Universidade Federal do Amazonas

Faculdade de Tecnologia

Programa de Pós-Graduação em Engenharia Elétrica

Localização de Falhas em Programas Concorrentes em C

Erickson Higor da Silva Alves

Manaus – Amazonas

Setembro de 2018

Erickson Higor da Silva Alves

Localização de Falhas em Programas Concorrentes em C

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia Elétrica, como requisito parcial para obtenção do Título de Mestre em Engenharia Elétrica. Área de concentração: Automação e Controle.

Orientador: Prof. D.Sc. Eddie Batista de Lima Filho

Erickson Higor da Silva Alves

Localização de Falhas em Programas Concorrentes em C

Banca Examinadora

Prof. D.Sc. Eddie Batista de Lima Filho – Presidente e Orientador
TP Vision

Prof. Ph.D. Lucas Carvalho Cordeiro – Co-orientador
Escola de Ciência da Computação – UoM

Prof. D.Sc. Raimundo da Silva Barreto
Instituto de Computação – UFAM

Prof. D.Sc. Arilo Claudio Dias Neto
Instituto de Computação – UFAM

Manaus – Amazonas
Setembro de 2018

À minha mãe e minha esposa.

Agradecimentos

Em primeiro lugar, agradeço a Deus pelo dom da vida e por todas as conquistas alcançadas. E o mais importante, agradeço pelo Seu cuidado e Seu amor.

Agradeço à minha esposa, Jéssica Alves, por todo amor, suporte, cuidado e carinho. Eu amo você e sei que você sempre está ao meu lado. Você é o motivo de eu querer ir além.

Agradeço à minha mãe, Francisca Silva, ao meu pai, Rubens Alves, e ao meu irmão, Erick Alves, que me educaram e sempre me motivaram a ir além do que eu imaginava poder ir.

Agradeço aos meus amigos de escola, Henrique Cavalcante, Juscelino Tanaka, Leandro Paes, Matheus Santos e Tayane Figueira, que apesar de tudo, sei que sempre torcem por mim; aos meus amigos da UFAM, Arllem Farias, Breno Linhares, Ciro Cativo, Helton Nogueira, Robson Cruz, Rosmael Miranda, Thiago Cavalcante e Weider Serruia, que foram meus companheiros durante os anos de graduação, mestrado e da vida; aos meus amigos do trabalho, do INDT e do SIDIA, em especial Allann Silva e Pablo Quiroga, por todo o apoio durante cada passo; e aos meus amigos da Igreja, em especial Labib Ismail, pelas risadas, conversas e lanches.

Agradeço também aos meus orientadores, Prof. Eddie Lima Filho e Prof. Lucas Cordeiro, pela amizade, direcionamento, suporte, paciência e conselhos. Saibam que me espelho em vocês como profissional. Continuem procurando dar um caminho melhor a alunos através da educação.

"Porque dEle, e por meio dEle, e para Ele são todas as coisas. A Ele, pois, a glória eternamente. Amém!".

Romanos 11:36

Resumo

Este trabalho descreve uma nova abordagem para localizar falhas em programas concorrentes, a qual é baseada em técnicas de verificação de modelos limitada e sequencialização. A principal novidade dessa abordagem é a ideia de reproduzir um comportamento defeituoso em uma versão sequencial do programa concorrente. De forma a apontar linhas defeituosas, analizam-se os contraexemplos gerados por um verificador de modelos para o programa sequencial instrumentado e procura-se um valor para uma variável de diagnóstico, o qual corresponde a linhas reais no programa original. Essa abordagem é útil para aperfeiçoar o processo de depuração para programas concorrentes, já que ela diz qual linha deve ser corrigida e quais valores levam a uma execução bem-sucedida. Essa abordagem foi implementada como uma transformação código-a-código de um programa concorrente para um não-determinístico sequencial, o qual é então usado como entrada para ferramentas de verificação existentes. Resultados experimentais mostram que a abordagem descrita é eficaz e é capaz de localizar falhas na maioria dos casos de teste utilizados, extraídos da suíte da *International Competition on Software Verification* 2015.

Palavras-chave: Verificação de Modelos, Localização de Falhas, Sequencialização, Depuração.

Abstract

We describe a new approach to localize faults in concurrent programs, which is based on bounded model checking and sequentialization techniques. The main novelty is the idea of reproducing a faulty behavior in a sequential version of the concurrent program. In order to pinpoint faulty lines, we analyze counterexamples generated by a model checker to the new instrumented sequential program and search for a diagnosis value, which corresponds to actual lines in the original program. This approach is useful to improve the debugging process for concurrent programs, since it tells which line should be corrected and what values lead to a successful execution. We implemented this approach as a code-to-code transformation from concurrent into non-deterministic sequential program, which is then used as an input to existing verification tools. Experimental results show that our approach is effective and it is capable of locating faults in most benchmarks we have used, extracted from the International Competition on Software Verification 2015 suite.

Keywords: Model Checking, Fault Localization, Sequentialization, Debugging.

Conteúdo

Lista de Figuras

Lista de Tabelas

Abreviações

AA - **Á**rvore de **A**lcançabilidade

BMC - ***B**ounded **M**odel **C**hecking*

CV - **C**ondição de **V**erificação

ESBMC - ***E**fficient **S**MT-based Context-**B**ounded **M**odel **C**hecker*

GFC - **G**ráfico de **F**luxo de **C**ontrole

MAX-SAT - ***Max**imal **Sat**isfiability Problem*

MU - ***M**emory **U**nwinding*

SAT - *Boolean **Sat**isfiability Problem*

SMT - ***S**atisfiability **M**odulo **T**heories*

SV-Comp - *International **Comp**etition on **S**oftware **V**erification*

Capítulo 1

Introdução

Recentemente, tem sido mais comum o uso da tecnologia para lidar com diversas tarefas do dia-a-dia, cada uma com uma complexidade associada. Assegurar que sistemas funcionem apropriadamente implica diretamente em redução de custos e, em alguns casos, até em segurança de vidas [1]. Quando se trata de software, descobrir erros em software é uma atividade que precisa ser executada nos primeiros estágios do processo de implementação, fazendo da depuração de programas uma tarefa merecedora de bastante atenção. A depuração de programas é uma tarefa muito importante, mas também consumidora de bastante tempo, pois necessita de uma análise minuciosa para que falhas sejam encontradas. Ela pode ser dividida em três passos: detecção de falhas, localização de falhas e correção de falhas. No entanto, é possível reduzir o tempo associado a essa tarefa drasticamente se métodos automáticos forem aplicados nesses passos. Vários métodos já foram propostos com o objetivo de encontrar erros em software, como o teste de software [2] e técnicas baseadas em modelos [3, 4, 5, 6, 7, 8, 9]. Mais precisamente, quando um erro é encontrado, a causa do mesmo deve ser rastreada dentro do código-fonte [10, 11].

Programas concorrentes tem sido amplamente usados na área de sistemas embarcados devido ao menor tempo de resposta associado e o uso otimizado dos recursos computacionais disponíveis. No entanto, quando se trata de assegurar a corretude de tais programas, torna-se uma tarefa complexa, visto que o número de possíveis intercalações pode crescer exponencialmente com o número de *threads* e linhas de código. Desta forma, o processo de depuração nesta classe de programas torna-se uma tarefa exaustiva para desenvolvedores, ao ponto de até não serem descobertos defeitos no código [12].

Neste trabalho, instruções de código mal-formuladas são definidas como falha e exceções geradas em tempo de execução como erro.

Em relação à detecção de falhas em programas concorrentes, a verificação de modelos vem se mostrando uma técnica eficaz. Devido à sua abordagem, a partir de um modelo de um sistema, *i.e.*, o código-fonte, ela gera um sistema de transição de estados e procura por um caminho que leve a uma violação de uma propriedade especificada. Com o intuito de explorar o sistema de transição de estados do modelo sem estourar os recursos computacionais disponíveis, o uso da verificação de modelos limitada (do inglês *bounded model checking*), tem sido usada para procurar por violações em códigos concorrentes, procurando por contraexemplos em uma profundidade k, produzindo uma fórmula mais simples de ser solucionada.

Qadeer *et. al.* [13] propuseram o uso de sequencialização, *i.e.*, transformações de código formulada com regras específicas, para simplificar a verificação de modelos de programas concorrentes, evitando o problema de crescimento exponencial da complexidade associada a esta tarefa. Desta forma, ao criar um programa sequencializado, que representa o programa original com relação a uma determinada intercalação, diminui o peso sobre o verificador de modelos na hora de buscar violações de propriedades a uma profundidade k. Apesar de não relatar falso-positivos, a técnica pode não encontrar alguns erros. Todavia, uma importante observação desta técnica foi que erros de concorrência geralmente se manifestam em poucas trocas de contexto [13].

Em relação à etapa de localização de linhas defeituosas, *i.e.*, que levam à uma execução mal-sucedida do código, a verificação de modelos por si só não é capaz de apontá-las diretamente, pois o contraexemplo mostra a sequência de estados do programa que levam à violação, sendo necessário uma análise precisa para tentar isolar essas linhas, nem sempre sendo possível. Porém, Clarke *et. al.* [14, 15] e Rocha *et. al.* [11] citam que os contraexemplos contém de fato informações relevantes à localização de falhas.

Griesmayer *et. al.* [16] discutem o uso de técnicas de verificação de modelos para produzir contraexemplos que informem não somente a existência de falhas mas também as suas respectivas localizações em códigos sequenciais, a ideia é instrumentar o código original e fazendo uso do não-determinismo para atribuir valores às variáveis do programa. Esta abordagem, no entanto, funciona apenas para programas sequenciais.

Assim, quando se trata de localização de falhas em programas concorrentes, *i.e.*, apontar as linhas envolvidas em execuções mal-sucedidas de um programa, os itens descritos anterior-

mente isolados não são capazes de abordar este problema de forma sistemática. Ao elaborar uma metodologia para abordar este problema com tais técnicas, reduz-se o tempo de depuração de programas concorrentes, auxiliando desenvolvedores de software na fase de implementação a encontrar defeitos complexos de forma menos exaustiva.

1.1 Descrição do Problema

Este trabalho visa resolver o problema da localização de falhas em programas concorrentes utilizando técnicas de verificação de modelos limitada de forma automática. Para tanto, utilizaram-se as seguintes abordagens: (1) aplicação de uma gramática e regras de transformação para possibilitar a sequencialização de programas concorrentes e (2) uso de contraexemplos obtidos por um verificador de modelos para localizar falhas em tais programas.

A primeira abordagem deriva do fato de programas concorrentes serem mais difíceis de serem verificados, devido ao problema de explosão de estados (o número de estados possíveis cresce exponencialmente de acordo com o número de *threads* e trocas de contexto possíveis). A segunda consiste em usar a verificação de modelos limitada para obter um contraexemplo para um programa sequencial não-determinístico instrumentado, com o objetivo de extrair do mesmo as linhas que levam a uma execução defeituosa do programa original.

Assim, o método proposto utiliza técnicas de sequencialização de programas concorrentes, técnicas de verificação de modelos limitada e contraexemplos obtidos por um verificador para obter linhas defeituosas de um programa concorrente.

1.2 Objetivos

O principal objetivo deste trabalho é propor um método para localizar falhas, *i.e.*, assertivas mal-formuladas, em programas concorrentes utilizando técnicas de verificação de modelos limitada.

Os objetivos específicos são:

- Propor uma gramática de transformação de instruções de programa concorrente para uma versão sequencial, de modo que essa nova instrução tenha a mesma funcionalidade que a original.

- Propor uma regra de transformação de modelo concorrente de programas para um modelo sequencial, que simule o comportamento original do programa, *i.e.*, a mesma sequência de execução, gerando um programa sequencial não-determinístico.

- Aplicar um método de localização de falhas utilizando verificação de modelos limitada para obter, a partir do programa sequencial não-determinístico, as linhas do programa original que levam à falha e também extrair os valores a serem atribuidos às variáveis do programa para produzir uma execução bem-sucedida do mesmo.

- Avaliar experimentalmente o método proposto utilizando programas de uma suíte de teste extraída da *International Competition on Software Verification* 2015.

1.3 Contribuições

Este trabalho tem três principais contribuições. Primeiro, um estudo para localização de falhas em programas concorrentes utilizando técnicas de BMC, sequencialização de programas e não-determinismo. Segundo, uma abordagem para propor correções de forma a obter uma versão do programa original que não apresente o comportamento defeituoso previamente encontrado. Terceiro, o método apresentado é capaz de assistir desenvolvedores a localizar e corrigir defeitos em programas concorrentes de forma mais rápida, independentemente do seu conhecimento prévio sobre o programa em questão.

1.4 Organização da Dissertação

A dissertação está organizada da seguinte maneira: no Capítulo 2 são apresentados os conceitos básicos de verificação de modelos limitada e programas concorrentes e ao fim do capítulo é feita uma descrição do uso de contraexemplos para localização de falhas e sua aplicação em programas sequenciais; o Capítulo 3 apresenta um resumo dos trabalhos relacionados à verificação de modelos limitada aplicada à programas concorrentes e à localização de falhas em programas e no fim do capítulo, os trabalhos são comparados, salientando as principais diferenças entre a abordagem proposta neste trabalho em relação às existentes; o Capítulo 4 descreve o método proposto para sequencializar programas concorrentes, juntamente da abordagem para localizar as falhas existentes em tais programas, baseando-se em contraexemplos obtidos atra-

vés de um verificador de modelos; no Capítulo 5 são apresentados os resultados de localização de falhas nos programas extraídos da suíte de teste e também é feita uma discussão dos resultados obtidos; e por fim o Capítulo 6 apresenta as conclusões do trabalho, além de apresentar sugestões para trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Neste capítulo, são apresentados os conceitos básicos utilizados durante o desenvolvimento desta dissertação. Primeiramente serão apresentados alguns conceitos importantes sobre a verificação de modelos. Em seguida, será discutido o uso de técnicas de verificação de modelos para auxiliar no processo de localização de falhas, como também será demonstrado um método para localizar falhas em programas sequenciais. Por fim, um resumo do capítulo sintetiza o conteúdo do capítulo.

2.1 Verificação de Modelos

2.1.1 Teoria dos Módulos da Satisfiabilidade (SMT)

SMT (*Satisfiability Modulo Theories*) verifica a satisfatibilidade de fórmulas de primeira ordem a partir de uma ou mais teorias de fundamentação, que são compostas por um conjunto de sentenças. De modo formal, σ – *theory* é uma coleção de sentenças sobre a assinatura σ. Dada uma teoria T, diz-se que φ é um módulo satisfatível de T se $T \subset \phi$. Em outra definição, pode-se dizer que uma teoria T é definida como uma classe de estruturas e φ é um módulo satisfatível se existe uma estrutura M em T que satisfaz φ (*i.e.*, $M \models \varphi$) [17].

Solucionadores SMT como o Z3 [18] e Boolector [19] suportam diferentes tipos de teorias, de modo que o seu desempenho pode variar conforme a sua implementação.

A Tabela 2.1 mostra algumas das teorias suportadas pelos solucionadores SMT utilizados neste trabalho. A teoria de igualdade permite verificações de igualdade e desigualdade entre predicados utilizando os operadores $(=)$ $(\leq)$ $(<)$. A teoria da aritmética linear é responsável

Tabela 2.1: Exemplos de teorias suportadas.

Teoria	Exemplo
Igualdade	$x_1 = x_2 \wedge \neg(x_2 = x_3) \Rightarrow \neg(x_1 = x_3)$
Aritmética Linear	$(7y_1 + y_2 \geq 5) \vee (y_2 + y_3 \leq 2)$
Vetores de *bit*	$(b \gg i)\&1 = 1$
Arranjos	$store(a, j, 5) \Rightarrow a[j] = 5$
Teorias Combinadas	$(j \leq k \wedge a[j] = 2) \Rightarrow a[i] < 3$

apenas pelas funções aritméticas (adição, subtração, multiplicação e divisão) entre variáveis e constantes numéricas. A teoria de vetores de *bit* permite operações *bit* a *bit* considerando diferentes arquiteturas (*e.g.*, 32 e 64 *bits*), nela estão presentes os operadores: e (&), ou (|), ou-exclusivo ($\bigoplus$), complemento ($\sim$), deslocamento para a direita ($\gg$) e deslocamento para a esquerda ($\ll$). Além disso, a teoria de arranjos permite a manipulação de operadores como *select* e *store*.

Em contraste com as fórmulas geradas na satisfação booleana, que são apenas compostas por variáveis booleanas, as quais podem assumir valores verdadeiro e falso e conectivos lógicos, as fórmulas de primeira ordem são formadas por conectivos lógicos, variáveis, quantificadores, funções e símbolos de predicado [17]. De modo geral, as teorias do módulo da satisfatibilidade tem sido aplicadas em diversos cenários [20], apresentando resultados mais promissores (se comparados à satisfação booleana), incluindo o suporte a diferentes teorias de decisão [18, 17].

