo a identificação de propriedades e comportamentos específicos. O mesmo faz

uso do GCSE, que é uma alternativa para otimizar a verificação formal de programas realizando a eliminação de sub-expressão comum, especialmente aqueles que envolvem operações complexas com ponteiros.

Finalmente o trabalho de Mohannad Aldughaim [Aldughaim et al. 2023] sobre contração de intervalos em verificação de programas em linguagem C, como já mencionado, este trabalho é parte de uma colaboração de um projeto de pesquisa (SWPERFI). A automação da contração em programas C está disponível no verificador usado no projeto, o ESBMC.

O **Quadro 3.2** fornece trabalhos onde ocorre a análise de intervalos e redução do espaço de busca em verificação formal, ou seja, trabalhos mais relacionados com a área de aplicação deste trabalho, apresentando o autor, ano, o método CSP/CP, a técnica CP usada, bem como os solvers usados, os programas a sua aplicação.

É importante destacar que essa revisão não é exaustiva, mas busca fornecer uma visão geral das contribuições significativas na área. Ao examinar esses trabalhos, foram identificados tendências, lacunas e oportunidades para contribuir com novos conhecimentos.

Análise de Intervalos e Redução do Espaço de Busca em Verificação Formal					
Trabalho	**CSP/CP**	**Técnica CP**	**BMC/Solvers**	**LPs/Programas**	**Aplicação**
Aleksandr Zak (2006)	Redução de espaço de busca (análise estática - interpretação abstrata)	Propagação (inteiros)	BDD/SAT	JAVA	-
Mohammed Bekkouche (2015)	Localização de erro em código (pós-verificação)	Alg. de Extração de MUSes com CSP	JBMC	JAVA	Localização de ERRO/ VIOLAÇÃO)
Rafael (2024*) OUR GENERAL GRIUP Membro Proj. SWPERFI	Redução de espaço de busca (análise estática - interpretação abstrata)	Eliminação de sub-expressão comum Contratação intervalos (reais / inteiros)	ESBMC ESBMC-JIMPLE	GERAL	GERAL
Mohannad (2024*) OUR GENERAL GROUP	Redução de espaço de espaço de busca - redução domínio de variáveis (pré-análise)	Contração de intervalos	ESBMC	C	-
ESTA DISSERTAÇÃO Membro Proj.SW PERFI	Redução de espaço espaço de busca - redução domínio de variáveis (pré-análise)	Contração de intervalos	ESBMC ESBMC-JIMPLE JBMC	C, JAVA e KOTLIN	Vulnerabilidades

Tabela 3.2. Quadro de características gerais de trabalhos relacionados.

Capítulo 4

Projeto de Experimentos com Linguagens de Programação

Neste capítulo será discutido o projeto de experimento usado nesta pesquisa. Destacando a seleção de técnicas e ferramentas, as linguagens de programação utilizadas e os benchmarks empregados para avaliar o desempenho e a eficácia do método proposto. Inicialmente é discutida a seleção de técnicas e ferramentas, abrangendo áreas como Programação por Contratores, o pacote IBEX-Lib e ferramentas como ESBMC e JBMC. Essas ferramentas desempenham papéis fundamentais na verificação formal de software e neste trabalho. Em seguida, é apresentada a seleção das linguagens de programação usadas, sendo linguagens C, Java e Kotlin. Além disso, a descrição dos benchmarks de cada linguagem adotados nos experimentos realizados. Por fim, uma discussão dos tipos de análises e as etapas dos procedimentos foram conduzidos nos experimentos, destacando a importância da metodologia para mostrar a efetividade da técnica desta pesquisa.

4.1. Seleção de técnica e ferramentas CSP/CP e BMC

Nesta seção, é descrita a abordagem de programação por restrições adotada no trabalho, conhecida como Contractor Programming, que é uma abordagem CP onde podemos usar contratores, aritmética intervalar e contratores. Além disso, é explanado sobre a biblioteca Ibex-Lib, a qual contém os algoritmos contratores. Por seguinte, um panorama dos verificadores formais ESBMC (ESBMC-Jimple) e JBMC que foram empregados nos experimentos.

4.1.1. Programação por Contratores

A Programação por Contratores (*Contractor Programming*) é um modelo de programação que busca alcançar soluções relacionadas ao processamento de restrições sobre o conjunto dos reais. Nesse modelo, o destaque é por dar mais liberdade no design do solver, introduzindo conceitos de programação onde antes apenas os parâmetros de configuração estavam disponíveis. A ideia geral é aplicar operações matemáticas em intervalos com auxílio de algoritmos chamados contratores. Assim, o contractor programming tem uma maneira de lidar harmoniosamente com um conjunto maior de problemas, ao mesmo tempo, em que fornece um controle fino sobre os mecanismos de resolução. O formalismo dos contratores e o sistema de pavimentação são as duas contribuições siginficativas deste modelo. [Chabert and Jaulin 2009]

4.1.2. Pacote IBEX-Lib

IBEX é uma biblioteca C++ para processamento de restrições sobre números reais. Ela fornece algoritmos confiáveis para lidar com restrições não lineares. Em particular, os erros de arredondamento também são levados em consideração. Baseia-se na aritmética intervalar e na aritmética afim. A principal característica do Ibex é sua capacidade de construir estratégias declarativamente através do paradigma de *contractor programming*. Também pode ser usado como um solucionador de caixa preta. Dois problemas emblemáticos que podem ser abordados são: **Resolução do sistema**. Um recinto garantido para cada solução de um sistema de equações (não lineares) é calculado. **Otimização global.** Um minimizador global de alguma função sob restrições não lineares é calculado com limites garantidos no mínimo objetivo. O Ibex permite criar intervalo, vetores de intervalo, matrizes de intervalo e matrizes de intervalo. Todos esses objetos representam conjuntos, o mesmo oferece a capacidade de construir algoritmos baseados em intervalos de alto nível de forma declarativa através do paradigma de programação por contratores [Chabert and Ninin 2016].

4.1.3. Ferramenta ESBMC e JBMC

ESBMC (*Efficient SMT-based Context-Bounded Model Checker*) é uma ferramenta de verificação de modelo de código-fonte aberta e permissiva. Sua função principal é verificar programas em linguagens como C/C++ em ambientes de thread único ou multithread. O que o torna diferenciado é que não requer que os usuários anotem seus programas

com pré ou pós-condições, o que simplifica significativamente o processo de verificação. Ao invés disso, os usuários podem declarar propriedades adicionais usando instruções assert, que são então verificadas pelo ESBMC. Além disso, o ESBMC oferece duas abordagens para modelar programas multithread durante a verificação: uma abordagem de gravação lenta e outra programada. Essas abordagens permitem uma análise detalhada do comportamento do programa em ambientes multithread. Uma característica importante do ESBMC é sua capacidade de converter as condições de verificação em diferentes teorias de base, adaptando-se assim a uma variedade de cenários de verificação. Essas condições são então encaminhadas diretamente para um solucionador SMT (*Satisfiability Modulo Theories*), responsável por determinar a satisfatibilidade das condições e, portanto, a correção do programa em questão. Essa abordagem facilita a detecção de falhas e vulnerabilidades nos programas verificados, contribuindo para a melhoria da segurança e confiabilidade do software. [Menezes et al. 2024a] Atualmente, no ESBMC, contém o primeiro verificador de modelo projetado para verificar programas escritos em Kotlin, utilizando a representação intermediária Jimple. Proporcionando uma abordagem robusta e eficiente para garantir a segurança de programas Kotlin. O ESBMC-Jimple se utiliza do framework Soot para obter o Jimple Intermediate Representation (IR), que representa uma versão simplificada do código-fonte Kotlin, limitada a no máximo três operandos por instrução. Isso facilita a análise do código e a verificação das propriedades de segurança do programa [Menezes et al. 2022].