2.1.2 *Efficient SMT-based Context-Bounded Model Checker*

O ESBMC é um verificador de modelos limitado ao contexto baseados nas teorias de módulo da satisfiabilidade (SMT), o qual é usado para verificar programas ANSI-C [21, 22]. O ESBMC pode verificar tanto programas sequenciais quanto concorrentes e verifica propriedades relacionadas a estouro aritmético, divisão por zero, acesso ilegal a posições em memória, segurança de ponteiros, bloqueios fatais e corrida de dados (do inglês, *data race*), possibilitando também a especificação de propriedades por parte do usuário através de assertivas e lógica temporal de tempo linear [23, 24]. Esse processo é totalmente automático e não requer interação de usuário para anotar programas com pré- e/ou pós-condições.

No ESBMC, o programa a ser verificado é modelado como um sistema de transição de estados $M = (S, R, s_0)$, que é extraído de um gráfico de fluxo de controle (GFC). S representa o conjunto de estados, $R \subseteq S \times S$ representa o conjunto de transições (*i.e.*, pares de estados

que especificam como o sistema pode navegar de um estado para outro) e $s_0 \subseteq S$ representa o conjunto de estados iniciais. Um estado $s \in S$ consiste de um valor do contador de programa *pc* e os valores de todas as variáveis do programa. Um estado inicial s_0 atribui a localização inicial do programa do GFC para o *pc*. Cada transição $\gamma(s_i, s_{i+1}) \in R$ entre dois estados s_i and s_{i+1} é identificada como uma fórmula lógica que captura as restrições nos valores do contador de programa e das variáveis do programa correspondentes.

Dado um sistema de transição M, uma propriedade de segurança ϕ, um limite de contexto C e um limite k, o ESBMC constrói uma árvore de alcançabilidade (AA) que representa o desdobramento do programa para C, k e ϕ. Ele então deriva uma condição de verificação (CV)ψ_k^π, para cada intercalação (ou caminho de computação) dada $\pi = \{v_1, ..., v_k\}$, que é dada pela seguinte fórmula lógica:

$$\psi_k^\pi = \overbrace{I(s_0) \wedge \bigvee_{i=0}^{k} \bigwedge_{j=0}^{i-1} \gamma(s_j, s_{j+1})}^{\text{restrições}} \wedge \overbrace{\neg\phi(s_i)}^{\text{propriedade}} \tag{2.1}$$

Aqui, I caracteriza o conjunto de estados iniciais M e $\gamma(s_j, s_{j+1})$ é a relação de M entre passos de tempo j e $j+1$. Consequentemente, $I(s_0) \wedge \bigvee_{j=0}^{i-1} \gamma(s_j, s_{j+1})$ representa a executação de M de largura i e ψ_k^π pode ser satisfatível se e somente se para algum $i \leq k$ existe um estado alcançável ao longo de π em um passo de tempo i no qual ϕ é violada. ψ_k^π é uma fórmula livre de quantificadores em um subconjunto de lógica de primeira ordem decidível, o qual sua satisfiabilidade é verificada por um solucionador SMT. Se ψ_k^π é satisfatível, então ϕ é violada ao longo de π e o solucionador SMT fornece uma atribuição satisfatória, da qual pode-se extrair valores para variáveis do programa e construir um contraexemplo. Um contraexemplo para uma propriedade ϕ é uma sequência de estados $s_0, s_1, ..., s_k$ com $s_0 \in S_0$, $s_k \in S$, e $\gamma(s_i, s_{i+1})$ para $0 \leq i < k$. Se ψ_k^π não é satisfatível, pode-se concluir que nenhum estado de erro é alcançável em k passos ou menos ao longo de π. Finalmente, pode-se definir $\psi_k = \bigwedge_\pi \psi_k^\pi$ e usá-la para verificar todos os caminhos.

No entanto, o ESBMC combina verificação de modelos simbólica com a exploração explícita do espaço de estados; em particular, ele explicitamente explora todas as possíveis intercalações (até o limite de contexto dado) enquanto ele trata cada intercalação em si simbolicamente. O ESBMC implementa diferentes variações dessa abordagem, que diferem no modo que elas são exploradas na AA. A variação mais eficaz simplesmente percore a AA em pro-

fundidade e chama o procedimento BMC sequencial para cada intercalação quando ela atinge um nodo folha da AA. Ele pára ou quando encontra erro ou sistematicamente explorou todas as possíveis intercalações da AA.

O modelo de memória do ESBMC usa análise estática de ponteiros, preenchimentos em estruturas, com o objetivo de alinhar todos os campos aos limites de palavra, forçamento de regras de alinhamento de acesso de memória e alocação de vetores de bytes, quando o tipo de alocação de memória não é claro, para que a fórmula SMT não seja tão extensa e suscetível a erros.

O ESBMC também implementa a prova por indução [25, 26, 27, 28] para verificar propriedades em programas, *i.e.*, utilizar uma abordagem iterativa de aprofundamento para checar se uma propriedade de segurança ϕ é satisfeita em cada passo k. O ESBMC usa um algoritmo *k-induction*, que consiste de três casos diferentes: caso base, condição adiante e passo indutivo. No caso base tenta-se encontrar um contraexemplo em até k desdobramentos de laço; na condição adiante verifica-se se os laços foram completamente desdobrados e que ϕ é válida em todos os estados alcançáveis dentro de k passos; no passo indutivo assegura-se que sempre que ϕ é válida para k desdobramentos, ela também é válida após o próximo desdobramento do sistema.

Uma outra aplicação do ESBMC é a verificação de programas concorrentes para a plataforma CUDA [29, 30]. Utilizando um modelo operacional, *i.e.*, uma representação abstrata das bibliotecas CUDA padrões que aproxima de forma conservadora as suas semânticas, o ESBMC é capaz de verificar propriedades de segurança em programas CUDA. Além disso, o ESBMC implementa a redução parcial de ordem e a análise de duas *threads* para minimizar a exploração de espaço de estados.

2.2 Localização de Falhas e Verificação de Modelos

2.2.1 Uso de Contraexemplos para Localizar Falhas

Em verificação de modelos, a atividade mais essencial, em relação à localização de falhas, é a de geração de um contraexemplo, o qual é produzido quando um programa não satisfaz uma dada especificação. Um contraexemplo não provê unicamente informações sobre a relação causa-efeito de uma dada violação, mas ele também pode auxiliar na localização de falhas, como Clarke *et al.* [14, 15] citam. Mas, visto que uma grande massa de informação é

obtida em um contraexemplo, as linhas de fato defeituosas não são facilmente identificadas.

Alguns métodos foram propostos, com o objetivo de localizar possíveis causas de falha, usando contraexemplos. Ball *et al.* [31] propuseram uma abordagem que tenta isolar possíveis causas de contraexemplos, gerados pelo verificador de modelos SLAM [32]. A ideia é que potenciais linhas defeituosas podem ser isoladas através de uma comparação entre as transições obtidas em contraexemplos e execuções bem-sucedidas, visto que transições não presentes em rastreamentos bem-sucedidos são potenciais causas de erros. Groce *et al.* [33] afirmam que se um contraexemplo existe, um caminho similar mas não-defeituoso também existe e pode ser obtido usando técnicas de BMC. Elementos de programa relacionados a uma dada violação são sugeridos pelas diferenças entre tal contraexemplo e um caminho bem-sucedido. Tal abordagem é implementada no verificador de modelos *Java PathFinder* [34] e também pode prover caminhos de execução que levam a estados errôneos, com relação a programas concorrentes (*e.g.*, corrida de dados). O conceito chave da abordagem descrita por Groce *et al.* [35] é similar ao anterior e usa alinhamento de restrições para associar estados, em um contraexemplo, com os estados correspondentes em uma execução não-defeituosa, os quais são gerados por um solucionador de restrições. Os estados mencionados são estados abstratos sobre predicados, os quais representam estados concretos em uma execução. Usando propriedades de métricas de distância, restrições podem ser aplicadas para representar execuções do programa, e restrições sem correspondentes que representam estados concretos possivelmente levam a falhas. E ainda, se uma propriedade de métricas de distânica não é satisfeita, um contraexemplo é gerado pelo verificador de modelos [35].

2.2.2 Localização de Falhas em Programas Sequenciais

Griesmayer *et al.* [16] propuseram um método baseado em técnicas de BMC que pode diretamente identificar potenciais falhas em programas. Em particular, o método usa variáveis numéricas adicionais, *e.g.* `diag`, para apontar linhas defeituosas em um dado programa.

Cada atribuição do programa, representanda por `S`, é transformada em uma versão lógica de tal declaração. Logo, o valor atribuído a `S` é ou não-deterministicamente escolhido pelo verificador de modelos (se o valor de `diag` for o mesmo que o representado pela linha relacionada à atribuição `S`) ou o especificado originalmente. Os valores de `diag` obtidos pelo verificador de modelos representam linhas do programa e estão estritamente ligados à falha obtida, visto

que, corrigindo essa linha no programa original, a falha em questão pode ser evitada. No caso de múltiplos valores de `diag`, corrigindo tais linhas levam a uma execução bem sucedida do programa. Com o intuito de encontrar o conjunto inteiro de linhas que causam o comportamento defeituoso no programa, uma nova especificação no comando de verificação[1] pode ser adicionada ao código-fonte, o qual então é executado novamente pelo verificador de modelos. Esse processo é executado repetidamente até que não sejam obtidos novos valores para `diag`, *i.e.*, a execução não falha [36].

Para ilustrar o funcionamento do método em questão, toma-se como exemplo um controlador digital baseado na fórmula da função horária do movimento retilíneo uniformemente variado (MRUV) [37] (veja Equação 2.2). A equação do controlador é definida na Equação 2.3 (os valores foram atribuídos arbritariamente).

$$s(t) = at^2/2 + v_0 t + s_0 \tag{2.2}$$

$$c(t) = t^2 - 3t + 2 \tag{2.3}$$

Um modelo na linguagem C do controlador é modelado como na Figura 2.1.

```
1 #include <stdio.h>
2 #include <assert.h>
3
4 const int A = 1;
5 const int B = -2;
6 const int C = 2;
7
8 int controller(int input) {
9   int output = A * input * input + B * input + C;
10  return output;
11 }
12
13 int main() {
14  assert(controller(0) == 2 && controller(1) == 0
        && controller(2) == 0 && controller(3) ==
       2);
15  return 0;
16 }
```

Figura 2.1: Código sequencial de um controlador qualquer.

[1] `assume(diag != a)`

Pode-se observar que o modelo não está em conformidade com a equação dada, no caso o termo B está com o valor -2 ao invés de -3. Dessa forma, espera-se que a assertiva falhe ao executar o programa em um verificador de modelos, como pode ser observado no trecho da Figura 2.2 (o contraexemplo completo está disponível no Apêndice B).

```
...
Violated property:
  file model.c line 14 function main
  assertion
  FALSE

VERIFICATION FAILED
```

Figura 2.2: Trecho do contraexemplo para o modelo.

Usando o ESMBC como verificador de modelos, o código instrumentado não-determinístico obtido é como na Figura 2.3.

```
#include <stdio.h>
#include <assert.h>
const int A = 1;
const int B = -2;
const int C = 2;
int nondet(int i) {
  int ret;
  __ESBMC_assume(ret != i);
  return ret;
}
int controller(int input) {
  int diag = nondet(0);
  int ta = (diag == 1 ? nondet(A) : A) * input *
      input;
  int tb = (diag == 2 ? nondet(B) : B) * input;
  int tc = (diag == 3 ? nondet(C) : C);
  int output = ta + tb + tc;
  return output;
}
int main() {
  __ESBMC_assume(controller(0) == 2 && controller
      (1) == 0 && controller(2) == 0 &&
      controller(3) == 2);
  assert(0);
  return 0;
}
```

Figura 2.3: Código sequencial instrumentado com o método descrito aplicado.

Ao executar o código da Figura 2.3 no ESBMC sucessivamente, *i.e.*, até que não sejam encontrados novos valores para *diag*, obtém-se os valores presentes na Figura 2.4 (o contrae-

xemplo completo está disponível no Apêndice C).

```
1 griesmayer::controller::1::diag=-2012462479
    (-2012462479)
2 griesmayer::controller::1::diag=2 (2)
3 griesmayer::controller::1::diag=2 (2)
4 griesmayer::controller::1::diag=2 (2)
```

Figura 2.4: Linhas defeituosas obtidas pela execução do código 2.3.

Segundo o contraexemplo obtido com o ESBMC, pode-se observar que o valor de *diag* é 2 em três casos e um inteiro negativo em um caso. Logo, o problema está no cálculo do segundo termo, como esperado. O contraexemplo completo mostra que o valor para corrigir tal falha é -3. Assim, pode-se corrigir a falha apontada e reexecutar o código no verificador de modelos.

```
1 #include <stdio.h>
2 #include <assert.h>
3 const int A = 1;
4 const int B = -3;
5 const int C = 2;
6 int controller(int input) {
7   int output = A * input * input + B * input + C;
8   return output;
9 }
10 int main() {
11   assert(controller(0) == 2 && controller(1) == 0
         && controller(2) == 0 && controller(3) ==
        2);
12   return 0;
13 }
```

Figura 2.5: Código sequencial corrigido.

Após a correção do problema apontado, executa-se o código corrigido 2.5 no ESBMC e obtém-se as linhas presentes na Figura 2.6 (o contraexemplo completo está disponível no Apêndice D). Os valores negativos para *diag* significam que não há uma linha defeituosa no programa, visto que não existem linhas negativas em um código. Em controladores digitais, é importante que os modelos sejam precisamente especificados para evitar falhas durante o funcionamento em ambiente real, visto que podem levar ao mal-funcionamento do equipamento e até danos, aumentando o custo do mesmo.

Dessa forma, foi possível observar o método proposto por Griesmayer *et al.* [16] aplicado em um programa sequencial.

```
1 griesmayer::controller::1::diag=-934770697
    (-934770697)
2 griesmayer::controller::1::diag=-1 (-1)
3 griesmayer::controller::1::diag=-1 (-1)
4 griesmayer::controller::1::diag=-1 (-1)
```

Figura 2.6: Linhas defeituosas obtidas pela execução do código 2.5.

Este método foi escolhido para ser utilizado neste trabalho, pois além de ser simples a sua implementação, ele aponta não só linhas que contém defeitos, como também possíveis valores que levam a uma execução bem-sucedida do programa original.

2.3 Resumo

Neste capítulo, foram introduzidos os conceitos básicos para o entendimento desta dissertação, relacionadas à verificação de modelos. Mais especificamente, explicou-se o conceito de verificação de modelos limitada usando teorias de módulo da satisfiabilidade (SMT) com o verificador de modelos ESBMC (*Efficient SMT-based Context-Bounded Model Checker*), que verifica propriedades de programas sequenciais e concorrentes. Também foram mostradas discussões sobre o uso de contraexemplos para auxiliar no processo de localização de falhas. Por fim, foi apresentado um método para localizar falhas em programas sequenciais, usando não-determinismo para instrumentar atribuições, de forma que o verificador de modelos escolhe o valor para cada variável do programa para que propriedade (em forma de assertiva) presente no código seja satisfeita. Como resultado, o contéudo deste capítulo fornece todo o embasamento necessário para compreensão do trabalho desenvolvido, que será descrito nas seções subsequentes.

Capítulo 3

Trabalhos Relacionados

Neste capítulo, serão descritos onze trabalhos relacionados que direta ou indiretamente abordam localização de falhas em programas concorrentes. Apesar de existirem outros estudos relacionados ao tema, foram selecionados apenas os que se assemelham em algum aspecto a este trabalho.

Cada trabalho relacionado será descrito de acordo com suas características mais relevantes. Ao final de cada subseção serão destacados os melhores aspectos e os pontos fracos de cada um. Por fim, será feito um paralelo das características que são importantes para os objetivos propostos neste trabalho, que servirá como base de comparação.

3.1 *Visualization of Test Information to Assist Fault Localization*

Jones *et al.* [38] apresentam uma abordagem para auxiliar na localização de falhas usando visualização de informação de testes. A ideia é executar uma suíte de testes e colorir declarações do programa relacionadas tanto a casos de teste bem-sucedidos quanto aos que falharam, de forma a apresentar dados visuais para que desenvolvedores sejam capazes de inspecionar execuções do programa, argumentar declarações associadas a comportamentos defeituosos e possivelmente identificar falhas. Os autores também descrevem a ferramenta desenvolvida, denominada TARANTULA, que faz uso da técnica de visualização.

O trabalho mostra que a técnica é útil para auxiliar na depuração de programas, dando uma visão global ao invés de uma apenas local, como o depurador padrão oferece, e o espaço

de busca por falhas dentro do programa é reduzido. No entanto, essa abordagem não é completamente automatizada, visto que ainda é necessário a execução iterativa do programa e seus respectivos casos de teste pelo usuário; linhas defeituosas podem ser marcadas como seguras dependendo do caso de teste; e a ferramenta não é tão eficaz quando um programa apresenta múltiplas falhas.

3.2 *Locating Causes of Program Failures*

Cleve *et al.* [7] discutem uma abordagem para localizar falhas em programas através de buscas em espaço (*i.e.*, estados do programa) e em tempo *i.e.*, transições de causa, que são instantes de tempo onde uma variável deixa de ser a causa de uma falha e uma outra varíavel passa a ser a causa. A comparação dos estados do programa de execuções bem-sucedidas e defeituosas é fundamental para que se possam encontrar os pontos em que transições de causa ocorrem, visto que esses pontos não são apenas bons locais para reparos, mas também apontam os defeitos que causam a falha.