O Java Bounded Model Checker (JBMC), é uma ferramenta de verificação de modelo especializada em verificar o bytecode Java. Desenvolvido com base na estrutura CPROVER, o JBMC é projetado para processar o bytecode Java juntamente com um modelo das bibliotecas Java padrão, permitindo a verificação de um conjunto de propriedades específicas do programa. Os resultados experimentais obtidos com o JBMC são promissores, demonstrando sua capacidade de verificar corretamente um conjunto diversificado de benchmarks Java encontrado na literatura. Além disso, os resultados mostram que o JBMC é competitivo em termos de desempenho e precisão quando comparado com dois verificadores Java de última geração. Essa ferramenta oferece uma abordagem valiosa para garantir a qualidade e a confiabilidade de programas Java, identificando potenciais falhas e vulnerabilidades por meio de uma verificação rigorosa do modelo. Com o JBMC, os desenvolvedores podem ter mais confiança na segurança e robustez de seus sistemas

Java, contribuindo assim para a construção de software de alta qualidade e livre de erros [Cordeiro et al. 2018].

4.2. Seleção das Linguagens de Programação e Benchmarks

Nessa seção, é apresentada a seleção das linguagens de programação e benchmarks utilizados nos experimentos, com foco no método de contração de intervalos com CSP/CP na verificação formal de software BMC. Onde se optou por incluir as linguagens C, Java e Kotlin, levando em consideração suas adoções no contexto da verificação formal BMC.

4.2.1. Linguagens C, Java e Kotlin

Como já relatado, o emprego deste método apresenta resultados efetivos para linguagens escritas em C, uma linguagem amplamente explorada na verificação BMC. Com base nesses resultados positivos, foi possível ampliar o método para as linguagens Kotlin e Java, utilizando verificadores formais específicos (ESBMC-jimple e JBMC) que são semelhantes ao verificador usado para C (ESBMC). Além disso, a inclusão da linguagem C neste trabalho tem como objetivo fornecer uma base de comparação com os resultados obtidos em Kotlin e Java, dado que já se sabe que a aplicação da técnica em C oferece uma garantia mais sólida de efetividade. A opção por não incluir outras linguagens de programação além de Kotlin e Java devido à ausência de verificadores formais nas ferramentas disponíveis nessa pesquisa. Assim, a escolha de Kotlin e Java, juntamente com C, foi feita para garantir a integridade e a relevância dos resultados obtidos.

4.2.2. Descrição dos Benchmarks

Os programas usados nos experimentos foram baseados em Benchmarks Java e C da competição de verificação de software (*Competition on Software Verification*, SV-COMP) [SVCOMP 2022], através dos benchmarks foi possível a geração de programas exploram estrutura de repetição, pois este tipo de estrutura pode promover um espaço de busca grande para a verificação BMC.

Os programas usados exploram loops, do tipo “while", "do while"e "for”, além de um caso do tipo condicional “if”. Em todos os benchmarks, usou-se uma propriedade a ser checada após o desdobramento do loop. Também foram fornecidos o mesmo programa para cada linguagem (Kotlin, C e Java), para comparar a eficiência e atuação do método. O **Quadro 4.1** apresenta os benchmarks utilizados na pesquisa, destacando sua estrutura de repetição e os programas associados, além das linguagens de programação utilizadas.

Descrição dos Benchmarks Usados		
Estrutura de Repetição	**Programas**	**Linguagens**
while	loop1, loop2, loop3, loop5, loop9, loop10	Kotlin/Java/C
for	loop4, loop7, loop11	Kotlin/Java/C
do while	loop6	Kotlin/Java/C
if	loop8	Kotlin/Java/C

Tabela 4.1. Quadro: Descrição dos programas usados: Mesmo programa em versão Kotlin, Java e C.

4.3. Tipos de Análises e Etapas dos Procedimentos

Ao usar o método nos programas Kotlin, Java e C, a construção de cada experimento ocorreu com uso de duas versões do mesmo programa. Uma contendo o programa inicial sem intervalos contraídos e outra com os novos valores de intervalo, sendo esta a abordagem para todas as linguagens. Posteriormente, foi empregado para cada linguagem Kotlin, Java e C, os verificadores formais ESBMC-jimple, JBMC e ESBMC, respectivamente. Usando ambas versões do mesmo programa (com intervalos contraídos e sem intervalos contraídos).

Vale salientar, que nos verificadores ESBMC-Jimple e ESBMC usados nos programas Kotlin e C, foi empregada a funcionalidade de $k - induction$, que realiza a verificação usando a k-indução, que começa com k = 1 e aumenta até um número máximo de iterações, analisando incrementalmente o programa. Do mesmo modo, foi utilizado o $unlimited - k - steps$ que não impõe um limite k para a verificação, podendo checar todos os estados do programa. No verificador JBMC, foi usado a funcionalidade de $incremental - loop$ que desdobra o loop especificado de forma incremental e em seguida checa a propriedade (No JBMC não se tem a k-indução).

Finalmente, em todos os procedimentos experimentais de cada linguagem, usou-se a ferramenta *Benchexec* [Wendler and Beyer 2023] para realizar a medição de tempo de forma confiável, levando em consideração apenas o tempo de CPU na execução de cada verificação. O **Quadro 4.2** contém as ferramentas e funcionalidades usadas nos experimentos, descrevendo o verificador, a linguagem e o medidor de tempo.

FERRAMENTAS USADAS E FUNCIONALIDADES			
Verificador	**Linguagem**	**Funcionalidade Usada no Verificador**	**Medidor de Tempo**
ESBMC	C	$k-induction, unlimited-k-steps$	Benchexec
ESBMC-JIMPLE	Kotlin	$k-induction, unlimited-k-steps$	Benchexec
JBMC	Java	$incremental-loop$	Benchexec

Tabela 4.2. Quadro: Ferramentas usadas para a execução dos experimentos e as funcionalidades usadas no verificador.

4.4. Aplicação e Procedimentos do Método em Vulnerabilidades de Software

Nesta seção, é explorada a aplicação do método de estimação de intervalos na verificação de vulnerabilidades de software comuns, utilizando a CWE *(Common Weakness Enumeration)* [CWE] como base para a seleção das vulnerabilidades relevantes. O objetivo principal é avaliar o desempenho do método na identificação em programas C que apresentem essas vulnerabilidades. O CWE fornece um conjunto de categorias e subcategorias para descrever as fraquezas de segurança encontradas em sistemas de software. Cada entrada na CWE é identificada por um número exclusivo e contém informações detalhadas sobre a fraqueza, incluindo descrição, sintomas, consequências e métodos de mitigação.

4.4.1. Descrição das CWE's utilizadas

Neste trabalho, foi usado algumas subcategorias de CWE que são particularmente relevantes para operações com intervalos aritméticos. Essa escolha se justifica pela exploração na redução de intervalos inicias das variáveis usando o método da pesquisa.

CWE-787: *Out-of-bounds Write* - Escrita fora dos limites: É uma categoria de vulnerabilidade de software onde um programa grava dados fora dos limites da área de memória alocada para um determinado objeto ou buffer. Isso pode levar a uma série de problemas de segurança, incluindo corrupção de memória, execução arbitrária de código, negação de serviço e vazamento de informações. Para realizar a verificação formal de software em relação à CWE-787, a técnica de contração de intervalos pode ser aplicada para detectar potenciais casos em que operações de gravação de dados podem ocorrer fora dos limites de um buffer. Isso envolve determinar intervalos de valores possíveis de variáveis, especialmente aquelas relacionadas ao tamanho do buffer e ao índice de acesso aos elementos do buffer.

CWE-20: *Improper Input Validation* - Validação de entrada inadequada: É uma categoria de vulnerabilidade de software onde um programa não valida adequadamente as entradas fornecidas pelos usuários ou por fontes externas antes de utilizá-las. Isso pode permitir que um usuário injete e execute código malicioso, comprometa a integridade ou a disponibilidade dos dados, ou exponha o sistema a outras ameaças de segurança. A técnica de contração de intervalos, pode ser aplicada para analisar a validação de entrada no código. Isso envolve examinar as condições de validação de entrada para determinar se elas cobrem adequadamente todos os casos possíveis e se há potenciais lacunas ou falhas na validação.