Os autores apresentam uma abordagem eficaz para localizar causas de falhas, de forma que é possível apontar trechos de código que resultam em problemas. Essa técnica foi aplicada dentro de uma ferramenta de depuração de código de código aberto, ASKIGOR [39] e, segundo os resultados apresentados no trabalho, se mostrou mais eficaz que outras técnicas estudadas à epoca. Apesar desses bons aspectos, ainda é necessária uma suíte de teste com alta cobertura de código, já que a técnica depende dessas entradas para encontrar falhas, e também é necessário escolher precisamente o espaço de busca no espaço e tempo para que a falha seja de fato encontrada.

3.3 *Fast Model-based Fault Localisation with Test Suites*

Birch *et al.* [40] mostram um algoritmo para localização de falhas rápida baseada em modelos. Ele executa uma suíte de testes e, com o uso de métodos de execução simbólica, automaticamente identifica um pequeno subconjunto de locais do programa onde reparos são necessários, baseados nos casos de teste malsucedidos. O algoritmo usa limites de tempo para aperfeiçoar a sua velocidade, de forma que se um caso de teste levar mais que o esperado para ser verificado, a execução atual é adiada e uma outra toma o seu lugar. Esse processo melhora

o desempenho geral da técnica, pois uma lista de possíveis locais onde reparos são necessários é mantida e o adiamento da análise para um determinado caso de teste pode posteriormente ser produtiva, uma vez que mais informações sobre o programa pode ser obtidas por meio da análise de outros casos de teste.

O algoritmo apresentado nesse trabalho se mostrou eficiente e otimizado para lidar com suítes de teste unitário extensas, já que ele consegue adiar a procura por falhas baseada em um determinado caso de teste, caso ela demore mais que o esperado. Para a execução simbólica do modelo do programa original, os autores utilizaram as ferramentas KLEE [41] e ESBMC [21]. Apesar de esta abordagem estreitar o espaço de busca por falhas, ela ainda depende inteiramente de uma suíte de testes bem elaborada, visto que caso tal suíte não consiga cobrir a parte do código com problemas, essas falhas não serão encontradas pela técnica.

3.4 *Verifying Concurrent Programs by Memory Unwinding*

Tomasco *et al.* [42] relatam uma abordagem que usa uma técnica chamada *desenrolamento de memória* (do inglês, *memory unwinding* – MU), a qual significa que operações são escritas em uma memória compartilhada, para simbolicamente verificar programas concorrentes que fazem uso de memória compartilhada e criação dinâmica de *threads*. Primeiramente uma possível MU é definida arbitariamente e então todas as execuções do programa que são compatíveis com essa definição são simuladas. Para cada simulação, a ideia é sequencializar programas concorrentes, em relação às regras de MU, e então usar a verificação de modelos no novo código, limitada ao número total de operações de escrita na memória compartilhada, usando uma ferramenta de verificação sequencial já existente. Caso um erro não seja encontrado para essa MU definida, uma nova é gerada e o processo de simulação é feito novamente, até que um erro seja encontrado ou todas as possibilidades tenham sido exploradas.

A abordagem descrita pelos autores é eficaz para verificar o modelo de programas concorrentes, usando um algoritmo de sequencialização que opera de forma gulosa [43], a qual emprega a noção de desenrolamento de memória. A modelagem das primitivas da biblioteca *Pthread* [44] é feita por completo, porém a alocação dinâmica de memória ainda é limitada. Apesar dos resultados do trabalho mostrarem que o algoritmo implementado na ferramenta MU-CSeq foi capaz de encontrar todos os defeitos da suíte de concorrência da *International Competition on Software Verification* 2015, ele consegue apenas assegurar se um erro existe ou

não, sem localizar as linhas que precisam ser consertadas.

3.5 *Verifying Multi-Threaded Software using SMT-based Context-Bounded Model Checking*

Cordeiro *et al.* [45] descrevem três abordagens (*preguiçosa*, *gravação de escalonamento* e *sob aproximação e ampliação*) para verificar programas concorrentes usando o verificador de modelos ESBMC [21], baseado em teoria de módulo da satisfiabilidade (SMT). A primeira abordagem gera todas as possíveis intercalações e chama o solucionador SMT em cada uma delas individualmente. A segunda codifica todas as possíveis intercalações em uma única fórmula e explora a rapidez do solucionador. A terceira reduz o espaço de estados abstraindo o número de intercalações das provas de insatisfiabilidade geradas pelo solucionador SMT. Modelando as primitivas de sincronização da biblioteca *Pthread* [44], o ESBMC cria um programa instrumentado, em relação ao original, e usa verificação de modelos limitada ao número de trocas de contexto nessa nova versão, com o objetivo de encontrar um erro ou explorar todas as intercalações possíveis.

De acordo com os resultados experimentais, este trabalho se mostra eficaz para tratar programas concorrentes, encontrando não somente erros de atomicidade e violação de ordem, como também bloqueios fatais locais e globais. Dentre as três abordagens propostas, a preguiçosa se mostrou mais eficiente que as outras, sendo capaz de verificar todos os programas propostos. Apesar destes pontos, o verificador de modelos pode apenas dizer se um erro existe ou não, e caso exista, ele não pode apontar diretamente onde tal erro se encontra.

3.6 *Automated Fault Localization for C Programs*

Griesmayer *et al.* [16] propuseram um método para localizar falhas em programas sequenciais em ANSI-C utilizando técnicas de BMC. Dados um programa, uma especificação e um contraexemplo para mostrar que a especificação não é satisfeita, ou seja, uma falha existe, os autores usam esse contraexemplo para criar uma versão estendida desse programa. As entradas do programa são fixadas de acordo com os valores do contraexemplo e introduzem predicados anormais para cada componente do programa, gerando uma versão instrumentada do código

original. As variáveis do programa são modeladas não-deterministicamente, de forma a encontrar os valores que satisfaçam a especificação original do programa. O contraexemplo do programa instrumentado contém as linhas que levam à falha e quais valores são necessários para produzir uma execução bem-sucedida do programa.

Um ponto positivo do método é o fato dos contraexemplos gerados pelo verificador de modelos indicarem não somente as linhas defeituosas do programa, mas também os valores necessários a serem atribuidos para as entradas do programa para corrigir tal falha. Apesar disso, o método funciona apenas para programas ANSI-C padrão, *i.e.*, programas procedurais/sequenciais e o tempo de conversão, desenrolamento e geração da representação interna do programa faz com que o tempo para localização das falhas seja alto.

3.7 *Effective Fault Localization Techniques for Concurrent Software*

Park *et al.* [46] apresentam um método de localização de falhas dinâmico para localizar as raízes da causa de defeitos de concorrência e a implementação de um protótipo da técnica, chamado FALCON. Usando detecção dinâmica de padrões e localização de falhas estatística, o FALCON é capaz de mostrar a existência de defeitos em programas concorrentes, tanto de atomicidade quanto violação de ordem, auxiliando desenvolvedores a corrigir falhas em códigos. A técnica utiliza dados providos por casos de teste para o programa em verificação e tenta encontrar padrões pré-estabelecidos de acesso à memória compartilhada. Tais padrões são organizados estatisticamente de forma a priorizar quais as possíveis falhas existentes no programa.

De acordo com o estudo empírico realizado pelos autores, a técnica aparentou ser eficaz para tratar violações de atomicidade e ordem em programa concorrentes, mostrando-se eficiente em termos de espaço e tempo utilizado. No entanto, essa abordagem foi desenvolvida apenas para lidar com programas Java, depende de casos de teste para a procura por padrões defeituosos e, no trabalho em questão, os autores apenas avaliaram a ferramenta com uma entrada de teste e múltiplas execuções do programa.

3.8 *Cause Clue Clauses: Error Localization using Maximum Satisfiability*

Jose *et al.* [47] discutem sobre um algoritmo para localização de causas de erro, considerando uma redução para o *problema da satisfiabilidade máxima* (MAX-SAT), que aponta o número máximo de cláusulas de uma fórmula booleana que uma atribuição pode satisfazer simultaneamente. A ideia chave é combinar uma fórmula de rastreamento booleana e uma fórmula não-satisfatível, ambas em relação ao desdobramento do programa e uma execução do programa malsucedida, e usar MAX-SAT para encontrar o conjunto máximo de cláusulas que podem ser satisfeitas ao mesmo tempo nessa fórmula. O complemento desse conjunto devolvido pelo solucionador MAX-SAT contém os locais do programa que levam ao erro, logo, corrigindo esses locais, é possível conseguir uma execução sem falhas do programa para o caso de teste dado. Vale ressaltar que o uso de MAX-SAT também foi abordado por Trindade *et al.* [48] no ESBMC para avaliar o problema de particionamento de hardware-software.

O algoritmo apresentado é capaz de localizar linhas defeituosas e os autores também realizaram experimentos para sugerir reparos de atribuições aritméticas e troca de operadores de comparação no código original. Apesar dessa abordagem ser útil para localizar linhas defeituosas, ela ainda depende de uma execução malsucedida, e funciona apenas para para programas ANSI-C padrão.

3.9 *Mutation-Based Fault Localization for Real-World Multilingual Programs*

Hong *et al.* [49] propõem uma técnica de localização de falhas baseada em mutantes para programas multilinguais, *i.e.*, programas escritos em mais de uma linguagem que provêm interfaces para comunicação, chamado MUSEUM. Os autores desenvolveram um novo conjunto de operadores de mutação que, junto aos convencionais, são capazes de localizar instruções defeituosas em todos os benchmarks adotados, o que melhora a acurácia ao se tratar de falhas multilinguais. Logo, ao usar esse conjunto de operadores mutantes extendido e métricas de suspeita apropriadas, o MUSEUM é capaz de identificar defeitos em programas JNI.

A avaliação empírica mostra que o MUSEUM é mais preciso e eficaz que técnicas estado-

da-arte de localização de falhas baseadas em espectro. No entanto, a técnica proposta precisa de ao menos um caso de teste que falha e ainda não foi avaliada com programas concorrentes.

3.10 *Metallaxis-FL: Mutation-Based Fault Localization*

Papadakis *et al.* [50] apresentam uma abordagem de localização de falhas baseadas em análise de mutação, chamada *Metallaxis*. A abordagem consiste em usar mutantes e associá-los com potenciais locais defeituosos do programa, e quando mutantes são mortos (principalment por casos de testes que falham), tais são essencialmente bons indicadores sobre os locais defeituosos de um programa. Ainda mais, a abordagem apresentada usa em seu favor um número alto de casos de teste para ordenar instruções mutadas, com relação à sua pontuação de suspeita, o qual leva a falhas no programa. De acordo com dados experimentais, *Metallaxis* é substancialmente mais efetivo que abordagens baseadas em instruções, até quando técnicas de redução de custo de mutação (*e.g.*, amostragem de mutação) são usada. *Metallaxis* localiza falhas efetivamente e, visto que ele funciona sobre uma suíte de teste, ele só pode ser usado durante o teste para localizar potenciais falhas.

A abordagem se mostra superior a abordagens baseadas em instruções; no entanto, ela ainda deende de uma extensa suíte de teste e só foi avaliada em software ANSI-C procedural/-sequencial.

3.11 *Understanding Programming Bugs in ANSI-C Software Using Bounded Model Checking Counter-Examples*

Rocha *et al.* [11] propõem uma ferramenta para automatizar a coleta de contraexemplos, obtidos por meio de um verificador de modelos limitado, e a manipulação dos mesmos com o intuito de gerar um novo programa instanciado para reproduzir o erro identificado. A metodologia proposta consiste em primeiramente pré-processar o programa de entrada, de forma a deixá-lo em um padrão de formatação, e então verificar esse novo programa no ESBMC [21]. Caso um contraexemplo seja obtido durante a verificação do programa, lê-se os valores de variáveis que levam à violação e gera-se um novo código que reproduz o erro encontrado, para cada propriedade violada em tal contraexemplo. Finalmente, cada programa gerado no passo

anterior é executado, de forma a comprovar a falha existente.

A ferramenta apresentada é capaz de reproduzir erros apontados por um verificador de modelos limitado, tornando o processo de identificação e comprovação de falhas automático. No entanto, como o método depende da capacidade do verificador de modelos, é possível deparar-se com o problema da explosão de estados, fazendo com que o usuário tenha que manipular parâmetros e/ou a forma de verificação do código, e em algumas situações, como os autores descrevem, isso pode levar a resultados inválidos.

3.12 Comparação entre os Trabalhos

Os trabalhos relacionados tratam ou de localização de falhas em programas ou em verificação de programas concorrentes. Foi possível observar que o problema de localização de falhas é pertinente, visto que este processo toma muito tempo dos desenvolvedores no ciclo de desenvolvimento de software. O tratamento de programas concorrentes também foi observado como um problema complexo de ser tratado devido ao grande número de possibilidades nas tentativas de simular um comportamento defeituoso. Tendo esses pontos em mente, as principais diferenças entre a abordagem proposta nesse trabalho para as aqui discutidas podem ser listadas: o método proposto requer apenas o código-fonte do programa, em contraste com outros métodos, onde mais dados são necessários, como uma execução malsucedida; o método proposto funciona para programas concorrentes em C, amplamente usados em sistemas embarcados; e, finalmente, é possível apontar linhas defeituosas de forma metódica, enquanto que outras abordagens verificam a segurança de um programa. A Tabela 3.1 mostra um quadro comparativo entre o método proposto nesta dissertação e os trabalhos relacionados selecionados.

Abordagens de localização de falhas baseadas em espectro (SBFL), tal como a proposta por Jones *et al.* [38], tentam identificar blocos de código suspeitos, *i.e.*, aqueles que são executados quando um caso de teste falha e leva a falhas no programa. Por um lado, técnicas SBFL não realizam análises semânticas complexas em programas, visto que as pontuações de suspeita são calculadas usando apenas informações provenientes da execução de suítes de teste. Por outro lado, a acurácia de abordagens SBFL são geralmente abaixo do desejado, quando aplicadas em programas reais grandes e complexos [49].

Abordagens de localização de falhas baseadas em mutantes (MBFL), tal como a apresentada por Hong *et al.* [49], também tentam encontrar blocos de código suspeitos, mas elas geram

Tabela 3.1: Comparação dos trabalhos relacionados

Trabalhos relacionados	**Localiza falhas**	**Aplicável em programas concorrentes**	**Verifica código C**	**Necessita de casos de teste**
Jones *et al.* (2009)	X		X	X
Cleve *et al.* (2005)	X		X	X
Birch *et al.* (2015)	X		X	X
Tomasco *et al.* (2015)		X	X	
Cordeiro *et al.* (2011)		X	X	
Griesmayer *et al.* (2007)	X		X	
Park *et al.* (2014)	X	X		
Jose *et al.* (2011)	X		X	
Hong *et al.* (2015)	X		X	X
Papadakis *et al.* (2015)	X		X	X
Rocha *et al.* (2012)	X		X	
Método proposto	X	X	X	

mutantes para tais blocos e executam novamente os casos de teste: se um bloco de código mutante é morto, então o bloco de código tem uma pontuação de suspeita mais alta. Esse esquema de mutantes geralmente supera a performance de abordagens SBFL [49], mas ele também leva mais tempo para gerar resultados úteis, visto que há uma variedade de possíveis mutantes para expressão de código.

Dado o conhecimento atual em localização de falhas, não há outra técnica de localização de falhas que combine sequencialização, não-determinismo, e contraexemplos de verificação de modelos para solucionar o problema e que aborde programas concorrentes. Ainda mais, aplicando as abordagens deste trabalho em outras soluções não seria viável, visto que o passo de introdução de não-determinismo faz com que haja uma dependência de um verificador de modelos, anulando os passos de outras técnicas. Portanto, não é possível comparar o método proposto diretamente com abordagens SBFL e MBFL, visto que seria necessário reimplementá-las para abrangerem programas C baseados na biblioteca *Pthread*, o que poderia introduzir erros e resultados inconclusivos.

3.13 Resumo

Neste capítulo foram apresentados diversos trabalhos relacionados a algum dos seguintes problemas: localização de falhas, suporte a programas concorrentes, verificação de código

escrito na linguagem C e/ou dependência de casos de testes externos. Após a apresentação de cada técnica, assim como uma breve discussão sobre as vantagens e desvantagens de cada uma delas, foi feita uma comparação com o método proposto neste trabalho. Foi possível observar que o método proposto aborda os três primeiros itens sem a necessidade de casos de testes externos, o que o diferencia dos demais trabalhos. O próximo capítulo apresenta a metodologia proposta para localizar falhas em programas concorrentes utilizando técnicas de sequencialização e BMC.

Capítulo 4

A Metodologia Proposta

Neste capítulo, o método proposto para localizar falhas em programas concorrentes em C será completamente descrito. Primeiramente, um exemplo de motivação será apresentado para descrever a abordagem. Os passos anteriores à localização de falhas, onde técnicas de BMC são aplicadas, são mostrados a seguir. Também são descritas as regras de transformação para sequencializar um programa concorrente. Finalmente, uma explicação é dada sobre o processo de localização de falhas no programa concorrente transformado.