CWE-125: *Out-of-bounds Read* - Leitura fora dos limites: É uma categoria de vulnerabilidade de software onde um programa tenta acessar dados fora dos limites da área de memória alocada para um determinado objeto ou buffer. Essa vulnerabilidade geralmente ocorre quando um programa tenta acessar um índice de um array que está além dos limites válidos do array, resultando em comportamento indefinido e potenciais problemas de segurança. Na abordagem, os contratores são responsáveis por reduzir os intervalos dos domínios das variáveis durante a análise do programa. Isso pode ajudar a identificar cenários onde há acesso fora dos limites de um array ou buffer.

CWE-190: *Integer Overflow* - Estouro de número inteiro ou wraparound: É uma categoria de vulnerabilidade de software que ocorre quando uma operação de aritmética inteira resulta em um valor que excede o intervalo representável para o tipo de dado em questão. Especificamente, essa vulnerabilidade pode resultar em um estouro de inteiro quando um valor excede o valor máximo representável (*overflow*) ou em um *wraparound* quando ocorre uma transição entre os valores máximos e mínimos representáveis para o tipo de dado (por exemplo, em complemento de dois). A aplicação dos contratores, é realizada analisando as operações aritméticas presentes no programa e assim reduzir os intervalos dos domínios das variáveis envolvidas nessas operações. Isso pode ajudar a identificar cenários onde uma operação de aritmética inteira pode resultar em um estouro de inteiro.

CWE-119: *Improper Restriction of Operations within the Bounds of a Memory Buffer* - Restrição inadequada de operações dentro dos limites de um buffer de memória: Esta categoria de vulnerabilidade de software ocorre quando um programa realiza operações em um buffer de memória sem verificar adequadamente se essas operações ex-

cedem os limites válidos do buffer. Isso pode levar a corrupção de memória, execução de código arbitrário, negação de serviço e outros problemas de segurança. Um exemplo comum dessa vulnerabilidade é o uso de funções de manipulação de strings, como strcpy, strcat e sprintf, sem verificar se o tamanho dos dados de entrada excede o tamanho do buffer de destino. Isso pode resultar em um estouro de buffer, onde dados são escritos além dos limites do buffer, sobrescrevendo potencialmente dados importantes na memória. O método de redução de intervalos pode ser aplicado para detectar e mitigar vulnerabilidades de restrição inadequada de operações dentro dos limites de um buffer de memória (CWE-119) durante a verificação formal de software. Ao analisar operações de manipulação de buffers de memória, como cópia de strings, os contratores são usados para reduzir os intervalos dos domínios das variáveis envolvidos nessas operações. Isso pode ajudar a identificar cenários onde uma operação pode exceder os limites válidos do buffer.

CWE-835: *Loop with Unreachable Exit Condition* - Loop com condição de saída inacessível ('Loop Infinito'): Esta categoria refere-se a uma vulnerabilidade de software onde um loop é configurado com uma condição de saída que nunca será alcançada, resultando em um loop infinito. Esses loops podem consumir recursos do sistema de forma contínua e impedir que o programa avance, levando a uma condição de travamento ou negação de serviço. Ao analisar loops, o método de contração de intervalos pode ser usado para reduzir os intervalos dos domínios das variáveis envolvidas nas condições do loop. Isso pode ajudar a identificar cenários onde a condição de saída do loop é inatingível e pode levar a um loop infinito.

4.4.2. Programas com Vulnerabilidade Usados

Com base nas categorias de vulnerabilidades disponíveis no o site `www.cwe.mitre.org`, os programas usados nos experimentos foram modelados a partir dos exemplos disponíveis para cada categoria no site, uma vez que cada CWE contém sua descrição e exemplos com códigos. Desse modo, os programas foram moldados como um CSP/CP para realizar a estimação dos intervalos, tendo como restrições as vulnerabilidades de cada uma CWE. A escolha de aplicação apenas na linguagem C é explicada devido essa linguagem conter uma gama maior de vulnerabilidades envolvidas, incluindo vulnerabilidades de memória que pode ser representada com intervalos, o que se encaixa com o método usado na pesquisa.

4.4.3. Exemplo de Aplicação da Técnica nas Vulnerabilidades CWE

Para explorar as vulnerabilidades usando o método de pré-processamento de verificação BMC, cada vulnerabilidade foi empregada como uma restrição no problema de satisfação de restrições do programa a ser verificado. Cada programa possui suas próprias fraquezas, portanto, os resultados de verificação devem retornar "violação" e um contra-exemplo ao término da verificação, com e sem a utilização do método.

```
1 #include <assert.h>
2 #include <stdio.h>
3 #define TAM 1000
4
5 int main() {
6 int idSequence[TAM];
7
8       // Preencher o array de idSequence.
9   for (int i = 0; i <= TAM; i++) {
10        idSequence[i] = 10 + i;
11
12   }
13
14   return 0;
15 }
```

Figura 4.1. Exemplo de da aplicação do método de CSP/CP no pré-processamento BMC e Vulnerabilidades: CWE-787 em um programa C.

Desse modo, como exemplo de aplicação, considere a CWE-787, onde nesta fraqueza um programa escreve um valor (ou valores) fora dos limites de um array, acessando uma área fora da memória alocada. Como no programa da **Figura 4.1** que contém um array de tamanho definido pelo tamanho da variável TAM. Note que neste caso, temos o valor de TAM com 1000 e o array é preenchido até este valor pelo incremento da variável i. No entanto, o primeiro valor do array ocorre na posição i=0, ou seja, ao fazer i <= TAM o loop faz com que um valor seja escrito fora do limite do array, tornando o programa vulnerável.

Para alcançar o CSP/CP do programa, as variáveis envolvidas e seus intervalos são considerados e analisados. Bem como, a restrição que na aplicação em vulnerabilidades, usamos a propriedade violada em cada CWE. No caso do programa, a restrição consiste no tamanho do array que não deve ser menor que a quantidade de valores inscritos nele. Assim, o CSP/CP é escrito como:

CSP :

Variáveis e seus domínios (intervalos):

- $TAM = [1000, 1000] = 1000$ (Intervalo pontual representado um valor real);
- $idSequence = [1, TAM] = [1, 1000]$ (Intervalo representando a quantidade máxima do array);
- $i = [0, TAM] = [0, 1000]$ (Intervalo representando a quantidade de valores a serem alocados no array).

Restrição: Como restrição, é usado a condição do tamanho do array não ser menor que a quantidade de valores inscritos nele, nesse caso temos a quantidade de valores representado por i. Assim, é estabelecido uma relação entre $i = [1, TAM] = [0, 1000]$ (quantidade de valores) e $idSequence = [1, TAM] = [1, 10000]$ (tamanho do array):

$$\frac{[idSequence]}{[i]} \geq 1.$$

Esta relação garante que $idSequence$ seja sempre maior ou igual que a quantidade de elementos i que neles contém. No entanto, em termos de redução de intervalos decorre que o caso mínimo (=1), os intervalos são do mesmo tamanho do outro e não ocorre redução. Assim, para alcançar uma redução maior é utilizado $\frac{[idSequence]}{[i]} \geq 100$**.**

```c
#include <assert.h>
#include <stdio.h>
#define TAM 10 //O valor de TAM foi de 1000 para 10

int main() {
   int idSequence[TAM];

      // Preencher o array de id.
   for (int i = 0; i <= TAM; i++) {
       idSequence[i] = 10 + i;

    }

   // Restrição: Comprimento do array é igual a 10, usando assert.
    assert(sizeof(idSequence) / sizeof(idSequence[0]) <= TAM);

  return 0;
}
```

Figura 4.2. Exemplo de da aplicação do método de CSP/CP no pré-processamento BMC e Vulnerabilidades: Valores de intervalos atualizados, TAM = 10.

Finalmente, passando o CSP e os valores para realizar a contração junto aos contratores. Retor-se os valores os domínios:

$i = [0, 10]$

O intervalo da variável i é reduzido de $[0, 1000]$ para $[0, 10]$.

Temos ainda, $i = [0, TAM]$ e $idSequence[1, TAM]$. Assim, $TAM = [10, 10] = 10$.

Observe que, com essa abordagem não se perde a relação entre cada tamanho de intervalo envolvido no CSP. Mas houve uma redução significativa de intervalo que, em termos de verificação BMC, não deve retornar resultado diferente sem a adoção do método. A **Figura 4.2** contém o valor de TAM = 10, atualizado e junto um *assert* para verificar se o tamanho do array $idSequence$ é menor que o tamanho TAM, em tal caso, o valor 10.

4.4.4. Procedimentos de Verificação Com as Vulnerabilidades

O método de estimação de intervalos foi empregado no pré-processamento da verificação BMC e usando o verificador para programas C (ESBMC). O verificador realizou a verificação com e sem pré-processamento (CSP/CP com e sem BMC). Desse modo, foi analisado o desempenho de verificação em algumas vulnerabilidades relevantes, especialmente aquelas que envolvem operações com intervalos aritméticos, visando a compatibilidade com a técnica.

Assim, primeiramente o programa C com a vulnerabilidade CWE especificada foi construído com duas versões, com e sem os intervalos reduzidos. Obviamente, o modelo se adequou as especificação do verificador formal ESBMC, onde se tem as especificações assertivas como as restrições e que, neste são as vulnerabilidades. O método CSP/CP foi assim aplicado nos programas usados utilizando o verificador ESBMC com as configurações adequadas com e sem CSP/CP,

As configurações do ESBM foram as flags *k-induction* que utiliza caso base e indução para encontrar contra exemplos em k-passos nos programas a serem verificados e *unlimited-k-steps* que não limita os k-passos, visando a maior exploração possível do espaço de busca. Com relação à medição de tempo, foi utilizado a ferramenta benchexec [Wendler and Beyer 2023] que retorna valores com maior precisão e confiabilidade o tempo de processamento de cada experimento usado no verificador. Finalmente, foi possível fazer as relações com o tempo de execução, a profundidade de busca e o status da detecção de vulnerabilidades usada.

Capítulo 5

Resultados e Análises

Neste capítulo, são discutidos e analisados os resultados obtidos com a utilização do método de Programação por Restrições aplicadas na verificação formal BMC. Neste trabalho foram usados benchmarks escritos em linguagem Kotlin, Java e C, onde foi possível a geração de tabelas comparativas que nos fornece uma comprovação empírica de efetividade do método. Do mesmo modo, contém resultados da aplicação do método para classes vulnerabilidades comuns de software onde se obteve resultados interessantes.

Todos benchmarks usados para os experimentos (Experimentos em Kotlin, Java e C) estão disponíveis no diretório online `https://drive.google.com/drive/folders/19HP5CLSbYtsrTfMHbCUHg2N1nReFEyp0?usp=drive_link`.

5.1. Resultados Programas Kotlin

A aplicação do método em verificação BMC em programas Kotlin pode ser observada na **Tabela 5.1**. As respostas da verificação com e sem o auxílio do método mostram a efetividade da estratégia na maioria dos programas usados (loop1.kt, loop2.kt, loop4.kt, loop5.kt, loop6.kt, loop10.kt e loop11.kt), ou seja, 7 de 11 programas utilizados. Os experimentos retornaram o **mesmo resultado de verificação** em ambos os casos (com e sem os intervalos contraídos), mostrando que mesmo ao alterar a quantidade de valores dos domínios das variáveis envolvidas não se perde a mesma resposta na verificação.

Em termos de **tempo**, os experimentos mostram que os programas com intervalos contraídos com o método se sobressaíram com relação aos programas sem intervalos reduzidos, a atuação do método promoveu encontrar mais rapidamente um veredito de ve-

Programas Kotlin Verificados no ESBMC-JIMPLE						
Programa	**Tempo**		**Profundidade de Exploração**		**Resultado**	
	Sem CSP/CP	**Com CSP/CP**	**Sem CSP/CP**	**Com CSP/CP**	**Sem CSP/CP**	**Com CSP/CP**
loop1.kt	40.996 s	**15.669 s**	k = 129	**k = 33**	Propriedade Violada	Propriedade Violada
loop2.kt	592.673 s	**9.828 s**	k = 501	**k=34**	Propriedade Violada	Propriedade Violada
loop3.kt	*0.205 s*	*0.204 s*	*k = 1*	*k = 1*	*Propriedade Violada*	*Propriedade Violada*
loop4.kt	76.837 s	**2.819 s**	k = 100	**k = 18**	Propriedade Violada	Propriedade Violada
loop5.kt *	10.852 s	**2.569 s**	k = 81	**k = 6**	Bem Sucedida	Bem Sucedida
loop6.kt *	185.414 s	**0.180 s**	k = 151	**k = 2**	Bem Sucedida	Bem Sucedida
loop7.kt	*13.723 s*	*13.697 s*	*k = 101*	*k = 101*	*Propriedade Violada*	*Propriedade Violada*
loop8.kt	*0.242 s*	*0.239 s*	*k = 1*	*k = 1*	*Propriedade Violada*	*Propriedade Violada*
loop9.kt *	*0.261 s*	*0.263 s*	*k = 1*	*k = 1*	*Propriedade Violada*	*Propriedade Violada*
loop10.kt	582.225 s	**0.228 s**	k = 501	**k = 1**	Propriedade Violada	Propriedade Violada
loop11.kt	152.649 s	**0.173 s**	k = 301	**k = 1**	Bem Sucedida	Bem Sucedida

Tabela 5.1. Programas Kotlin verificados no ESBMC-JIMPLE sem e com uso do método.

rificação para cada programa envolvido quando aplicado o método de CSP/CP em BMC. Nos resultados também foi explorado a profundidade de pesquisa k necessária para determinar a violação (ou não) da propriedade do programa, esse quesito foi incluído no método usado para verificação.

A verificação BMC realizou uma exploração profunda para os k-estados de cada programa e os resultados com os intervalos reduzidos apresentaram um menor k (exploração de espaço de estados). Vale ressaltar que, mesmo nos casos onde não houve mudança com a aplicação do método (*loop3.kt, loop7.kt, loop8.k e loop9.kt* *), os tempos não foram afetados, bem como a profundidade de busca k e nem o resultado de verificação.

Ainda, o verificador ESBMC-JIMPLE encontra-se ainda em fase de aprimoramento, o que, como veremos, implicou em resultados de verificação diferentes nos casos loop5.kt *, loop6.kt * e *loop9.kt* * com relação aos resultados nas outras duas linguagens (Java e C) por ainda não ter suporte para todos os casos de propriedades. Entretanto, o intuito desta pesquisa é validação do método independente do verificador usado retornar falso positivo ou falso negativo, pois o interesse maior é na atuação do método onde independe dos processos do verificador BMC, o método de contração não altera o veredito de checagem.

5.2. Resultados Programas Java

Como já mencionado, foram usados os mesmos programas em todas as linguagens (mesmo programa em versões de linguagens diferentes). Desse modo, a aplicação da estratégia se apresentaram de forma semelhante em programas Java. Os resultados obtidos podem ser observados na **Tabela 5.2** onde é possível notar uma redução significativa de exploração de espaço de estados k (Loop1.java, Loop2.java, Loop4.java, Loop5.java, loop6.java, Loop9.java, Loop10.java), ou seja, 7 de 11 programas usados. Os programas *Loop3.java, Loop7.java, Loop8.java* e *Loop11.java* não tiveram mudança ao aplicar o método, mas não teve impacto negativo no tempo e nem na exploração de estados. Em ambos os casos, a verificação com e sem os domínios das variáveis contraídos, se obteve **os mesmos resultados de verificação**.

Com relação às **diferenças dos tempos** de verificação, não há diferenças significativas decorrente ao processamento do JBMC ocorrer de maneira rápida, onde desdobrou os loops de forma "instantânea" devido aos loops não realizarem grandes iterações e o verificador não conter a k-indução. Ocorrendo praticamente a mesma exploração de estados k que os programas em kotlin, diferenciando por uma iteração (para a maioria dos casos), o que se explica por o JBMC não conter a funcionalidade de k-indução que contém no ESBMC e ESBMC-JIMPLE.