4.1 Visão Geral do Método

Nesta seção será descrito brevemente o método proposto, como mostrado na Figura 4.1, e uma explicação mais detalhada é exposta nas seções seguitnes. Dado um programa concorrente P, primeiramente checa-se se ele apresenta uma execução mal-sucedida com relação a uma determinada intercalação. Para realizar tal tarefa, P é executado em um verificador de modelos duas vezes: a primeira execução verifica se existe algum bloqueio fatal e a segunda é responsável por outros tipos de violação, tais como erros de aquisição de semáforo, divisão por zero, segurança de ponteiros, estouro aritmético e violação de limites de vetores. Não é possível verificá-lo apenas uma vez porque os verificadores de modelos separam essas verificações, *i.e.*, é necessário adicionar uma opção de linha de comando para habilitar a detecção de bloqueios fatais e ignorar violações devido a assertivas. Caso um contraexemplo possa ser obtido nesse passo, é possível prosseguir com o método. Caso contrário, o programa é marcado como correto pelo verificador de modelos e não há falhas a serem procuradas. Então, o próximo passo

define as regras de transformação, que são as instruções sequenciais que substituirão as originais concorrentes, e um arcabouço sequencial, o qual tem como objetivo simular a execução concorrente da intercalação mal-sucedida. O terceiro passo consiste em usar o método proposto por Griesmayer [16] para instrumentar atribuições e expressões, de forma a apontar locais e instruções defeituosas do programa. Tal programa instrumentado pode então ser executado, usando um verificador de modelos, e é possível coletar as linhas defeituosas, até que a verificação associada não produza diferentes elementos. Essa iteração é descrita na Figura 4.1 por meio dos passos 3, 4, já que o método busca novos contraexemplos, com o intuito de encontrar novas linhas defeituosas. Finalmente, as linhas defeituosas são obtidas, assim como atribuições necessárias para produzir uma execução bem-sucedida de *P*.

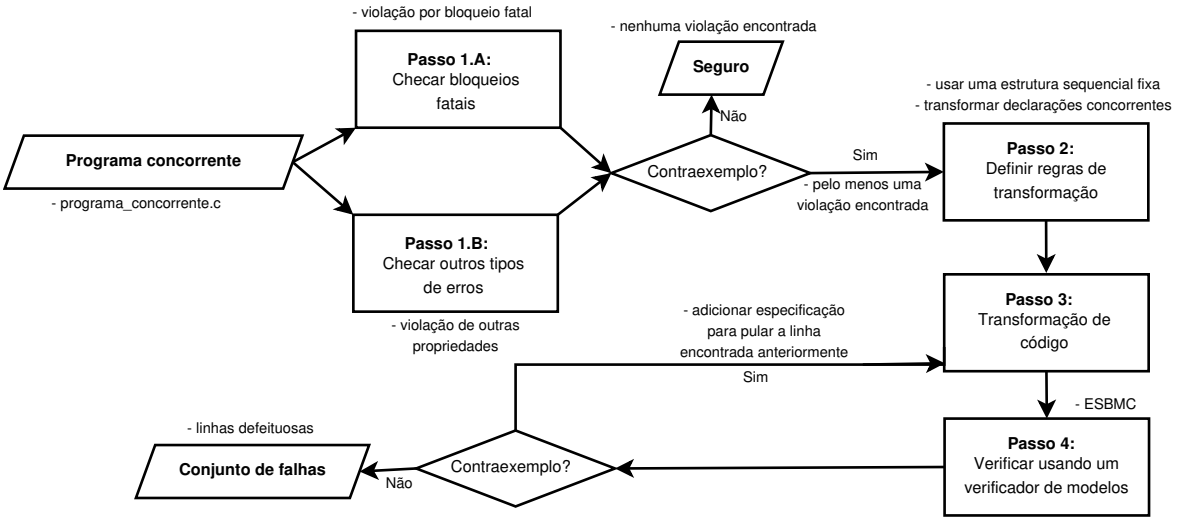

Figura 4.1: Metodologia proposta.

4.2 Exemplos Motivacionais

Dois programas concorrentes simples são utilizados para ilustrar a abordagem proposta. O primeiro (ver Figura 4.2) tem duas variáveis compartilhadas, os semáforos `mutex` e `lock`, usados para sincronizar as *threads* `A` e `B`. A função `main` inicializa cada contador da sua respectiva *thread* e começa a executar duas *threads*, executando as funções `threadA` e `threadB`, respectivamente. Cada função de *thread* adquire o semáforo `mutex`, incrementa o seu respectivo contador (`A_count` ou `B_count`), verifica se é possível adquirir o semáforo `lock` (caso seu contador seja igual a um), libera o semáforo `mutex`, tenta adquirir o semáforo `mutex` logo em seguida, decrementa o seu contador, verifica se é possível liberar o semáforo `lock` (caso seu

contador seja zero), e finalmente libera o semáforo `mutex`. Visto que não há assertivas, não se deve obter outros tipos de violações, a não ser erros de concorrência. Pode-se notar que o controle de acesso ao semáforo `lock` é feito de forma local. Assumindo que trocas de contexto podem ocorrer em qualquer linha do programa, é provável que uma execução específica leve a um erro de bloqueio fatal.

```
1  void *threadA(void *arg) {
2    pthread_mutex_lock(&mutex);
3    A_count++;
4    if (A_count == 1) pthread_mutex_lock(&lock);
5    pthread_mutex_unlock(&mutex);
6    pthread_mutex_lock(&mutex);
7    A_count--;
8    if (A_count == 0) pthread_mutex_unlock(&lock);
9    pthread_mutex_unlock(&mutex);
10 }
11 void *threadB(void *arg) {
12   pthread_mutex_lock(&mutex);
13   B_count++;
14   if (B_count == 1) pthread_mutex_lock(&lock);
15   pthread_mutex_unlock(&mutex);
16   pthread_mutex_lock(&mutex);
17   B_count--;
18   if (B_count == 0) pthread_mutex_unlock(&lock);
19   pthread_mutex_unlock(&mutex);
20 }
21 int main() {
22   pthread_t A, B;
23   A_count = 0; B_count = 0;
24   pthread_create(&A, NULL, threadA, NULL);
25   pthread_create(&B, NULL, threadB, NULL);
26   pthread_join(A, NULL);
27   pthread_join(B, NULL);
28   return EXIT_SUCCESS;
29 }
```

Figura 4.2: Primeiro exemplo motivacional.

Para transformar um código concorrente para sequencial, deve-se, em linhas gerais, transformar declarações de programa concorrentes para versões sequenciais que tenham o mesmo comportamento e adicionar uma estrutura fixa para que o novo programa sequencial execute da mesma forma que o original, processo que será descrito nas próximas seções. Aplicando tais passos no código da Figura 4.2, tem-se como resultado o código mostrado nas Figuras 4.3 e 4.4. Com o método aplicado no código concorrente, o restante do processo de localização de falhas depende apenas do verificador de modelos em uso, que retornará informações úteis relacionadas às falhas existentes no programa (o contraexemplo simplificado para esse código está disponível

no Apêndice E).

```
1 ... int non_det(), diag;
2 int A_count, B_count;
3 h_mutex mutex, lock;
4 void A_1(void *arg) {
5   int t;
6   h_cs cs;
7   h_lock(&mutex, &cs, 1, 9);
8   t = A_count;
9   A_count = (diag == 10 ? non_det() : t + 1);
10  if ((diag == 11 ? non_det() : A_count) == 1)
11    h_lock(&lock, &cs, 1, 11);
12  h_unlock(&mutex, &cs, 1, 12);
13 }
14 void A_2(void *arg) {
15  int t;
16  h_cs cs;
17  h_lock(&mutex, &cs, 1, 13);
18  t = A_count;
19  A_count = (diag == 14 ? non_det() : t - 1);
20  if ((diag == 15 ? non_det() : A_count) == 0)
21    h_unlock(&lock, &cs, 1, 15);
22  h_unlock(&mutex, &cs, 1, 16);
23 }
24 void B_1(void *arg) {
25  int t;
26  h_cs cs;
27  h_lock(&mutex, &cs, 2, 20);
28  t = B_count;
29  B_count = (diag == 21 ? non_det() : t + 1);
30  if ((diag == 22 ? non_det() : B_count) == 1)
31    h_lock(&lock, &cs, 2, 22);
32 }
33 void B_2(void *arg) { ... } ...
```

Figura 4.3: Método aplicado ao exemplo da Figura 4.2 (parte 1).

O segundo (ver Figura 4.5) contém duas variáveis globais, `x` e `y`, utilizadas por cada uma das *threads* existentes no programa. A primeira *thread* incrementa o valor de `x` e a segunda decrementa o valor de `y`. Ambas *threads* são executadas e finalmente é verificada uma propriedade, *i.e.*, ambas as variáveis devem ter um valor positivo.

Utilizando os mesmos passos ilustrados no exemplo anterior, obtém-se o código da Figura 4.5, tem-se como resultado o código mostrado na Figura 4.6. De forma similar, o verificador de modelos retornará informações úteis relacionadas às falhas existentes no programa (o contraexemplo simplificado para esse código está disponível no Apêndice F), assim como valores possíveis para reproduzir uma execução bem-sucedida do mesmo. A seguir, será explicada a metodologia por completo, resumida na Figura 4.1.

```
1  ... #define NCS 4
2  int cs[] = {11, 21, 31, 22};
3  int main() {
4    int i;
5    diag = non_det();
6    for (i = 0; i < NCS; i++) {
7      switch (cs[i]) {
8        case 1: {
9          case 11: {
10           A_count = 0; B_count = 0;
11           if (cs[i] == 11) break;
12         }
13       } break;
14       case 2: {
15         case 21: {
16           A_1(NULL);
17           if (cs[i] == 21) break;
18         }
19         case 22: {
20           A_2(NULL);
21           if (cs[i] == 22) break;
22         }
23       } break;
24       case 3: {
25         case 31: {
26           B_1(NULL);
27           if (cs[i] == 31) break;
28         }
29       } break;
30     }
31   }
32   assert(0);
33 }
```

Figura 4.4: Método aplicado ao exemplo da Figura 4.2 (parte 2).

4.3 Uso de BMC para Auxiliar na Localização de Falhas

4.3.1 Assegurando a Existência de Falhas em Programas Concorrentes

De forma que seja possível aplicar a metodolgia proposta, é necessário assegurar a existência de falhas do programa concorrente P sob verificação (**Passo 1.A** e **Passo 1.B** da Figura 4.1). Para alcançar tal objetivo, é necessário pelo menos uma propriedade violada (erro de bloqueio fatal, assertiva, aquisição de semáforo, divisão por zero, segurança de ponteiros, estouro aritmético, e/ou violação de limites de vetores) e o seu respectivo contraexemplo (veja Seção 2.1.2 para mais detalhes), informação que pode ser obtida por meio da verificação de P em um verificador de modelos duas vezes. Se um contraexemplo para P não puder ser encontrado, ou o limite k é aumentado (limitado aos recursos de máquina disponíveis) ou afirma-se

```
1  ... int x = 0, y = 0;
2  void* sum_x(void* args) {
3      x++;
4      return NULL;
5  }
6  void* sum_y(void* args) {
7      y--;
8      return NULL;
9  }
10 int main(void) {
11     pthread_t t1, t2;
12     pthread_create(&t1, NULL, sum_x, NULL);
13     pthread_create(&t2, NULL, sum_y, NULL);
14     pthread_join(t1, NULL);
15     pthread_join(t2, NULL);
16     assert(x > 0 && y > 0);
17     return 0;
18 }
```

Figura 4.5: Segundo exemplo motivacional.

que P é seguro (P não contém falhas) até a profundidade k.

4.3.2 Extração de Informações das Trocas de Contexto de Contraexemplos

Com posse de um contraexemplo C_{ex} para P, é necessário extrair as informações de trocas de contexto para posteriormente aplicá-las com o objetivo de encontrar um programa sequencial P_{seq} que reproduza o mesmo comportamento defeituoso que P tem (**Passo 2** da Figura 4.1).

Para que seja possível obter tal informação, seja C_{ex} composto por um conjunto de estados $s_0, s_1, ..., s_k$. Cada estado s_i contém a linha l_{s_i} e a *thread* T_{s_i} a qual tal estado pertence. Anotando a tupla (T_{s_i}, l_{s_i}) onde $T_{s_i} \neq T_{s_{i+1}}$ resulta nas trocas de contexto que ocorrem em P para o dado comportamento defeituoso. Por exemplo, em relação ao programa da Figura 4.2, a informação de troca de contexto obtida do seu respectivo contraexemplo é $CS = [(0,27),(1,5),(2,14),(1,6)]$, onde a *thread* 0 representa a função `main`, a *thread* 1 representa a *thread A* e a *thread* 2 representa a *thread B*.

```
1  ...
2  int main(void) {
3      int cs[] = { 11, 21, 31, 12 };
4      int ncs  = 4;
5      int diag = nondet_int();
6      int x, y;
7      for (int i = 0; i < ncs; i++) {
8          switch (cs[i]) {
9              case 1: {
10                 case 11: {
11                     x = 0;
12                     y = 0;
13                     if (cs[i] == 11) break;
14                 }
15                 case 12: {
16                     if (cs[i] == 12) break;
17                 }
18             } break;
19             case 2: {
20                 case 21: {
21                     int temp = x + 1;
22                     x = (diag == 7 ? nondet_int()
                           : temp);
23                     if (cs[i] == 21) break;
24                 }
25             } break;
26             case 3: {
27                 case 31: {
28                     int temp = y - 1;
29                     y = (diag == 12 ? nondet_int
                          () : temp);
30                     if (cs[i] == 31) break;
31                 }
32             } break;
33         }
34     }
35     __ESBMC_assume(x > 0 && y > 0);
36     assert(0);
37     return 0;
38 }
```

Figura 4.6: Método aplicado ao exemplo da Figura 4.5.

4.4 Sequencialização de Programas Concorrentes

4.4.1 Gramática de Transformação das Primitivas de Sincronização da Biblioteca *Phtread*

Levando em consideração a sequencialização, é necessário desenvolver uma gramática para transformar as declarações de programa originalmente concorrentes para obter uma ver-

são sequencial de tais declarações, mantendo suas respectivas funcionalidades (**Passo 2** da Figura 4.1).

Primeiramente, uma estrutura do tipo `struct` foi desenvolvida para representar trocas de contexto. São armazenadas as variáveis `thread_ID`, um identificador para a *thread* onde a troca de contexto ocorreu, e `program_line`, um identificador para a linha do programa onde a troca de contexto ocorreu. Essa estrutura é chamada `h_cs` e é definida como na Figura 4.7.

```
1 typedef struct
    h_context_switch {
2   int thread_ID;
3   int program_line;
4 } h_cs;
```

Figura 4.7: Estrutura que representa uma troca de contexto.

Então, desenvolveu-se uma estrutura do tipo `struct` para representar uma variável do tipo `pthread_mutex_t`. São armazenadas as variáveis `status`, um identificador para dizer se tal semáforo está adquirido ou não, e `last_cs`, identificando as informações do programa relacionadas à aquisição ou liberação de tal semáforo. Essa estrutura é chamada `h_mutex` e é definida como na Figura 4.8.

```
1 typedef struct h_mutex
      {
2   int status;
3   h_cs last_cs;
4 } h_mutex;
```

Figura 4.8: Estrutura que representa o tipo `pthread_mutex_t`.

Implementações para a manipulação de semáforos, que são a aquisição de um semáforo, `pthread_mutex_lock`, e a liberação de um semáforo, `pthread_mutex_unlock`. A função que representa a aquisição de um semáforo tem 4 argumentos, que são `m` (o semáforo que uma *thread* está tentando adquirir), `cs` (uma variável de troca de contexto), `id` (um identificador para a *thread* que chamou a função de aquisição), e `line` (um identificador para a linha do programa de onde a função foi chamada). A função de liberação também tem 4 argumentos, que são `m` (o semáforo que uma *thread* está tentando liberar), `cs` (uma variável de troca de contexto), `id` (um identificador para a *thread* que chamou a função de liberação), e `line` (um identificador para a linha do programa de onde a função foi chamada). A função de aquisição é chamada `h_lock` e

é definida como na Figura 4.11. A função de liberação é chamada `h_unlock` e é definida como na Figura 4.12.

Variáveis condicionais são implementadas em *pthread* como na Figura 4.9. De forma a simular o mesmo comportamento, a execução não precisa ser parada. Visto que se simula apenas um comportamento defeituoso, a condição necessária é assumida como já satisfeita, então basicamente muda-se a variável do tipo `pthread_cond_t` para o tipo inteiro, a função `pthread_cond_signal` atribui 0 a tal variavel, e a função `pthread_cond_signal` atribui 1. Esse processo é descrito na Figura 4.10, como uma modelagem do código da Figura 4.9.

```
1 pthread_cond_t c;
2 ...
3   pthread_cond_signal(&c,
        &m);
4 ...
5 while (!condition) {
6   pthread_cond_wait(&c, &m
        );
7 }
```

Figura 4.9: Uso de variáveis condicionais padrão.

```
1 int c;
2 ...
3   c = 0;
4 ...
5 while (!condition) {
6   c = 1;
7   break;
8 }
```

Figura 4.10: Modelagem de condicionais no método proposto.