5.3. Resultados Programas C

Para os programas C, como nos experimentos em Kotlin e Java, ocorreu uma comparação de eficiência de verificação por meio do tempo e profundidade de exploração, os resultados podem ser observados na **Tabela 5.3**. Como nos experimentos em Java, onde teve melhor desempenho foram os programas loop1.c, loop2.c, loop4.c, Loop5.c, loop6.c,

Programas Java Verificados no JBMC						
Programa	**Tempo**		**Profundidade de Exploração**		**Resultado**	
	Sem CSP/CP	**Com CSP/CP**	**Sem CSP/CP**	**Com CSP/CP**	**Sem CSP/CP**	**Com CSP/CP**
Loop1.java	0.0318 s	0.0275 s	k = 128	**k = 32**	Propriedade Violada	Propriedade Violada
Loop2.java	0.0669 s	0.0271 s	k = 500	**k=33**	Propriedade Violada	Propriedade Violada
Loop3.java	*736.424 s*	*269.104 s*	*k = 5000*	*k = 5000*	*Propriedade Violada*	*Propriedade Violada*
Loop4.java	0.0565 s	0.0409 s	k = 100	**k = 17**	Propriedade Violada	Propriedade Violada
Loop5.java *	0.0634 s	0.0508 s	k = 80	**k = 8**	Propriedade Violada	Propriedade Violada
Loop6.java *	0.0637 s	0.0398 s	k = 150	**k = 1**	Propriedade Violada	Propriedade Violada
Loop7.java	*0.0545 s*	*0.0543 s*	*k = 100*	*k = 100*	Propriedade Violada	Propriedade Violada
Loop8. java	*0.0411 s*	*0.0416 s*	*k = 1*	*k = 1*	*Bem Sucedida*	*Bem Sucedida*
Loop9.java *	0.0586 s	0.0584 s	k = 300	**k = 290**	Bem Sucedida	Bem Sucedida
Loop10.java	0.131 s	0.0464 s	k = 500	**k = 45**	Propriedade Violada	Propriedade Violada
Loop11.java	*0.0948 s*	*0.0952 s*	*k = 300*	*k = 300*	*Bem Sucedida*	*Bem Sucedida*

Tabela 5.2. Programas Java verificados no JBMC sem e com uso do método.

loop9.c, loop10.c, que correspondem aos mesmos programas que tiveram melhores resultados em Java. Do mesmo modo, os programas *loop3.c, loop7.c, loop8.c e loop11.c* não tiveram mudança nos resultados com a aplicação do método.

Os resultados de verificação (violação ou bem-sucedido) ocorreram de **maneira igual em ambos os casos**, com e sem intervalos reduzidos. Alcançando o mesmo veredito de verificação, comprovando que o método não influencia no resultado da verificação, melhorando apenas o tempo e o espaço de exploração. Com relação ao **tempo**, os programas C tiveram eficiência melhor com relação aos programas sem intervalos reduzidos, mostrando que, como já esperado, ocorre de forma eficaz em programas C, ressaltamos que o verificador ESBMC é um verificador robusto para programas C e contém a fun-

Programas C Verificados no ESBMC						
Programa	**Tempo**		**Profundidade de Exploração**		**Resultado**	
	Sem CSP/CP	**Com CSP/CP**	**Sem CSP/CP**	**Com CSP/CP**	**Sem CSP/CP**	**Com CSP/CP**
loop1.c	503.972 s	**37.722 s**	k = 129	**k = 33**	Propriedade Violada	Propriedade Violada
loop2.c	3077.805	**8.267**	k = 501	**k=34**	Propriedade Violada	Propriedade Violada
loop3.c	*0.0718 s*	*0.0783 s*	*k = 1*	*k = 1*	*Propriedade Violada*	*Propriedade Violada*
loop4.c	84.886 s	**2.298 s**	k = 101	**k = 18**	Propriedade Violada	Propriedade Violada
loop5.c *	61.962 s	**0.791 s**	k = 81	**k = 9**	Propriedade Violada	Propriedade Violada
loop6.c *	235.658 s	**0.0555 s**	k = 151	**k = 2**	Propriedade Violada	Propriedade Violada
loop7.c	*79.392 s*	*79.471 s*	*k = 101*	*k = 101*	*Propriedade Violada*	*Propriedade Violada*
loop8.c	*0.0729 s*	*0.0719 s*	*k = 1*	*k = 1*	*Bem Sucedida*	*Bem Sucedida*
loop9.c *	54.959 s	49.542 s	k = 302	**k = 292**	Bem Sucedida	Bem Sucedida
loop10.c	1581.954 s	**15.913**	k = 501	**k = 46**	Propriedade Violada	Propriedade Violada
loop11.c	*152.649 s*	*370.152*	*k = 301*	**k = 301**	*Bem Sucedida*	*Bem Sucedida*

Tabela 5.3. Programas C verificados no JBMC sem e com uso do método.

cionalidade de k-indução que realiza uma pesquisa profunda, implicando em um tempo de verificação muito mais efetivo em termos de comparação com o verificador para Java, JBMC. A **profundidade de exploração de estados k** na verificação também se apresenta de forma discrepante para maioria dos casos, devido ao longo tempo de verificação sem os intervalos reduzidos, comprovando a influência positiva de adoção deste método.

5.4. Discussões Sobre a Aplicação nas Linguagens Kotlin, Java e C

Os resultados da aplicação do método de CSP/CP com a contração de intervalos em programas escritos em Kotlin, Java e C revelam uma boa alternativa de BMC com a aplicação do método, onde 7 de 11 programas em todas as linguagens usadas nos experimentos ti-

veram uma redução no espaço de exploração, que pode ser notado com a diminuição do número k de estados explorados nos casos. Independentemente da linguagem de programação utilizada, com uma menor profundidade de exploração, como consequência os tempos de verificação tornam-se mais curtos, na **Tabela 5.4** contém as porcentagens de redução de tempo de verificação das linguagem usadas nos experimentos.

Ganho de Tempo de Cada Linguagem							
Linguagem	**Programas Efetivos**	**Quantidade de Programas**	**Tempo Total Sem o Método**	**Tempo Médio Sem o Método**	**Tempo Total Com o Método**	**Tempo Médio Com o Método**	**GANHO MÉDIO**
Kotlin	7	11	1656.077 s	150.552 s	45.869 s	4,169 s	**97.23 %**
Java	7	11	737.086 s	67.007 s	269.586 s	24.507 s	**63.42 %**
C	7	11	5833.381 s	530.307 s	564.411 s	51.310 s	**90.32 %**
Total	**21**	**33**	**8226.544 s**	**249.289 s**	**879.866 s**	**26,662 s**	**89.30 %**

Tabela 5.4. Porcentagens média de redução do tempo de verificação em programas eficazes.

Vale destacar que os experimentos usados são com programas que usam estruturas de repetição e foram pensados visando a usabilidade dos verificadores usados. Ou seja, programas simples, principalmente por conta do verificador Kotlin (ESBMC-JIMPLE) que ainda não tem suporte para qualquer caso de programa. Ainda, ressaltamos que, quando não ocorre uma contração dos intervalos, a quantidade k na verificação não diminui (loop7, loop8) e, consequentemente não há diminuição no tempo de verificação, nos casos loop3, loop11 tiveram contração de intervalo, mas não diminuíram a quantidade de exploração k, pois esses programas mesmo diminuindo os valores inicias dos domínios das variáveis, não implica na diminuição de geração dos estados gerados. Isso ocorre devido à semântica do programa não permitir alterar a quantidade de estados a serem explorados sem mudar sua semântica.

Sobre as linguagens de programação utilizadas, o potencial de ganho nos programas verificados em Kotlin e C. Que tiveram mais de 90% de ganho de tempo com a adoção do método. Importante relatar que os ganhos ocorreram utilizando o verificador ESBMC/ESBMC-Jimple que realizam a checagem da propriedade nos k-estados que o programa pode gerar usando a k-indução. Além disso, não se teve ganho em todos os programas (loop3, loop11, por exemplo), mas que não implica no aumento de tempos de verificação. Nos programas escritos em Java o ganho de tempo médio foi de $63,42\%$. Nesse tipo de linguagem, o gerenciamento de memória é realizado pela JVM (*Java Vir-*

tual Machine) e é levado em conta para o verificador JBMC. Finalmente, o ganho geral da adoção do método, que foi de $89,30\%$ indicando o grande potencial na adoção dessa estratégia na verificação de programas usando a verificação automática de modelos, pelo menos para tais linguagens utilizadas,

5.5. Resultados e Análise da Verificação em Vulnerabilidades de Software

A verificação em cada CWE ocorreu de maneira semelhante em cada caso, onde os intervalos e as condições das vulnerabilidades foram avaliados para aplicar o contrator e estimar os intervalos reduzidos. As categorias de vulnerabilidades estão dispostas na **Tabela 5.5** que contém todas CWEs usadas no trabalho.

Há dois programas de cada categoria de CWE, totalizando 12 experimentos realizados e como podemos observar, o método conseguiu bons resultados na maioria dos experimentos realizados e também não afetou o resultado de verificação, encontrando as vulnerabilidades em ambos os casos.

Verificação de Vulnerabilidades CWEs de Software								
CWE	Programas	Linguagem	Tempo BMC		Profundidade de Busca		Resultado BMC	
			Sem CSP/CP	Com CSP/CP	Sem CSP/CP	Com CSP/CP	Sem CSP/CP	Com CSP/CP
CWE-787	cwe_787_1.c	C	763.89 s	**0.170 s**	k= 2001	k = 11	Vulnerabilidade Detectada	Vulnerabilidade Detectada
	cwe_787_2.c	C	119.08 s	**0.171 s**	k=1001	k=11	Vulnerabilidade Detectada	Vulnerabilidade Detectada
CWE-20	cwe_20_1.c	C	**0.137 s**	0.519 s	k=1	k=1	Bem sucedida	Vulnerabilidade Detectada
	cwe_20_2.c	C	0.186 s	0,187 s	k=1	k=1	Bem sucedida	Bem sucedida
CWE-125	cwe_125_1.c	C	0.00730 s	0.00762 s	k=1	k=1	Vulnerabilidade Detectada	Vulnerabilidade Detectada
	cwe_125_2.c	C	1330.268 s	**6.765 s**	k = 501	k=51	Vulnerabilidade Detectada	Vulnerabilidade Detectada
CWE-190	cwe_190_1.c	C	+20 min	**127.71 s**	indefinido	k=1000	Vulnerabilidade Detectada *	Vulnerabilidade Detectada
	cwe_190_2.c	C	+20 min	**0.150 s**	indefinido	k = 2	Vulnerabilidade Detectada *	Vulnerabilidade Detectada
CWE - 119	cwe_119_1.c	C	1.062 s	1.039 s	k=11	k=11	Vulnerabilidade Detectada	Vulnerabilidade Detectada
	cwe_119_2.c	C	23.753 s	**0.174 s**	k = 101	k = 1	Vulnerabilidade Detectada	Vulnerabilidade Detectada
CWE-835	cwe_835_1.c	C	+20 min	**0,546 s**	indefinido	k=1	Vulnerabilidade Detectada *	Vulnerabilidade Detectada
	cwe_835_2.c	C	+20 min	**0,944 s**	indefinido	k=1	Vulnerabilidade Detectada *	Vulnerabilidade Detectada

Tabela 5.5. Tabela de resultados de experimentos realizados com a aplicação do método de CSP/CP no pré-processamento BMC em Vulnerabilidades de software.

Na categoria **CWE-787** (escrita fora dos limites de array) os dois programas ti-

veram uma boa redução de tempo e profundidade de busca, a metodologia usada nesta categoria promoveu uma contração no tamanho do array do programa, mas mesmo assim não modificou a vulnerabilidade envolvida, o que explica a detecção com menos profundidade de busca.

Na categoria **CWE-20** (validação de entrada inadequada), como podemos observar, não mostrou ganho de tempo e profundidade de busca usando a estratégia, os programas usados tem como característica validar valores que não deveriam ser validados no programa, ou seja, admite valores sem passar por uma especificação. Na verificação, isto não requer realizar uma profundidade de busca com grandes valores. Por exemplo, a validação de um determinado valor, $x < 0$, sendo que deveria apenas ter valores maiores que zero. Nesse caso, é inserida a especificação (*assert*) para verificação usando $x >= 0$ e ocorre da verificação encontrar tal vulnerabilidade por conta da especificação, por outro lado, quando não se tem a especificação o verificador não encontra essa vulnerabilidade, pois não interpreta como um bug no programa (cwe_20_1.c). No entanto, há casos onde mesmo inserindo uma determinada especificação o verificador não identifica uma entrada inválida devido ao programa não fornecer comandos que permitem identificar como entrada inválida. Como no programa cwe_20_2.c, onde se admite valores inteiros positivos e negativos, mas não especifica quando ocorre valores diferentes de inteiros, o que não se pode fazer usando a especificação com *assert*, pois teriam que envolver fazer comparações com inteiros e strings (por exemplo), e ocorre de não identificar as vulnerabilidades em ambos os casos.

Na categoria **CWE-125** (leitura fora dos limites), se teve situações diferentes para os dois programas usados. No primeiro, cwe_125_1.c, a vulnerabilidade foi encontrada em ambos os casos, mas não teve nenhum tipo de ganho, isso pode ser explicado pelo verificador encontrar o erro de escrita dos limites em sua primeira iteração de verificação. No segundo programa, cwe_125_2.c, houve um ganho significativo de tempo de profundidade de exploração, nesse caso, verificador teve que ler todo o array para encontrar o bug, e consequente com sua redução faz isso em menos procedimentos.

Na categoria **CWE-190** (Estouro de número inteiro), nos programas usados houve redução de tempo e profundidade de exploração no verificador, nos casos sem o método se teve tempos superiores a 20 minutos e com profundidade de busca indefinidas. No entanto, o verificador fornece a funcionalidade *–overflow-check* (indicado com * na tabela)

e quando acionada identifica esta vulnerabilidade, o que ocorreu de forma igual com e sem a aplicação da estratégia adotada.

Na categoria **CWE-119** (restrição inadequada de operações dentro dos limites de um buffer de memória), os resultados foram os mesmos nos dois programas usados, como podemos observar a vulnerabilidade foi identificada. No entanto, teve redução de tempo e profundidade de busca apenas no segundo programa usado (cwe-199_2.c), que é um array de string e o verificador teve que percorrer todo o array antes do veredito de verificação, o que foi reduzido ao aplicar o método.

Na última categoria usada **CWE-835** (Loop Infinito), a aplicação da estratégia influenciou de forma muito eficaz devido especificar esta vulnerabilidade, o que promove um escape para o loop infinito, que devido às configurações usadas (para explorar todos os possíveis estados) não ocorre sem a aplicação do método. Como observamos que em ambos os casos tempos superiores a 20 minutos e indefinição na profundidade de busca. No entanto, o verificador fornece a a função de *–overflow-check*, e ao acionar resulta em uma definição de verificação indicado com * na tabela.

Deste modo, o método demonstrou ser promissor na aplicação em verificação formal de programas envolvendo vulnerabilidades em linguagem C, especialmente em categorias que envolvem erros de memória (e loops infinitos). O método proporcionou uma busca mais rápida e foi capaz de identificar com sucesso as vulnerabilidades. Mesmo nos casos onde não houve redução, foi possível ter um mesmo resultado, ou seja, não se tem influência negativa com o método nos experimentos realizados. Ou seja, a estratégia apresenta uma eficácia variável, o desempenho do método varia conforme a categoria da vulnerabilidade, como na CWE-20, onde o ganho não foi significativo. Assim, ainda há necessidade de uma investigação mais profunda, ampliar a base de programas e ter um maior conjunto para uma análise abrangente e ainda, estender para mais casos, adicionando mais categorias de vulnerabilidades, além de outras linguagens de programação. Finalmente, a continuidade da pesquisa e o aprimoramento do método podem contribuir significativamente para a segurança de sistemas computacionais.