A Tabela 4.1 resume a gramática desenvolvida neste trabalho para transformar programas concorrentes em sequenciais. A coluna *"#"* representa os dois grupos de instruções de programas, *i.e.*, (1) normal e (2) concorrente, a coluna *"Fragmento de código"* mostra o código a ser transformado, e as colunas *"Sem bloqueio fatal"* and *"Com bloqueio fatal"* dizem qual regra de transformação precisa ser executada, caso um programa apresente ou não um bloqueio fatal, respectivamente. O símbolo ε indica que é necessário remover a declaração correspondente.

```
1  void h_lock(h_mutex *m,
2         h_cs *cs,
3         int id,
4         int line) {
5    int status = m->status;
6    if (status == 1) {
7      __VERIFIER_error();
8    } else {
9      cs->thread_ID = id;
10     cs->program_line = line;
11     m->status = 1;
12     m->last_cs.thread_ID = cs->
           thread_ID;
13     m->last_cs.program_line = cs->
           program_line;
14   }
15 }
```

Figura 4.11: Transformação da função `pthread_mutex_lock` na função `h_lock`.

```
1  void h_unlock(h_mutex *m,
2           h_cs *cs,
3           int id,
4           int line) {
5    if (m->status == 1 &&
6        m->last_cs.thread_ID == id) {
7      cs->thread_ID = id;
8      cs->program_line = line;
9      m->status = 0;
10     m->last_cs.thread_ID = cs->
           thread_ID;
11     m->last_cs.program_line = cs->
           program_line;
12   } else {
13     __VERIFIER_error();
14   }
15 }
```

Figura 4.12: Transformação da função `pthread_mutex_unlock` na função `h_unlock`.

4.4.2 Adição de uma Estrutura Fixa para Simulação

Uma estrutura fixa provê a mesma sequência de execução presente no programa original. Ela consiste basicamente em escrever cada código de *thread* dentro de um bloco condicional `case`, e as suas respectivas sequências de execução são especificadas no vetor `cs`. Tal estrutura é usada como a estrutura base para novas versões sequenciais de programas concorrentes (**Passo 2** da Figura 4.1) e a Figura 4.13 mostra como é tal codificação.

Como se pode notar, a estrutura fixa mencionada provê novas posições fixas para cada parte do código original e a Tabela 4.2 mostra a relação entre as novas posições e o tipo de

Tabela 4.1: Gramática para transformar programas concorrentes

#	Fragmento de código	Sem bloqueio fatal	Com bloqueio fatal
1	Declaração Expressão Instrução	Sem alterações Desdobramento Sem alterações	Sem alterações Desdobramento Sem alterações
2	`pthread_t`	ε	ε
	`pthread_attr_t`	ε	ε
	`pthread_cond_attr_t`	ε	ε
	`pthread_create`	ε	ε
	`pthread_join`	ε	ε
	`pthread_exit`	ε	ε
	`pthread_mutex_t`	ε	Semáforo é declarado
	`pthread_mutex_lock`	ε	Nova função de aquisição é chamada usando a variável
	`pthread_mutex_unlock`	ε	Nova função de liberação é chamada usando a variável
	`pthread_cond_t`	ε	Variável condicional é declarada
	`pthread_cond_wait`	ε	Parada é chamada usando a variável
	`pthread_cond_signal`	ε	Sinalização é chamada usando a variável

fragmento de código, isto é, ela resume como o novo código sequencial é estruturado. Em particular, elementos globais, variáveis globais, declarações de arquivos de bibliotecas e outros tipos de declarações globais são posicionadas antes da função `main` do código sequencial. O corpo da sua função `main`, do código original, é posicionado entre a declaração do `case 1` e seu respectivo comando `break`, o corpo da primeira *thread* é posicionado entre a declaração do `case 2` e seu respectivo comando `break`, e assim por diante. Esse processo é repetido até que não existam mais *threads* para serem inseridas na versão sequencial do código. Adicionalmente, os argumentos passados para a função `main` do programa original são todos passados para a função `main` da versão sequencial. Em casos onde *threads* são parcialmente executadas, uma troca de contexto ocorre, outra *thread* é executada ou uma *thread* anterior continua a execução do ponto onde a mesma parou, os respectivos trechos de código são inseridos em cada bloco `case` dentro do bloco maior N^{th} `case` (o N^{th} `case` representa a N^{th} *thread*), de tal forma que a ordem de execução permance a mesma.

Tabela 4.2: Relação entre as posições no programa e o código original

Tipo de fragmento de código no código original	Posição no novo código sequencial
elementos globais	antes da linha 1
corpo da função principal	entre o ”case 1” e o ”break”
corpo da *thread n*	entre o ”case $n+1$” e o ”break”

De forma a manter a mesma ordem de execução encontrada no programa original, o

```
1 #define NCS X
2 int cs[] = {...};
3 int main(int argc, char *argv[]) {
4   int i;
5   for(i = 0; i < NCS; i++) {
6     switch(cs[i]) {
7       case 1:
8           case 11: { ... }
9           ...
10          case 20: { ... }
11      break;
12      case 2:
13          case 21: { ... }
14          ...
15          case 30: { ... }
16      break;
17      case 3:
18          case 31: { ... }
19          ...
20          case 40: { ... }
21      break;
22      ...
23      default:
24      break;
25    }
26  }
27  return 1;
28 }
```

Figura 4.13: A estrutura padrão para sequencialização de programas concorrentes.

controle da ordem do `switch` é necessário. Uma ordem de trocas de contexto, obtida por meio de um contraexemplo do programa concorrente, pode ser copiada para o novo código sequencial, controlando os blocos `case` e as declarações condicionais[1] dentro do bloco `switch`. Em linhas gerais, o processo de adição de controle de ordem de trocas de contexto para o novo programa sequencial pode ser dividido em dois passos. Com o intuito de mostrar uma situação simples de tal passo, assume-se que existem no máximo 10 trocas de contexto em cada *thread* ($\forall N_{ti}, N_{ti} < 10$), um contraexemplo, dado pelo verificador de modelos, tem N trocas de contexto, e dentre essas N trocas de contexto, N_{t0} ocorrem na função `main` do programa, N_{t1} ocorrem na *thread* 1, N_{t2} na *thread* 2, e assim por diante, de forma que $(N_{t0} + ... + N_{tn} = N)$.

O primeiro passo é obter informações dos contraexemplos gerados pelo verificador de modelos, *i.e.*, o número total de trocas de contexto no programa original e em cada *thread*, a ordem de todas as trocas de contexto por todo o programa e também em cada *thread* isolada, e a posição correspondente onde uma troca de contexto ocorreu. Com tais dados, é possível

adicionar declarações condicionais[1] para manter a mesma ordem de execução do programa original, de forma que quando uma linha é executada, o código sequencial executa o próximo bloco `case`, o qual representa a próxima *thread* no programa original.

Pode-se notar que se existem laços iterativos no programa original concorrente, para cada laço, uma variável global `loopcounter` é adicionada. Além disso, a declaração para incrementar o valor da variável `loopcounter` também é adicionado ao fim do bloco de cada laço. Essa nova variável global adicionada é usada como uma condição para controlar diretamente os comandos `break`, de forma que quando uma troca de contexto ocorre, dentro de um laço, o valor atribuído à variável `loopcounter` também deve ser respectivamente usado no comando `break`, de forma a manter a sequência de execução original do programa.

O segundo passo consiste em modificador os valores relacionados ao vetor `cs`, de tal forma que a ordem de execução é mantida, no novo programa sequencial. Mudando as linhas 1 e 2, na Figura 4.13, de acordo com o número específico de trocas de contexto existentes e as suas respectivas ordens de execução, é possível garantir a ordem de execução original, visto que o bloco `switch` (linha 6) seleciona qual trecho de código (representando *threads* do programa original) é executado, baseado no valor de `cs[i]`.

Por exemplo, na Figura 4.2, a ordem de execução é: *thread* 0, *thread* 1, *thread* 2 e, finalmente *thread* 1 (essa informação é obtida através da análise do contraexemplo gerado pelo verificador de modelos, como descrito na Seção 4.3.2). O vetor `cs` terá os valores 11, 21, 31 e 22, conforme a Figura 4.4 (vale ressaltar que tais valores são diferentes dos valores disponíveis no Apêndice E devido a redução do código original, em termos de linhas em branco e importações de bibliotecas para melhor apresentação do mesmo neste trabalho), significando que o primeiro `case` será executado, então o primeiro `case` mais interno dentro do segundo, o terceiro e, por último, o segundo `case` mais interno dentro do segundo.

Vale ressaltar que a tarefa de considerar todos as possíveis intercalações é atribuída ao verificador de modelos, já que o mesmo procurará alguma intercalação que não satisfaça alguma propriedade de segurança. O método em questão é responsável por, dada uma intercalação malsucedida, encontrar as linhas envolvidas na falha presente no programa.

[1] `if (cs[i]) == Y) break;`, onde `Y` representa o número da troca de contexto relacionada

4.5 Aplicação de um Método Sequencial para Localizar Falhas

Finalmente, o método proposto por Griesmayer [16] é aplicado (**Passo 3** da Figura 4.1). Em linhas gerais, cada atribuição em P é convertida para uma versão não-determinística dela mesma e esse valor é escolhido pelo verificador de modelos (**Passo 4** da Figura 4.1) e também é relacionado à variável de diagnóstico, *i.e.*, `diag`. Desta forma, se um contraexemplo for obtido para P_{seq}, existem valores para `diag` em tal rastro, os quais podem compor o conjunto de linhas defeituosas em P. A correção de tais linhas leva a uma execução bem-sucedida de P. Por exemplo, na Figura 4.5, o valor obtido para *diag* é 12. Isso significa que esta linha no programa original causa a falha da assertiva. De fato, ao analizar o programa é possível perceber que a assertiva espera um valor positivo para `y`, mas a operação de decremento na linha 12 da Figura 4.5 impossibilita isso.

4.6 Resumo

Neste capítulo foi apresesentado o método proposto para localizar falhas em programas concorrentes usando técnicas de verificação de modelos e sequencialização, sendo esta a contribuição maior do presente trabalho. Foi dada uma explicação detalhada das transformações necessárias, assim como um exemplo para ilustrar melhor o processo de localização de falhas. Mostrou-se também a importância do contraexemplo obtido antes da aplicação do método proposto, visto que o mesmo contém a ordem das *threads* executadas e os pontos onde trocas de contexto ocorreram. A modelagem necessária para transformar declarações de programa concorrentes em sequenciais também foi explicada, assim como a estrutura fixa necessária para reproduzir a mesma ordem de execução do programa original. Por fim, um método sequencial pode ser aplicado no novo código para obter as linhas que levam às falhas do programa. Como resultado, tem-se o embasamento para a avaliação experimental realizada no próximo capítulo.

Capítulo 5

Resultados e Discussões

Este capítulo está dividido em três partes. A primeira parte descreve os objetivos dos experimentos conduzidos neste trabalho. A segunda é dedicada à descrição da configuração na qual os experimentos foram realizados, incluindo programas, versões e ambientes. A terceira apresenta os resultados obtidos quando se realizaram os experimentos com os benchmarks selecionados, assim como uma discussão sobre os resultados obtidos e avaliação geral do método proposto nesta dissertação.

5.1 Objetivos do Experimento

Utilizando os benchmarks propostos, o experimento tem os seguintes objetivos:

a) Demonstrar a aplicabilidade da metodologia para a localização de falhas em programas concorrentes em C.

b) Avaliar o tempo usado pelo verificador de modelos para verificar o código gerado pelo método proposto.

5.2 Configuração Experimental

De forma a verificar e validar o método proposto, foram usados o ESBMC v3.0.0 com o solucionador SMT Boolector [19] e o Lazy-CSeq v1.1, com o CBMC [51] v5.3 (backend). Todos os experimentos foram conduzidos em um processador ocioso Intel Core i7 – 4500 1.8Ghz, com 8 GB de RAM e executando sistema operacional Fedora 24 64-bits.

Os benchmarks na Tabela 5.1 são uma suíte composta de tarefas de verificação extraídas da SV-Comp 2015 [52] e dois benchmarks desenvolvidos baseados em primitivas de sincronização de programas concorrentes. *account_bad.c* é um programa que representa as operações básicas em contas bancárias: depósito, saque e saldo atual, com um semáforo para controlá-las. *arithmetic_prog_bad.c* é um programa básico produtor-consumidor, usando semáforos e variáveis condicionais para sincronizar operações. *carter_bad.c* é um programa extraído de uma aplicação de banco de dados, o qual usa um semáforo para sincronizar as *threads*. *circular_buffer_bad.c* simula um buffer, usando variáveis compartilhadas para sincronizar as operações de recebimento e envio. *lazy*01*_bad.c* usa um semáforo para controlar operações de soma sobre uma variável compartilhada e então verificar o valor dela. *queue_bad.c* é um programa que simula uma estrutura de dados de fila. *sync*01*_bad.c* e *sync*02*_bad.c* são programas produtores e consumidores: o primeiro nunca consome os dados e o último inicializa uma variável compartilha com algum dado (arbritário). *token_ring_bad.c* propaga valores por meio de variáveis compartilhadas e verifica se elas são equivalentes, por meio de *threads* diferentes. *twostage_bad.c* simula um grande número de *threads* executando simultaneamente e *wronglock_bad.c* simula um extenso número de *threads* produtoras e a propagação dos seus respectivos valores para outras *threads*. $race-1\_1-join\_true-unreach-call.c$ é um programa que testa condições de corrida em uma variável compartilhada usando um semáforo para sincronizar o acesso à mesma. $bigshot\_p\_false-unreach-call.c$ é um programa que aloca e copia dados para um ponteiro de *string*, em duas *threads* diferentes, e por último verifica se o ponteiro foi manipulado como esperado. $fib\_bench\_false-unreach-call.c$ e $fib\_bench\_longer\_false-unreach-call.c$ são programas que iteram um número de vezes pré-determinado e incrementam duas variáveis compartilhadas, sem um semáforo para sincronizar acessos, e então checam se alguma delas atingiu um valor específico ou não. $lazy01\_false-unreach-call.c$ usa um semáforo para controlar operações de soma sobre uma variável compartilhada e então verifica o seu valor. $stateful01\_false-unreach-call.c$ usa duas variáveis compartilhadas guardadas por dois diferentes semáforos, realiza operações aritméticas nas mesmas e então verifica seus valores finais. $qw2004\_false-unreach-call.c$ testa operações aritméticas não guardadas sobre variáveis compartilhadas. Finalmente, *half_sync.c* e *no_sync.c* compartilham um contador para incrementar seu valor e verificar uma propriedade sem uma sincronia completa de *threads* e foram desenvolvidos para posteriormente comporem uma suíte de benchmarks para avaliação de métodos de localização de falhas.

O procedimento de avaliação experimental pode ser dividido em três passos diferentes. Primeiro, é necessário obter um contraexemplo para o dado programa. Caso o resultado dado pelo ESBMC for *verification failed*, então o benchmark não é seguro, *i.e.*, ele apresenta um erro de bloqueio fatal, assertiva, aquisição de semáforo, divisão por zero, segurança de ponteiros, estouro aritmético, e/ou violação de limites de vetores, e pode-se traduzir este compartamento defeituoso em um programa sequencial que simula tal execução. No segundo passo, é necessário extrair as informações de troca de contexto , *i.e.*, número de trocas de contexto (instruções onde trocas de contexto ocorreram) e o número de *threads* em execução, por meio do método apresentado no Capítulo 4, o qual é obtido por meio da remoção da opção `-deadlock-check` na linha de comando do ESBMC em questão[1] (no caso do Lazy-CSeq, removendo a opção `-deadlock` do comando line[2] irá produzir o resultado esperado), e então realizar o processamento do contraexemplo gerado. No terceiro passo, o programa original é transformado em um sequencial, com as informações obtidas nos passos 1 e 2, aplicando as regras definidas no Capítulo 4 e o método proposto por Griesmayer *et al.* [16]. Finalmente, a versão sequencial pode ser verificada no ESBMC, usando a linha de comando[2] sem a opção `-deadlock-check`, mudando o arquivo especificado e aplicando a mesma estratégia demonstrada no Capítulo 4.

5.3 Resultados Experimentais

A Tabela 5.1 resume os resultados experimentais. **F** descreve o nome do benchmark, **L** representa o número de linhas no código em questão, **T** é o número de *threads* no código, **D** identifica se um bloqueio fatal ocorreu (caso tal valor seja `1`), **FE** é o número de erros encontrados durante o processo de localização de falhas, isto é, o número de diferentes valores para `diag` retornados pelo ESBMC, **AE** é o número falhas verdadeiras, **R** representa o resultado final (`1` se a informação obtida pelo ESBMC é de fato útil), e, finalmente, **VT** é o tempo que o ESBMC levou para verificar o benchmark em questão. O ponto de interrogação é usado para identificar testes dos quais nenhuma informação foi obtida, devido a limitações do sistema.

A verificação do arquivo *account_bad.c* apresentou 3 diferentes valores para `diag`, os quais estão em diferentes partes do código; no entanto, eles, por fim, identificaram a falha real existente no código original: uma assertiva mal-formulada.