Capítulo 6

Considerações Finais

Nesta pesquisa de mestrado, foi apresentada uma proposta que pode ser adotada nos processos de verificação formal de programas através do uso da Programação por Restrições. Usamos algoritmos contratores no CSP do programa a ser verificado para reduzir os domínios das variáveis envolvidas dentro do código.

Durante a verificação BMC, a redução de k-estados no espaço de busca resultou em melhorias significativas no desempenho, especialmente no que diz respeito ao tempo de execução. Este avanço foi crucial para a eficiência do processo de verificação. Empiricamente, a efetividade do método foi demonstrada na verificação BMC para linguagens C, Java e Kotlin. A aplicação do método em diferentes verificadores formais forneceu uma certificação de eficácia mais consistente, mostrando que a técnica é robusta e adaptável a várias plataformas de verificação. Adicionalmente, a aplicação do método em vulnerabilidades apresentou resultados promissores para as CWES utilizadas, destacando seu potencial na identificação e mitigação de vulnerabilidades de segurança.

A relevância deste trabalho foram reconhecidos em importantes conferências, como o 10th Latin American Workshop on Cliques in Graphs (LAWCG 2022) [Deveza et al. 2022b], a XI Latin-Iberoamerican Conference on Operations Research (CLAIO 2022) [Deveza et al. 2022a] e o LV Simpósio Brasileiro de Pesquisa Operacional (SBPO 2023). [Deveza et al. 2023] Essas apresentações contribuíram para a divulgação e validação dos resultados obtidos, solidificando a importância da pesquisa na comunidade científica.

Referências

Common vulnerabilities and exposures. <https://www.cvedetails.com>. Acesso em: 09/02/2024.

Common weakness enumeration. <https://cwe.mitre.org>. Acesso em: 09/02/2024.

CyBOK kernel description. urlhttps://www.cybok.org/. Accessed: 2022-07-2022.

Aldughaim, M., Alshmrany, K. M., Gadelha, M. R., de Freitas, R., and Cordeiro, L. C. (2023). Fusebmc_ia: Interval analysis and methods for test case generation: (competition contribution). In *International Conference on Fundamental Approaches to Software Engineering*, pages 324–329. Springer.

Aldughaim, M., Alshmrany, K. M., Mustafa, M., Cordeiro, L. C., and Stancu, A. (2020). Bounded model checking of software using interval methods via contractors. *CoRR*, abs/2012.11245.

Alhawi, O. M., Rocha, H., Gadelha, M. R., Cordeiro, L. C., and de Lima Filho, E. B. (2021). Verification and refutation of C programs based on k-induction and invariant inference. *Int. J. Softw. Tools Technol. Transf.*, 23(2):115–135.

Allen, F. E. (1970). Control flow analysis. *SIGPLAN Not.*, 5(7):1–19.

B. Desrochers, b, L. J. (2016). A minimal contractor for the polar equation: Application to robot localization.

Bekkouche, M. (2015). *Combinaison des techniques de Bounded Model Checking et de Programmation Par Contraintes pour l'aide à la localisation d'erreurs*. PhD thesis, Université Nice Sophia Antipolis.

Biere, A., Heule, M., Maaren, H. V., and T.Walsh (2009). *Handbook of Satisfiability*. IOS press.

Branescu, I., Grigorescu, O., and Dascalu, M. (2024). Automated mapping of common vulnerabilities and exposures to mitre att&ck tactics. *Information*, 15(4):214.

Chabert, G. and Jaulin, L. (2009). Contractor programming. *Artificial Intelligence*, 173(Issue 11):1079–1100.

Chabert, G. and Ninin, J. (2016). Global optimization based on contractor programming: An overview of the ibex library. pages 555–559.

Clarke, E. M., Grumberg, O., Kroening, D., Peled, D., and Veith, H. (2018). *Model Checking,Second Edition*. The MIT Press.

Clarke, E. M., Klieber, W., Nováček, M., and Zuliani, P. (2012). *Model Checking and the State Explosion Problem*, pages 1–30. Springer Berlin Heidelberg, Berlin, Heidelberg.

Cordeiro, L., Fischer, B., and Marques-Silva, J. (2009). Smt-based bounded model checking for embedded ansi-c software. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 137–148.

Cordeiro, L., Kesseli, P., Kroening, D., Schrammel, P., and Trtik, M. (2018). JBMC: A bounded model checking tool for verifying Java bytecode. In *Computer Aided Verification (CAV)*, volume 10981 of *LNCS*, pages 183–190. Springer.

Cordeiro, L. C. (2010). Smt-based bounded model checking for multi-threaded software in embedded systems. In Kramer, J., Bishop, J., Devanbu, P. T., and Uchitel, S., editors, *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, pages 373–376. ACM.

Cordeiro, L. C., de Lima Filho, E. B., and de Bessa, I. V. (2020). Survey on automated symbolic verification and its application for synthesising cyber-physical systems. *IET Cyper-Phys. Syst.: Theory & Appl.*, 5(1):1–24.

Cordeiro, L. C., Fischer, B., and Marques-Silva, J. (2012). Smt-based bounded model checking for embedded ANSI-C software. *IEEE Trans. Software Eng.*, 38(4):957–974.

CyBOK (2019). *CyBOK: The Cyber Security Body of Knowledge*, volume verison 1. 31st October, 2019.

Deveza, J., Freitas, R., and Cordeiro, L. (2022a). Abstract: Application of interval arithmetic and constraint programming in the optimization of formal verification processes. In *XXI Latin-Iberoamerican Conference on Operations Research - CLAIO 2022*, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, Buenos Aires, Argentina.

Deveza, J., Freitas, R., Cordeiro, L., and Menezes, R. (2023). Pôster: Aplicando técnicas de programação por restrições e aritmética intervalar em processos de verificação formal de software. In *LV Simpósio Brasileiro de Pesquisa Operacional - SBPO 2023*,

São José dos Campos, SP, Brasil.

Deveza, J., Freitas, R., Santos, L., and Cordeiro, L. (2022b). Resumo: Control flow graph, formal verification and constraint programming techniques. In *10th Latin American Workshop on Cliques in Graphs - LAWCG 2022*, Curitiba, PR, Brazil.

Emerson, E. A. and Clarke, E. M. (1982). Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer programming*, 2(3):241–266.

Gadelha, M. R., Monteiro, F. R., Morse, J., Cordeiro, L. C., Fischer, B., and Nicole, D. A. (2018). ESBMC 5.0: An industrial-strength C model checker. In *ACM/IEEE Int. Conf. on Automated Software Engineering (ASE'18)*, pages 888–891, New York, NY, USA. ACM.

Jaulin, L., Kieffer, M., Didrit, O., and Walter, E. (2001a). *Subpavings*, pages 45–63. Springer London, London.

Jaulin, L., Kieffer, M., Didrit, O., and Walter, (2001b). *Applied Interval Analysis. Springer, London*.

Menezes, R., Aldughaim, M., Farias, B., Li, X., Manino, E., Shmarov, F., Song, K., Brauße, F., Gadelha, M. R., Tihanyi, N., Korovin, K., and Cordeiro, L. C. (2024a). ESBMC 7.4: Harnessing the Power of Intervals. In 30^{th} *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'24)*. Springer.

Menezes, R., Cordeiro, L., Freitas, R., Moura, D., and Cavalcante, H. (2022). Esbmc-jimple: Verifying kotlin programs via jimple intermediate representation. In *ISSTA*, Mon 18 - Fri 22 July 2022 Online.

Menezes, R. S., Aldughaim, M., Farias, B., Li, X., Manino, E., Shmarov, F., Song, K., Brauße, F., Gadelha, M. R., Tihanyi, N., Korovin, K., and Cordeiro, L. C. (2024b). ESBMC v7.4: Harnessing the power of intervals - (competition contribution). In Finkbeiner, B. and Kovács, L., editors, *Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part III*, volume 14572 of *Lecture Notes in Computer Science*, pages 376–380. Springer.

Monteiro, F. R., da S. Alves, E. H., da Silva, I., Ismail, H., Cordeiro, L. C., and de Lima Filho, E. B. (2018). ESBMC-GPU A context-bounded model checking tool to verify CUDA programs. *Sci. Comput. Program.*, 152:63–69.