[1] `esbmc -no-bounds-check -no-pointer-check -no-div-by-zero-check -no-slice -deadlock-check -boolector <file>`

[2] `cseq -deadlock -cex -rounds 5 -i <file>`

Tabela 5.1: Resultados do experimento

F	L	T	D	FE/AE	VT	R
account_bad.c	49	2	0	3/3	0.102	1
arithmetic_prog_bad.c	82	2	1	2/2	0.130	1
carter_bad.c	43	4	1	1/1	0.289	1
circular_buffer_bad.c	109	2	0	7/7	0.227	1
queue_bad.c	153	2	0	4/4	0.934	1
sync01_bad.c	64	2	1	1/0	0.451	0
sync02_bad.c	39	2	1	2/2	0.116	1
token_ring_bad.c	56	4	0	1/1	0.307	1
twostage_bad.c	128	9	0	1/1	0.284	1
wronglock_bad.c	111	7	0	1/1	0.310	1
race-1_1-join_true-unreach-call.c	58	1	0	1/1	0.399	1
bigshot_p_false-unreach-call.c	35	2	0	2/2	10.724	1
fib_bench_false-unreach-call.c	44	2	0	2/2	8.966	1
fib_bench_longer_false-unreach-call.c	44	2	0	2/2	11.147	1
lazy01_false-unreach-call.c	51	3	0	1/1	0.259	1
stateful01_false-unreach-call.c	56	2	0	2/2	0.264	1
qw2004_false-unreach-call.c	60	1	0	1/1	0.250	1
half_sync.c	22	2	0	1/0	0.322	0
no_sync.c	21	2	0	1/0	0.320	0

Os 7 valores diagnosticados em relação ao *circular_buffer_bad.c* levam à uma assertiva errônea no programa, que está relacionada a um laço. Dessa forma, os valores de `diag` indicam tal laço.

Durante a verificação do *arithmetic_prog_bad.c*, a metodologia proposta informou 2 valores diferentes para `diag`, os quais direcionam a um laço na *thread* 2 deste programa, significando que a falha está neste laço específico.

A análise de *queue_bad.c* apresentou 4 erros. As falhas identificadas são relacionadas a *flags* que provêm controle de acesso a uma variável compartilhada e um loop onde elas são modificadas, isto é, o problema reside novamente em má operação.

sync02_bad.c apresentou 2 valores diferentes de diagnóstico, relacionados à *thread* consumidora no programa original, cujas linhas estão relacionadas a um bloqueio fatal presente no benchmark.

carter_bad.c, *token_ring_bad.c*, *twostage_bad.c* e *wronglock_bad.c* apresentaram 1 linha defeituosa durante o diagnóstico de cada programa sequencializado. No primeiro, o método proposto foi capaz de encontrar uma atribuição a uma variável compartilhada que corrige o bloqueio fatal presente neste programa. No segundo, o valor de diagnóstico leva a expressão lógica

mal-formulada na assertiva. O terceiro foi diagnosticado com uma linha defeituosa que aponta para uma assertiva mal-formulada e, finalmente, o quarto benchmark referido contém uma assertiva mal-formulada, a qual a metodologica foi capaz de apontar.

A metodologia foi capaz de informar 1 linha defeituosa diferente para *race-1_1-join_true-unreach-call.c*, o qual leva a uma assertiva mal-formulada no programa, e mudando o valor de comparação produziria uma execução bem-sucedida do programa.

Os benchmarks *bigshot_p_false-unreach-call.c*, *fib_bench_false-unreach-call.c* e *fib_bench_longer_false-unreach-call.c* foram diagnosticados com 2 erros. No primeiro, as linhas defeituosas obtidas se referem a uma assertiva, significando que a expressão lógica não é verdadeira em uma determinada intercalação. Os outros dois benchmarks são essencialmente os mesmos, diferindo apenas em um valor avaliado na assertiva. As linhas defeituosas associadas também estão relacionadas a tais valores, visto que os valores são usados nas comparações mencionadas podem ser diferentes dos esperados.

lazy01_false-unreach-call.c e *qw2004_false-unreach-call.c* apresentaram 1 erro cada. Em ambos, esses erros estão relacionados a assertivas, significando que as expressões associadas deveriam ser avaliadas com valores diferentes.

stateful01_false-unreach-call.c foi diagnosticado com 2 falhas que levam a expressões em assertivas. Logo, para que seja possível produzir uma execução bem-sucedida do programa, deve-se alterar os valores utilizados.

Embora *sync01_bad.c* não tenha apresentado erros, ele foi diagnosticado com uma falha. De fato o ESBMC encontrou um `diag` com valor 0, o que é particularmente estranho, visto que não há uma linha 0. Além disso, mesmo após adicionar uma assertiva, o ESBMC ainda retorna o valor 0. De fato, o programa apresenta problemas de sincronização, mas a metodologica proposta não foi capaz de prover informações úteis. Ainda mais, esse benchmark apresenta um bloqueio fatal devido a suas condicionais, característica que a metodologia não modela atualmente. Para considerar tal programa, seria necessária uma melhoria na gramática de sequencialização, de forma a modelar o comportamento de bloqueios fatais presentes em programas como este.

Os casos principais que a metodologia proposta falhou foram *half_sync.c* e *no_sync.c*. Mesmo que o ESBMC tenha provido valores de `diag` razoáveis, eles não levam às falhas de verdade, visto que aplicando os consertos propostos não corrige o programa de entrada e tal procedimento ao invés disso leva a diferentes caminhos defeituosos. As falhas nesses programas

estão relacionadas a sincronização de *threads*, *i.e.*, a propriedade é validada pelo verificador de modelos antes que outras *threads* terminem sua execução. Ainda mais, a falta de uma chamada apropriada do método `pthread_join` leva a essas falhas de sincronização, as quais não são suportadas pela abordagem proposta.

De acordo com os resultados apresentados na Tabela 5.1, pode-se notar que a metodologia proposta foi capaz de encontrar falhas (informações úteis) em 16 de 19 benchmarks, o que representa um total de 84.21%. É possível notar que os benchmarks os quais a verificação falhou e, consequentemente, para os quais nenhum contraexemplo foi obtido também estão incluídos nessa avaliação. A metodologia em si se mostrou ser útil para diagnosticar violações de corrida de dados, visto que a maioria dos benchmarks apresentam uma falha relacionada a esse problema; no entanto, o método proposto precisa ser aprimorado, de modo a verificar bloqueios fatais de uma maneira mais eficiente, transformações de laços também necessitam de um trabalho significativo, para que intercalações de *threads* dentro de laços possam ser melhor representadas, e também um modelo apropriado para espera de *threads* também necessita ser proposto, para que problemas de sincronização sejam completamente cobertos.

Com relação ao tempo de verificação dos programas sequenciais instrumentados, na maioria dos benchmarks esse tempo é menor que 2 segundos. Isso ocorre porque a metodologia proposta provê uma versão minimizada e sequencializada do programa concorrente, e também com menores fontes de não-determinismo, basicamente os valores de `diag`, os quais levam a uma sobrecarga pequena no verificador de modelos. Em 3 benchmarks, no entanto, obtiveram-se tempos de verificação significantemente maiores, 5 a 6 mais que o normal. A execução do verificador de modelos foi analizada e, em tais casos, o solucionador SMT levou mais tempo que o esperado para avaliar a fórmula SMT do programa, possivelmente devido ao não-determinismo em `diag`.

Com relação aos benchmarks em que nenhuma informação útil foi obtida, isso leva à conclusão de que gramática e regras aprimoradas são necessárias, de modo a localizar falhas. Além disso, os resultados experimentais mostraram a viabilidade da metodologia proposta para localizar falhas em programas concorrentes em C, visto que o ESBMC e o Lazy-CSeq são capazes de prover informações de diagnóstico úteis relacionadas a potenciais falhas.

É importante mencionar que é possível aplicar a metodologia em programas onde o verificador de modelos é capaz de obter um contraexemplo para uma propriedade de segurança violada. Este contraexemplo é imprescindível para a etapa de sequencialização do programa,

visto que ela depende inteiramente da intercalação presente no contraexemplo. Desta forma, cabe ao verificador escolhido explorar as intercalações possíveis para encontrar tal violação. No entanto, existem pontos em que o método pode ser melhorado, que serão explicitados posteriormente (cf. Capítulo 6).

5.3.1 Escalonabilidade

A metodologia proposta depende inteiramente de técnicas de verificação de modelos e uma execução mal-sucedida do programa, *i.e.*, um contraexemplo proveniente de um verificador de modelos, para localizar falhas em programas concorrentes. Logo, pode-se afirmar que a metodologia é escalável caso o verificador de modelos usados para aplicá-la é escalável e pode prover corretamente um contraexemplo para o programa em verificação.

Atualmente, verificadores de modelos são escaláveis para lidar com programas concorrentes grandes [53]. A princípio, verificadores de modelos tentavam modelos programas concorrentes em seus estados naturais, modelando instruções e execuções concorrentes, mas essa abordagem geralmente se deparava com o problema da explosão de estados. Então, pesquisadores propuseram diferentes abordagens, de forma a escalar a verificação de modelos com relação a programas concorrentes. Como exemplo, o Lazy-CSeq [54] é capaz de encontrar violações de suítes de teste extensas, tais como a suíte de concorrência da *SV-Comp*, que contém erros de diferentes naturezas, e também já foi premiado como um dos verificadores de modelos mais eficientes para tratar concorrência em edições da *SV-Comp*.

5.3.2 Aplicando a Abordagem Proposta na Prática

Para avaliar a dificuldade da aplicação da metodologia proposta, conduziu-se um experimento simples com 6 desenvolvedores que trabalham no seu dia-a-dia com software concorrente embarcado, com diferentes níveis de experiência. Pediu-se para que eles depurassem o benchmark *account_bad.c* da forma que eles fariam normalmente tal tarefa. Então, explicaram-se os passos da abordagem proposta e pediu-se para que eles refizessem a tarefa usando a metodologia e anotou-se o tempo utilizado por cada um. Vale a pena apontar que os voluntários não tinham conhecimento prévio sobre a metodologia e cada um foi indepentemente contactado e avaliado.

Um resumo dos resultados obtidos está disponível na Tabela 5.2, onde **Descrição do cargo** representa o nível de experiência de cada desenvolvedor, **Tempo de depuração não-assistida** representa o tempo, em minutos, que cada desenvolvedor levou para encontrar um erro no programa, e, finalmente, **Tempo de depuração assistida** representa o tempo, em minutos, que cada desenvolvedor levou para encontrar um erro usando a metodologia proposta.

Tabela 5.2: Medindo os benefícios da metodologia proposta na prática.

Descrição do cargo	Tempo de depuração não-assistida	Tempo de depuração assistida
Desenvolvedor Sr A	13.20	2.50
Desenvolvedor Sr B	14.00	2.75
Desenvolvedor Pl A	19.50	3.20
Desenvolvedor Pl B	20.00	3.00
Desenvolvedor Jr A	38.30	5.10
Desenvolvedor Jr B	41.20	4.25

Embora o grupo de voluntários não tenha sido grande e não foi levada em consideração a familiaridade prévia com o erro proposto, assim como sua experiência prática em depuração, uma tendência parece existir em relação à redução do tempo utilizado associado para identificar o erro existente no programa escolhido. A principal causa é que a intuição e a experiência são substituídas pela metodologia, a qual consiste em prover uma execução mal-sucedida para o programa dado e então aplicar o arcabouço de sequencialização, de forma a detectar linhas que precisam de reparos. Como consequência, a tarefa de localização de falhas tornou-se um processo sistemático composto pelos seguintes passos: obtenção de um contraexemplo, criação de um código sequencial instrumentado não-determinístico equivalente e a extração de linhas defeituosas do novo contraexemplo. No entanto, para que se obtenham resultados expressivos, um experimento com um programa concorrente relevante (*e.g.*, um produto de mercado de uma empresa) deve ser realizado, avaliando o efeito da aplicação do método no ciclo de desenvolvimento e a eficácia na tarefa de localização. Para tal experimento, o método deve estar disponível de forma automática, de forma que desenvolvedores não necessitem analisar contraexemplos e gerar códigos sequenciais para detecção de falhas.

5.3.3 Considerações Finais

Os benchmarks aplicados basicamente apresentam um tipo de erro: bloqueios fatais, assertivas, aquisição de semáforos, divisão por zero, segurança de ponteiros, estouro aritmético,

ou violação de limites de vetores. Logo, aplicaram-se diferentes regras, dependendo da falha detectada. A Figura 5.1 mostra um resumo de todos os resultados obtidos, o qual mostra que a metodologia proposta gera informações úteis sobre execuções mal-sucedidas, com relação ao erro presente no benchmark, em 84.21% dos benchmarks utilizados, além de identificar quais linhas estão relacionadas à falha associada e quais valores podem ser atribuídos para produzir uma execução bem-sucedida. Com relação aos benchmarks utilizados, a abordagem não foi capaz de diagnosticar linhas defeituosas em três deles, devido à modelagem da biblioteca concorrente do C, visto que tais benchmarks apresentaram problemas de bloqueios fatais e sincronização.

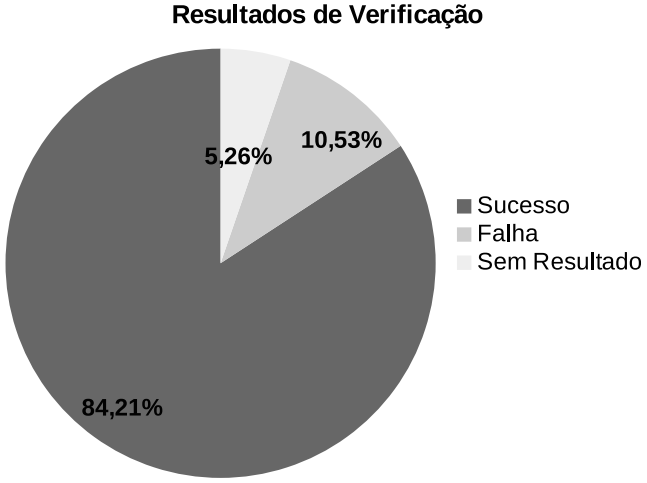

Figura 5.1: Resumo dos resultados de verificação

A Figura 5.2 mostra um resumo do tempo de verificação necessário para cada benchmark. Treze programas levaram menos de 2 segundos, de forma a avaliar as falhas existentes e as atribuições associadas para produzir uma execução bem-sucedida, três programas não foram verificados corretamente, com tempos associados de verificação menores que 2 segundos, e, finalmente, três benchmarks levaram cerca de 4 a 5 vezes mais para produzir resultados úteis. Com relação ao último caso, analizando o processo de verificação do verificador de modelos, foi possível apontar o gargalo como sendo o solucionador SMT, devido à quantidade de fontes de não-determinismo presentes em tais benchmarks. No entanto, o tempo de verificação em média ainda é baixo, se comparado com o que pode ser obtido com depuração de programas não-assistida [3].

Em resumo, os resultados apresentados mostram que a metodologia proposta é apropriada para localizar falhas em programas concorrentes em C, e que o tempo de verificação

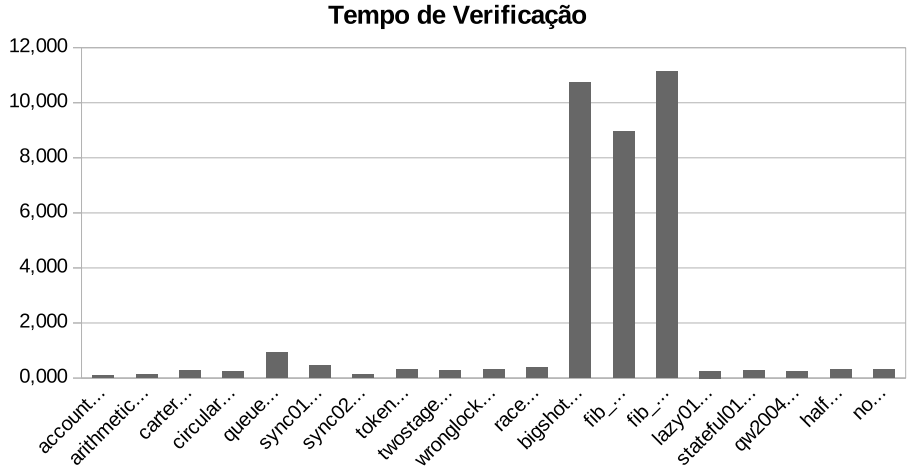

Figura 5.2: Resumo dos resultados de todos os benchmarks

necessário para produzir uma execução bem-sucedida do programa é curto. Logo, a metodologia proposta pode ser útil para auxiliar desenvolvedores a depurar programas.

5.4 Resumo

Neste capítulo apresentou a avaliação experimental realizada para o método proposto neste trabalho, assim como considerações a serem feitas sobre a metodologia apresentada. O experimento foi conduzido com um computador padrão e os dados obtidos foram analizados, mostrando a viabilidade do método para encontrar linhas defeituosas em códigos concorrentes, tendo uma taxa de sucesso de 84.21%, considerando todos os benchmarks propostos. A metodologia proposta mostrou-se útil para determinar as linhas defeituosas em um programa concorrente. No entanto, a dependência de um contraexemplo para iniciar o processo de transformação de código levou a alguns benchmarks não serem avaliados, pois o ESBMC não foi capaz de retornar um contraexemplo para os mesmos. Também é importante notar que após a transformação de código ser feita, o tempo de verificação do código instrumentado foi sempre menor que 1 segundo. De modo geral, a metodologia garante que a correção no código original das linhas obtidas nos contraexemplos do novo código sequencial levam a uma execução bem-sucedida do programa.

Capítulo 6

Conclusões

6.1 Considerações Finais

Nesta dissertação foi apresentado um método para localizar falhas em programas concorrentes em C, usando regras de sequencialização e técnicas de verificação de modelos limitada. Este trabalho consistiu na transformação linha a linha do código original concorrente com o intuito de simular o mesmo comportamento presente no último, e então aplicar o méteodo proposto por Griesmayer *et al.* [16] para obter as linhas que levam a uma violação de uma dada propriedade do programa, representando linhas existentes no programa concorrente.

Com relação aos resultados experimentais, a metodologia proposta foi capaz de identificar potenciais falhas em software concorrente, em 84.21% dos benchmarks escolhidos, enquanto falhou em obter linhas defeituosas úteis em três dos benchmarks adotados. De fato, um deles apresentou um bloqueio fatal e os outros dois falham devido a problemas de sincronização de *threads*, o que necessita de uma investigação mais a fundo, de forma a prover aprimoramentos relacionados às abordagens de modelagem das primitivas de sincronização do C. No entanto, o método é útil para avaliar programas concorrentes que apresentam falhas relacionadas a assertivas mal-formuladas e bloqueios fatais, o que o torna interessante para auxiliar desenvolvedores a encontrar atribuições que levem a execuções bem-sucedidas de programas e, consequentemente, minimiza o esforço em depuração de código.

Ao ser comparado com outras abordagens, o método proposto é capaz de localizar falhas em programas concorrentes usando apenas o código-fonte do programa, ao invés de um caminho mal-sucedido e uma suíte de teste. Ele também é capaz de não somente apontar as linhas

defeituosas, como também prover possíveis consertos para produzir execuções bem-sucedidas do programa.

Durante o desenvolvimento deste trabalho foram identificados pontos que diminuem a eficiência do método proposto. Dentre esses pontos, também observados na avaliação experimental, pode-se destacar a necessidade de um contraexemplo para a posterior aplicação do método, dependendo inteiramente da capacidade do verificador de modelos de encontrar violações no programa original. Este problema está relacionado diretamente à tarefa de custo mais elevado no método proposto, que é a definição do vetor `cs`. Uma definição arbritária, por meio de uso de técnicas de BMC, pode ser estudado em trabalhos posteriores.

Outro ponto a se considerar é a modelagem de estruturas da biblioteca *pthread*. Internamente, o ESBMC implementa um modelo para as primitivas de sincronização da biblioteca, e neste trabalho define-se uma modelagem em um nível anterior à tradução interna do ESBMC. Deve-se também investigar qual das duas modelagens deve ser adotada com o objetivo de melhorar os resultados obtidos.

Apesar dos pontos descritos anteriormente, quando se trata do método proposto para localizar falhas, o tempo de verificação é curto (geralmente menor que 1 segundo) para o programa sequencial instrumentado não-determinístico, possibilitando um diagnóstico rápido para o mesmo.

De maneira geral, pode-se concluir que os objetivos específicos desta dissertação também foram atingidos.

Em uma empresa de desenvolvimento de software é comum o uso de programação concorrente para prover soluções com tempo menor de resposta. Porém, esta classe de programas está sujeita a erros mais difíceis de serem corrigidos, e por consequência, erros que levem mais tempo para serem encontrados. O uso de um método para localizar falhas em tais programas reduziria drasticamente o tempo associado a esta tarefa, melhorando o processo de desenvolvimento de software concorrente. O método proposto nesta dissertação visa ser uma alternativa para depuração comum, usando uma técnica também em ascensão para busca por defeitos, a verificação de modelos.

6.2 Propostas para Trabalhos Futuros

Nesta seção serão apresentadas algumas propostas para desenvolvimentos futuros relacionados ao método proposto descrito nesta dissertação. Estas propostas podem ser dividas em duas categorias. Na primeira, é considerado o problema de sequencialização e modelagem de código, e, na segunda, serão apresentadas propostas que visam melhorar a aplicabilidade do método em geral.

Quanto ao problema de sequencialização e modelagem de código:

- Novas regras para transformação devem ser adicionadas para que seja possível representar melhor o programa original em relação às intercalações entre as *threads* existentes.
- Melhorias na gramática para que seja possível diagnosticar melhor problemas relacionados a declarações relacionadas à biblioteca *pthread*.
- O uso de desdobramento de laços para melhor representação de laços existentes em programas concorrentes, onde também podem existir trocas de contexto.

Quanto ao problema de aplicabilidade do método em geral:

- Um *plugin* deve ser desenvolvido em um ambiente de desenvolvimento, como o *Eclipse*, de forma a automatizar o processo de localização de falhas.
- Propor uma estratégia para simular todas as intercalações possíveis, retirando o processo de definição do escalonamento definido em código do verificador de modelos.
- Incorporar a abordagem proposta por Gadelha *et al.* [55] para minimizar o esforço necessário para encontrar um contraexemplo para um programa concorrente defeituoso.
- Aplicar o método em outros seguimentos de programas, *e.g.*, programas concorrentes em Qt [56].

Bibliografia

[1] BARANIUK, C. *The Number Glitch That Can Lead to Catastrophe*. 2015. [Online; posted 5-May-2015]. Disponível em: <http://goo.gl/qabuJF>.

[2] MYERS, G.; BADGETT, T.; SANDLER, C. *The Art of Software Testing*. 3. ed. : Wiley, 2011.

[3] MAYER, W.; STUMPTNER, M. Evaluating Models for Model-Based Debugging. In: *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. 2008. p. 128–137.

[4] TIP, F. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, p. 121–189, 1995.

[5] OFFUTT, A.; LEE, A.; ROTHERMEL, G.; UNTCH, R.; ZAPF, C. An Experimental Determination of Sufficient Mutant Operators. *ACM Transactions on Software Engineering and Methodology*, n. 2, p. 99–118, 1996.

[6] HE, H.; GUPTA, N. Automated Debugging using Path-Based Weakest Preconditions. In: *Fundamental Approaches to Software Engineering, Springer, LNCS*. 2004. p. 267–280.

[7] CLEVE, H.; ZELLER, A. Locating Causes of Program Failures. In: *Proceedings of the 27th International Conference on Software Engineering*. 2005. p. 342–351.

[8] FRIEDRICH, G.; STUMPTNER, M.; WOTAWA, F. Model-Based Diagnosis of Hardware Designs. *Artificial Intelligence*, p. 3–39, 1996.

[9] CHAKI, S.; GROCE, A.; STRICHMAN, O. Explaining Abstract Counterexamples. In: *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2004. p. 73–82.

[10] ZELLER, A. *Why Programs Fail: A Guide to Systematic Debugging*. 2. ed. : Morgan Kaufmann, 2009.

[11] ROCHA, H.; BARRETO, R. S.; CORDEIRO, L. C.; NETO, A. D. Understanding Programming Bugs in ANSI-C Software Using Bounded Model Checking Counter-Examples. In: *Integrated Formal Methods - 9th International Conference, IFM 2012, Pisa, Italy, June 18-21, 2012. Proceedings*. : Springer, 2012. (Lecture Notes in Computer Science, v. 7321), p. 128–142.

[12] GODEFROID, P.; NAGAPPAN, N. Concurrency at Microsoft: An Exploratory Survey. In: *CAV Workshop on Exploiting Concurrency Efficiently and Correctly*. 2008.

[13] QADEER, S.; WU, D. KISS: Keep It Simple and Sequential. In: *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*. 2004. p. 14–24.

[14] CLARKE, E. M.; VEITH, H. Counterexamples Revisited: Principles, Algorithms, Applications. In: *Verification: Theory and Practice, Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday*. 2003. p. 208–224.

[15] CLARKE, E. M.; GRUMBERG, O.; MCMILLAN, K. L.; ZHAO, X. Efficient Generation of Counterexamples and Witnesses in Symbolic Model Checking. In: *Proceedings of the 32nd Annual ACM/IEEE Design Automation Conference*. 1995. p. 427–432.

[16] GRIESMAYER, A.; STABER, S.; BLOEM, R. Automated Fault Localization for C Programs. *Electronic Notes in Theoretical Computer Science*, p. 95–111, 2007.

[17] MOURA, L.; BJØRNER, N. Satisfiability Modulo Theories: An Appetizer. In: *Formal Methods: Foundations and Applications*. 2009. p. 23–36.

[18] MOURA, L.; BJØRNER, N. Z3: An Efficient SMT Solver. In: *Tools and Algorithms for the Construction and Analysis of Systems*. 2008. p. 337–340.

[19] BRUMMAYER, R.; BIERE, A. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In: *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference*. 2009. p. 174–177.

[20] BJØRNER, N.; MOURA, L. Z310: Applications, Enablers, Challenges and Directions. 2018.

[21] CORDEIRO, L.; FISCHER, B.; MARQUES-SILVA, J. Smt-Based Bounded Model Checking for Embedded ANSI-C Software. *IEEE Transactions on Software Engineering*, p. 957–974, 2012.

[22] MORSE, J.; RAMALHO, M.; CORDEIRO, L. C.; NICOLE, D. A.; FISCHER, B. ESBMC 1.22 - (competition contribution). In: *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*. 2014. (Lecture Notes in Computer Science, v. 8413), p. 405–407.

[23] MORSE, J.; CORDEIRO, L. C.; NICOLE, D. A.; FISCHER, B. Context-bounded model checking of LTL properties for ANSI-C software. In: *Software Engineering and Formal Methods - 9th International Conference, SEFM 2011, Montevideo, Uruguay, November 14-18, 2011. Proceedings*. : Springer, 2011. (Lecture Notes in Computer Science, v. 7041), p. 302–317.

[24] MORSE, J.; CORDEIRO, L. C.; NICOLE, D. A.; FISCHER, B. Model checking LTL properties over ANSI-C programs with bounded traces. *Software and System Modeling*, v. 14, n. 1, p. 65–81, 2015.

[25] MORSE, J.; CORDEIRO, L. C.; NICOLE, D. A.; FISCHER, B. Handling unbounded loops with ESBMC 1.20 - (competition contribution). In: *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. 2013. (Lecture Notes in Computer Science, v. 7795), p. 619–622.

[26] ROCHA, H.; ISMAIL, H.; CORDEIRO, L. C.; BARRETO, R. S. Model checking embedded C software using k-induction and invariants. In: *2015 Brazilian Symposium on Computing Systems Engineering, SBESC 2015, Foz do Iguacu, Brazil, November 3-6, 2015*. : IEEE Computer Society, 2015. p. 90–95.

[27] GADELHA, M. Y. R.; ISMAIL, H. I.; CORDEIRO, L. C. Handling Loops in Bounded Model Checking of C Programs via k-Induction. *Software Tools for Technology Transfer*, p. 97–114, 2017.

[28] ROCHA, W.; ROCHA, H.; ISMAIL, H.; CORDEIRO, L. C.; FISCHER, B. Depthk: A k-induction verifier based on invariant inference for C programs - (competition contribution). In: *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II*. 2017. (Lecture Notes in Computer Science, v. 10206), p. 360–364.

[29] PEREIRA, P. A.; ALBUQUERQUE, H. F.; MARQUES, H.; SILVA, I. da; CARVALHO, C.; CORDEIRO, L. C.; SANTOS, V.; FERREIRA, R. Verifying CUDA programs using smt-based context-bounded model checking. In: *Proceedings of the 31st Annual ACM Symposium on Applied Computing, Pisa, Italy, April 4-8, 2016*. 2016. p. 1648–1653.

[30] PEREIRA, P.; ALBUQUERQUE, H.; SILVA, I. D.; MARQUES, H.; MONTEIRO F. FERREIRA, R.; CORDEIRO, L. SMT-Based Context-Bounded Model Checking for CUDA Programs. *Concurrency and Computation: Practice and Experience*, 2016.

[31] BALL, T.; NAIK, M.; RAJAMANI, S. K. From Symptom to Cause: Localizing Errors in Counterexample Traces. In: *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 2003. p. 97–105.

[32] BALL, T.; RAJAMANI, S. K. Automatically Validating Temporal Safety Properties of Interfaces. In: *Proceedings of the 8th International SPIN Workshop on Model Checking of Software*. 2001. p. 103–122.

[33] GROCE, A.; VISSER, W. What Went Wrong: Explaining Counterexamples. In: *Proceedings of the 10th International Conference on Model Checking Software*. 2003. p. 121–136.

[34] JAVA Pathfinder: Framework for Verification of Java Programs. Disponível em: <http://babelfish.arc.nasa.gov/trac/jpf>.

[35] GROCE, A.; CHAKI, S.; KROENING, D.; STRICHMAN, O. Error Explanation with Distance Metrics. *International Journal on Software Tools for Technology Transfer*, p. 229–247, 2006.

[36] GRIESMAYER, A. *Dissertation Debugging Software: From Verification to Repair*. 2007.

[37] OHANIAN, H.; MARKERT, J. *Physics for Engineers and Scientists*. 3. ed. : W. W. Norton & Company, 2006.

[38] JONES, J. A.; HARROLD, M. J.; STASKO, J. Visualization of Test Information to Assist Fault Localization. In: *Proceedings of the 24th International Conference on Software Engineering*. 2002. p. 467–477.

[39] ZELLER, A. *AskIgor: Automated Debugging*. 2006. Disponível em: <http://www.st.cs.uni-saarland.de/askigor/>.

[40] BIRCH, G.; FISCHER, B.; POPPLETON, M. R. Fast Model-Based Fault Localisation with Test Suites. In: *9th International Conference Tests and Proofs*. 2015. p. 38–57.

[41] CADAR, C.; DUNBAR, D.; ENGLER, D. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. 2008. p. 209–224.

[42] TOMASCO, E.; INVERSO, O.; FISCHER, B.; La Torre, S.; PARLATO, G. Verifying Concurrent Programs by Memory Unwinding. In: *21st International Conference Tools and Algorithms for the Construction and Analysis of Systems*. 2015. p. 551–565.

[43] CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. *Introduction to Algorithms*. 3. ed. : The MIT Press, 2009.

[44] BUTENHOF, D. R. *Programming with POSIX Threads*. : Addison-Wesley Longman Publishing Co., Inc., 1997.

[45] CORDEIRO, L.; FISCHER, B. Verifying Multi-Threaded Software Using SMT-Based Context-Bounded Model Checking. In: *Proceedings of the 33rd International Conference on Software Engineering*. 2011. p. 331–340.

[46] PARK, S. M. *Effective Fault Localization Techniques for Concurrent Software*. 2014.

[47] JOSE, M.; MAJUMDAR, R. Cause Clue Clauses: Error Localization Using Maximum Satisfiability. In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2011. p. 437–446.

[48] B., T. A.; DEGELO, R. F.; JUNIOR, E. G. D. S.; ISMAIL, H. I.; SILVA, H. C. D.; CORDEIRO, L. C. Multi-Core Model Checking and Maximum Satisfiability Applied to Hardware-Software Partitioning. *Journal of Embedded Systems*, 2016.

[49] HONG, S.; LEE, B.; KWAK, T.; JEON, Y.; KO, B.; KIM, Y.; KIM, M. Mutation-Based Fault Localization for Real-World Multilingual Programs. In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2015. p. 464–475.

[50] PAPADAKIS, M.; TRAON, Y. L. Metallaxis-FL: Mutation-Based Fault Localization. *Software Testing, Verification and Reliability*, p. 605–628, 2015.

[51] CLARKE, E.; KROENING, D.; LERDA, F. A Tool for Checking ANSI-C Programs. In: *Tools and Algorithms for the Construction and Analysis of Systems*. : Springer, 2004. p. 168–176.

[52] BEYER, D. Software Verification and Verifiable Witnesses. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 2015. p. 401–416.

[53] BEYER, D. Reliable and Reproducible Competition Results with BenchExec and Witnesses (Report on SV-COMP 2016). In: ____. *Tools and Algorithms for the Construction and Analysis of Systems: 22nd International Conference, TACAS 2016*. 2016. p. 887–904.

[54] INVERSO, O.; TOMASCO, E.; FISCHER, B.; TORRE, S. L.; PARLATO, G. Bounded Model Checking of Multi-Threaded C Programs via Lazy Sequentialization. In: *Proceedings of the 16th International Conference on Computer Aided Verification*. 2014. p. 585–602.

[55] RAMALHO, M.; CORDEIRO, L.; NICOLE, D. A. Counterexample-Guided k-Induction Verification for Fast Bug Detection. 2017.

[56] MONTEIRO, F. R.; GARCIA, M. A. P.; CORDEIRO, L. C.; FILHO, E. B. D. L. Bounded Model Checking of C++ Programs Based on the Qt Cross-Platform Framework. *Software Testing, Verification and Reliability*, 2017.

[57] ALVES, E.; CORDEIRO, L.; FILHO, E. L. Fault localization in multi-threaded C programs using bounded model checking. In: *2015 Brazilian Symposium on Computing Systems Engineering, SBESC 2015, Foz do Iguacu, Brazil, November 3-6, 2015*. : IEEE Computer Society, 2015. p. 96–101.

[58] ALVES, E.; CORDEIRO, L.; FILHO, E. L. A Method to Localize Faults in Concurrent C Programs. *Journal of Systems and Software*, v. 132, p. 336–352, 2017. Disponível em: <https://doi.org/10.1016/j.jss.2017.03.010>.