Monteiro, F. R., Gadelha, M. R., and Cordeiro, L. C. (2022). Model checking C++ programs. *Softw. Test. Verification Reliab.*, 32(1).

Moore, R.E.and Kearfott, R. and Cloud, M. (2009). *Introduction to Interval Analysis*. SIAM, Fhiladelphia.

Morse, J., Cordeiro, L. C., Nicole, D. A., and Fischer, B. (2011). Context-bounded model checking of LTL properties for ANSI-C software. In Barthe, G., Pardo, A., and Schneider, G., editors, *Software Engineering and Formal Methods - 9th International Conference, SEFM 2011, Montevideo, Uruguay, November 14-18, 2011. Proceedings*, volume 7041 of *Lecture Notes in Computer Science*, pages 302–317. Springer.

Mustafa, M., Stancu, A., Delanoue, N., and Codres, E. (2018). Guaranteed slam—an interval approach. *Robotics and Autonomous Systems*, 100(February 2018):160–170.

Neveu, B., de la Gorce, M., and Trombettoni, G. (2015). Improving a constraint programming approach for parameter estimation. In *2015 IEEE 27th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 852–859.

O Globo (2024). Boeing 737 max: o que é o mcas, o software no centro da tragédia fatal envolvendo dois aviões e 346 mortos. `https://oglobo.globo.com/economia/negocios/noticia/2024/07/08/boeing-737-max-o-que-e-o-mcas-o-software-no-centro-da-tragedia-fatal-ghtml`. Acessado: 14-ago-2024.

Olivier Reynet, Luc Jaulin, G. C. (2009). Robust tdoa passive location using interval analysis and contractor programming.

Rastello, F. and Tichadou, F. B. (2022). *SSA-based Compiler Design*. Springer Nature.

Rossi, F., van Beek, P., and Walsh, T. (2006). *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science; 1ª edição (27 outubro 2006).

Sainz, M. A., Armengol, J., Calm, R., Herrero, P., Jorba, L., and Vehi, J. (2014). *Modal Interval Analysis: New Tools for Numerical Information*. Springer International Publishing.

SVCOMP (2022). Sv benchmarks.

Tihanyi, N., Bisztray, T., Ferrag, M. A., Jain, R., and Cordeiro, L. C. (2024). Do neutral prompts produce insecure code? formai-v2 dataset: Labelling vulnerabilities in code generated by large language models. *CoRR*, abs/2404.18353.

Tihanyi, N., Bisztray, T., Jain, R., Ferrag, M. A., Cordeiro, L. C., and Mavroeidis, V. (2023). The formai dataset: Generative AI in software security through the lens of formal verification. In McIntosh, S., Choi, E., and Herbold, S., editors, *Proceedings of the 19th International Conference on Predictive Models and Data Analytics in Software Engineering, PROMISE 2023, San Francisco, CA, USA, 8 December 2023*, pages 33–43. ACM.

Wendler, P. and Beyer, D. (2023). sosy-lab/benchexec: Release 3.17.

Zaks, A., Shlyakhter, I., Ivancic, F., Cadambi, H., Yang, Z., Ganai, M., Ashar, P., and Gupta, A. (2006). Using range analysis for software verification.

Apêndice A

Sistema de Transição de Estados: Estrutura Kripke

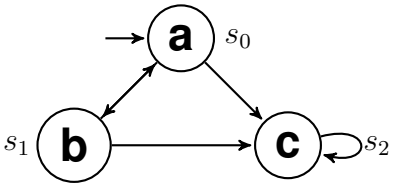
Na verificação de modelo, usa-se um sistema de transição de estados conhecido por estrutura Kripke, que é uma aboragem matemática usada em lógica modal e verificação de modelos para representar sistemas de estados com transições entre eles [Clarke et al. 2012].

Definição 6. *Uma estrutura Kripke $M = (S, S_0, T, L)$ é um sitema de transição de estados onde:*

- *S é o conjunto de estados do sitema;*
- *S_0 é o conjunto de estados iniciais, onde $S_0 \subseteq S$;*
- *T é uma relação de transição entre os estados, ou seja, $T \subseteq S \times S$;*
- *L é um função de rotulagem, isto é, $L : S \to \wp(\nu)$, onde ν é um conjunto de proposições atômicas.*

Um caminho finito de algum estado $s \in S$ é uma sequência $s_0, s_1, ..., s_n$ tal que $s_0 = s$ e $R(s_i, s_i + 1)$ para todo $0 \leq i < n$.

Como exemplo de estrutura Kripke, considere o grafo abaixo onde temos três vértices que represetam estados e as arestas as transições de cada estado.



Temos uma Estrutura Kripke $M(S, S_0, T, L)$, onde:

- Conjunto de estados: $S = \{s_0, s_1 e s_3\}$.
- conjunto inicial de estados: $S_0 = \{s_0\}$
- Relação de transição : $T = \{(s_0, s_1), (s_0, s_2), (s_1, s_0), (s_1, S2), (s_2, s_2)\}$.
- Rotulagem: $L(s_0) = a, L(s_1) = b, L(s_2) = c$.

Apêndice B

Lógicas Temporais

Lógicas temporais são formalismos que possibilitam descrever e analisar os comportamentos de estruturas de Kripke a partir do comportamento das proposições atômicas ao longo das trajetórias, nesta abordagem o tempo físico não é considerado no problema de descriçã.

Analisar e descrever a dinânimca de uma estrutura Kripke ocorre por meio das lógicas temporais. Neste tipo de lógica, uma fórmula pode especificar que uma propriedade é válida no próximo instante de tempo, após uma etapa de computação, pode especificar que eventualmente algum estado designado seja alcançado ou que um estado de erro nunca seja inserido [Clarke et al. 2012]. Ou seja, incluem em sua lógica, operadores lógicos temporais para expressar suas instâncias, fazendo-se de duas aboragem conhecidas como Lógica Temporal Linear (*Linear Temporal Logic* - LTL) e Lógica de Árvore de Computação (*Computation Tree Logic* - CLT) [Emerson and Clarke 1982].

Na **lógica temporal linear LTL**, o formalismo é usado para raciocinar sobre o comportamento de sistemas ao longo do tempo. As propriedades dos sistemas são expressas em termos de operadores temporais que descrevem as relações temporais entre estados ou sequências de estados.

Agorna, na **lógica temporal CTL** a formalidade é empregada para especificar propriedades de sistemas de estados finitos ou infinitos. Permitindo a expressão de propriedades sobre o comportamento do sistema em diferentes pontos no tempo, possibilitando a verificação formal de propriedades como correção, vivacidade e segurança. Baseada em árvores de computação, cada nó representa um estado do sistema e as transições entre

estados são representadas pelas arestas da árvore. Os operadores modais que são usados para quantificar sobre caminhos na árvore de computação. Alguns dos operadores da lógicas CTL e CTL são:

- **G** (Globalmente): Uma propriedade é verdadeira para todos os estados futuros.
- **F** (Eventualmente): Uma propriedade é verdadeira em algum estado futuro.
- **X** (Próximo): Uma propriedade é verdadeira no próximo estado.
- **A** (Sempre): Uma propriedade é verdadeira em todos os caminhos possíveis.
- **E** (Existe um caminho): Uma propriedade é verdadeira em um caminho infinito a partir deste estado.
- **U** (Até): Uma propriedade é verdadeira até que outra propriedade se torne verdadeira.

Para proposições atomicas p e q, temos como exemplo:

- $\mathbf{X}p$ "p é verdadeiro no próximo estado";
- $\mathbf{F}p$ "p é verdadeiro em algum estado no futuro";
- $\mathbf{G}q$ "q é verdaderiro para todos os estados no futuro";
- $q\mathbf{U}p$ "q é verdadeiro até que p seja verdadeiro".

Neste contexto, considere M uma estruta Kripke de um sistema, s um estado e ϕ uma fórmula que representao o estado do sistema. Um algoritmo de verificação de modelo é um procedimento de decisão para M, $s \models \phi$ (s satisfaz ϕ) [Clarke et al. 2012].