[59] MONTEIRO, F. R.; ALVES, E. H. da S.; SILVA, I. da; ISMAIL, H.; CORDEIRO, L. C.; FILHO, E. B. de L. ESBMC-GPU A context-bounded model checking tool to verify CUDA programs. *Sci. Comput. Program.*, v. 152, p. 63–69, 2018.

Apêndice A

Publicações

A.1 Referente à Pesquisa

- **Alves, E. H. S.**, Cordeiro, L. C., Lima Filho, E. B. *Fault Localization in Multi-Threaded C Programs using Bounded Model Checking*. Em V Simpósio Brasileiro de Engenharia de Sistemas Computacionais (SBESC), 2015. (**Publicado**) [57]

- **Alves, E. H. S.**, Cordeiro, L. C., Lima Filho, E. B. *A Method to Localize Faults in Concurrent C Programs*. Em *Journal of Software and Systems*, 2017. (**Publicado**) [58]

A.2 Contribuições em outras Pesquisas

- Cavalcante, T. R. F., **Alves, E. H. S.**, Carvalho, C. B. *Radio Communication to Control and Run an Autonomous Mission for UAVs via a Mobile Application*. Em IV Escola Regional de Informática (ERIN), 2017. (**Publicado**)

- Monteiro, F. R. M., **Alves, E. H. S.**, Silva, I. S., Ismail, H. I., Cordeiro, L. C., Lima Filho, E. B. *ESBMC-GPU A Context-Bounded Model Checking Tool to Verify CUDA Programs*. Em *Science of Computer Programming*, 2018. (**Publicado**) [59]

Apêndice B

Contraexemplo para o código da Figura 2.1

```
1 ESBMC version 2.1.0 64-bit x86_64 linux
2 file model.c: Parsing
3 Converting
4 Type-checking model
5 Generating GOTO Program
6 GOTO program creation time: 0.246s
7 GOTO program processing time: 0.004s
8 Starting Bounded Model Checking
9 Symex completed in: 0.002s
10 size of program expression: 26 assignments
11 Slicing time: 0.000s
12 Generated 1 VCC(s), 1 remaining after simplification
13 Encoding remaining VCC(s) using bit-vector arithmetic
14 Encoding to solver time: 0.000s
15 Solving with solver Boolector
16 Runtime decision procedure: 0.001s
17 Building error trace
18
19 Counterexample:
20
21 State 1 file model.c line 14 function main thread 0
22 <main invocation>
23 ------------------------------------------------------------
```

```
24 Violated property:
25   file model.c line 14 function main
26   assertion
27   FALSE
28
29 VERIFICATION FAILED
```

Apêndice C

Contraexemplo para o código da Figura 2.3

```
1 ESBMC version 2.1.0 64-bit x86_64 linux
2 file griesmayer.c: Parsing
3 Converting
4 Type-checking griesmayer
5 Generating GOTO Program
6 GOTO program creation time: 0.277s
7 GOTO program processing time: 0.005s
8 Starting Bounded Model Checking
9 Symex completed in: 0.010s
10 size of program expression: 126 assignments
11 Slicing time: 0.004s
12 Generated 1 VCC(s), 1 remaining after simplification
13 Encoding remaining VCC(s) using bit-vector arithmetic
14 Encoding to solver time: 0.002s
15 Solving with solver Boolector
16 Runtime decision procedure: 0.051s
17 Building error trace
18
19 Counterexample:
20
21 State 2 file griesmayer.c line 12 function nondet thread 0
22 c::nondet at griesmayer.c line 16
23 c::controller at griesmayer.c line 37
```

```
<main invocation>
------------------------------------------------------
  c::controller::$tmp::return_value_nondet$1=-2012462479 (-2012462479)

State 3 file griesmayer.c line 16 function controller thread 0
c::controller at griesmayer.c line 37
<main invocation>
------------------------------------------------------
  griesmayer::controller::1::diag=-2012462479 (-2012462479)

State 4  thread 0
c::controller at griesmayer.c line 37
<main invocation>
------------------------------------------------------
  c::controller::$tmp::tmp$2=FALSE

State 5  thread 0
c::controller at griesmayer.c line 37
<main invocation>
------------------------------------------------------
  c::controller::$tmp::tmp$4=FALSE

State 6  thread 0
c::controller at griesmayer.c line 37
<main invocation>
------------------------------------------------------
  c::controller::$tmp::tmp$6=FALSE

State 7 file griesmayer.c line 19 function controller thread 0
c::controller at griesmayer.c line 37
<main invocation>
------------------------------------------------------
  griesmayer::controller::1::tc=2 (2)

State 8 file griesmayer.c line 20 function controller thread 0
c::controller at griesmayer.c line 37
<main invocation>
------------------------------------------------------
```

```
62   griesmayer::controller::1::output=2 (2)
63
64 State 9 file griesmayer.c line 21 function controller thread 0
65 c::controller at griesmayer.c line 37
66 <main invocation>
67 ------------------------------------------------------------
68   c::main::$tmp::return_value_controller$1=2 (2)
69
70 State 10  thread 0
71 <main invocation>
72 ------------------------------------------------------------
73   c::main::$tmp::tmp$2=TRUE
74
75 State 12 file griesmayer.c line 12 function nondet thread 0
76 c::nondet at griesmayer.c line 16
77 c::controller at griesmayer.c line 37
78 <main invocation>
79 ------------------------------------------------------------
80   c::controller::$tmp::return_value_nondet$1=2 (2)
81
82 State 13 file griesmayer.c line 16 function controller thread 0
83 c::controller at griesmayer.c line 37
84 <main invocation>
85 ------------------------------------------------------------
86   griesmayer::controller::1::diag=2 (2)
87
88 State 14  thread 0
89 c::controller at griesmayer.c line 37
90 <main invocation>
91 ------------------------------------------------------------
92   c::controller::$tmp::tmp$2=FALSE
93
94 State 15 file griesmayer.c line 17 function controller thread 0
95 c::controller at griesmayer.c line 37
96 <main invocation>
97 ------------------------------------------------------------
98   griesmayer::controller::1::ta=1 (1)
99
```

```
100 State 16  thread 0
101 c::controller at griesmayer.c line 37
102 <main invocation>
103 ------------------------------------------------------------
104   c::controller::$tmp::tmp$4=TRUE
105
106 State 18 file griesmayer.c line 12 function nondet thread 0
107 c::nondet at griesmayer.c line 18
108 c::controller at griesmayer.c line 37
109 <main invocation>
110 ------------------------------------------------------------
111   c::controller::$tmp::return_value_nondet$5=-3 (-3)
112
113 State 19 file griesmayer.c line 18 function controller thread 0
114 c::controller at griesmayer.c line 37
115 <main invocation>
116 ------------------------------------------------------------
117   griesmayer::controller::1::tb=-3 (-3)
118
119 State 20  thread 0
120 c::controller at griesmayer.c line 37
121 <main invocation>
122 ------------------------------------------------------------
123   c::controller::$tmp::tmp$6=FALSE
124
125 State 21 file griesmayer.c line 19 function controller thread 0
126 c::controller at griesmayer.c line 37
127 <main invocation>
128 ------------------------------------------------------------
129   griesmayer::controller::1::tc=2 (2)
130
131 State 22 file griesmayer.c line 20 function controller thread 0
132 c::controller at griesmayer.c line 37
133 <main invocation>
134 ------------------------------------------------------------
135   griesmayer::controller::1::output=0 (0)
136
137 State 23 file griesmayer.c line 21 function controller thread 0
```

```
138 c::controller at griesmayer.c line 37
139 <main invocation>
140 ------------------------------------------------------
141   c::main::$tmp::return_value_controller$3=0 (0)
142
143 State 24  thread 0
144 <main invocation>
145 ------------------------------------------------------
146   c::main::$tmp::tmp$4=TRUE
147
148 State 26 file griesmayer.c line 12 function nondet thread 0
149 c::nondet at griesmayer.c line 16
150 c::controller at griesmayer.c line 37
151 <main invocation>
152 ------------------------------------------------------
153   c::controller::$tmp::return_value_nondet$1=2 (2)
154
155 State 27 file griesmayer.c line 16 function controller thread 0
156 c::controller at griesmayer.c line 37
157 <main invocation>
158 ------------------------------------------------------
159   griesmayer::controller::1::diag=2 (2)
160
161 State 28  thread 0
162 c::controller at griesmayer.c line 37
163 <main invocation>
164 ------------------------------------------------------
165   c::controller::$tmp::tmp$2=FALSE
166
167 State 29 file griesmayer.c line 17 function controller thread 0
168 c::controller at griesmayer.c line 37
169 <main invocation>
170 ------------------------------------------------------
171   griesmayer::controller::1::ta=4 (4)
172
173 State 30  thread 0
174 c::controller at griesmayer.c line 37
175 <main invocation>
```

```
176 ----------------------------------------------------
177   c::controller::$tmp::tmp$4=TRUE
178
179 State 32 file griesmayer.c line 12 function nondet thread 0
180 c::nondet at griesmayer.c line 18
181 c::controller at griesmayer.c line 37
182 <main invocation>
183 ----------------------------------------------------
184   c::controller::$tmp::return_value_nondet$5=-3 (-3)
185
186 State 33 file griesmayer.c line 18 function controller thread 0
187 c::controller at griesmayer.c line 37
188 <main invocation>
189 ----------------------------------------------------
190   griesmayer::controller::1::tb=-6 (-6)
191
192 State 34  thread 0
193 c::controller at griesmayer.c line 37
194 <main invocation>
195 ----------------------------------------------------
196   c::controller::$tmp::tmp$6=FALSE
197
198 State 35 file griesmayer.c line 19 function controller thread 0
199 c::controller at griesmayer.c line 37
200 <main invocation>
201 ----------------------------------------------------
202   griesmayer::controller::1::tc=2 (2)
203
204 State 36 file griesmayer.c line 20 function controller thread 0
205 c::controller at griesmayer.c line 37
206 <main invocation>
207 ----------------------------------------------------
208   griesmayer::controller::1::output=0 (0)
209
210 State 37 file griesmayer.c line 21 function controller thread 0
211 c::controller at griesmayer.c line 37
212 <main invocation>
213 ----------------------------------------------------
```

```
214   c::main::$tmp::return_value_controller$5=0 (0)
215
216 State 38  thread 0
217 <main invocation>
218 ----------------------------------------------------------
219   c::main::$tmp::tmp$6=TRUE
220
221 State 40 file griesmayer.c line 12 function nondet thread 0
222 c::nondet at griesmayer.c line 16
223 c::controller at griesmayer.c line 37
224 <main invocation>
225 ----------------------------------------------------------
226   c::controller::$tmp::return_value_nondet$1=2 (2)
227
228 State 41 file griesmayer.c line 16 function controller thread 0
229 c::controller at griesmayer.c line 37
230 <main invocation>
231 ----------------------------------------------------------
232   griesmayer::controller::1::diag=2 (2)
233
234 State 42  thread 0
235 c::controller at griesmayer.c line 37
236 <main invocation>
237 ----------------------------------------------------------
238   c::controller::$tmp::tmp$2=FALSE
239
240 State 43 file griesmayer.c line 17 function controller thread 0
241 c::controller at griesmayer.c line 37
242 <main invocation>
243 ----------------------------------------------------------
244   griesmayer::controller::1::ta=9 (9)
245
246 State 44  thread 0
247 c::controller at griesmayer.c line 37
248 <main invocation>
249 ----------------------------------------------------------
250   c::controller::$tmp::tmp$4=TRUE
251
```

```
252 State 46 file griesmayer.c line 12 function nondet thread 0
253 c::nondet at griesmayer.c line 18
254 c::controller at griesmayer.c line 37
255 <main invocation>
256 ------------------------------------------------------
257   c::controller::$tmp::return_value_nondet$5=-3 (-3)
258
259 State 47 file griesmayer.c line 18 function controller thread 0
260 c::controller at griesmayer.c line 37
261 <main invocation>
262 ------------------------------------------------------
263   griesmayer::controller::1::tb=-9 (-9)
264
265 State 48  thread 0
266 c::controller at griesmayer.c line 37
267 <main invocation>
268 ------------------------------------------------------
269   c::controller::$tmp::tmp$6=FALSE
270
271 State 49 file griesmayer.c line 19 function controller thread 0
272 c::controller at griesmayer.c line 37
273 <main invocation>
274 ------------------------------------------------------
275   griesmayer::controller::1::tc=2 (2)
276
277 State 50 file griesmayer.c line 20 function controller thread 0
278 c::controller at griesmayer.c line 37
279 <main invocation>
280 ------------------------------------------------------
281   griesmayer::controller::1::output=2 (2)
282
283 State 51 file griesmayer.c line 21 function controller thread 0
284 c::controller at griesmayer.c line 37
285 <main invocation>
286 ------------------------------------------------------
287   c::main::$tmp::return_value_controller$7=2 (2)
288
289 State 53 file griesmayer.c line 38 function main thread 0
```

```
290 <main invocation>
291 ------------------------------------------------------------
292 Violated property:
293   file griesmayer.c line 38 function main
294   assertion
295   FALSE
296
297 VERIFICATION FAILED
```

Apêndice D

Contraexemplo para o código da Figura 2.5

```
1 ESBMC version 2.1.0 64-bit x86_64 linux
2 file model.c: Parsing
3 Converting
4 Type-checking model
5 Generating GOTO Program
6 GOTO program creation time: 0.235s
7 GOTO program processing time: 0.004s
8 Starting Bounded Model Checking
9 Symex completed in: 0.003s
10 size of program expression: 31 assignments
11 Slicing time: 0.000s
12 Generated 0 VCC(s), 0 remaining after simplification
13 VERIFICATION SUCCESSFUL
14 BMC program time: 0.003s
```

Apêndice E

Contraexemplo simplificado para o código da Figura 4.2

```
1 Counterexample:
2
3 State 1 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 32 thread 0
4
5 State 2 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 27 thread 0
6
7 State 3 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 28 thread 0
8
9 State 4 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 30 thread 0
10
11 State 5 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 31 thread 0
12
13 State 6 file concurrent.c line 6 thread 0
14
15 State 7 file concurrent.c line 5 thread 0
16
17 State 8 file concurrent.c line 6 thread 0
18
```

```
19 State 9 file concurrent.c line 5 thread 0
20
21 State 10 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
       pthread_lib.c line 26 thread 0
22
23 State 11 file <built-in> line 12 thread 0
24
25 State 12 file <built-in> line 13 thread 0
26
27 State 13 file <built-in> line 14 thread 0
28
29 State 14 file <built-in> line 15 thread 0
30
31 State 15 file <built-in> line 57 thread 0
32
33 State 16 file <built-in> line 56 thread 0
34
35 State 17 file <built-in> line 58 thread 0
36
37 State 18 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
       pthread_lib.c line 40 function pthread_start_main_hook thread 0
38 c::pthread_start_main_hook at  line
39
40 State 19 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
       pthread_lib.c line 41 function pthread_start_main_hook thread 0
41 c::pthread_start_main_hook at  line
42
43 State 20 file concurrent.c line 32 function main thread 0
44 <main invocation>
45
46 State 21 file concurrent.c line 33 function main thread 0
47 <main invocation>
48
49 State 26 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
       pthread_lib.c line 86 function pthread_create thread 0
50 c::pthread_create at concurrent.c line 34
51 <main invocation>
52
```

```
53 State 27 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
       pthread_lib.c line 92 function pthread_create thread 0
54 c::pthread_create at concurrent.c line 34
55 <main invocation>
56
57 State 28 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
       pthread_lib.c line 92 function pthread_create thread 0
58 c::pthread_create at concurrent.c line 34
59 <main invocation>
60
61 State 29 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
       pthread_lib.c line 93 function pthread_create thread 0
62 c::pthread_create at concurrent.c line 34
63 <main invocation>
64
65 State 30 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
       pthread_lib.c line 94 function pthread_create thread 0
66 c::pthread_create at concurrent.c line 34
67 <main invocation>
68
69 State 31 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
       pthread_lib.c line 95 function pthread_create thread 0
70 c::pthread_create at concurrent.c line 34
71 <main invocation>
72
73 State 32 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
       pthread_lib.c line 96 function pthread_create thread 0
74 c::pthread_create at concurrent.c line 34
75 <main invocation>
76
77 State 33 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
       pthread_lib.c line 101 function pthread_create thread 0
78 c::pthread_create at concurrent.c line 34
79 <main invocation>
80
81 State 39 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
       pthread_lib.c line 86 function pthread_create thread 0
82 c::pthread_create at concurrent.c line 35
```

```
83 <main invocation>
84
85 State 40 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
       pthread_lib.c line 92 function pthread_create thread 0
86 c::pthread_create at concurrent.c line 35
87 <main invocation>
88
89 State 41 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
       pthread_lib.c line 92 function pthread_create thread 0
90 c::pthread_create at concurrent.c line 35
91 <main invocation>
92
93 State 42 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
       pthread_lib.c line 93 function pthread_create thread 0
94 c::pthread_create at concurrent.c line 35
95 <main invocation>
96
97 State 43 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
       pthread_lib.c line 94 function pthread_create thread 0
98 c::pthread_create at concurrent.c line 35
99 <main invocation>
100
101 State 44 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
       pthread_lib.c line 95 function pthread_create thread 0
102 c::pthread_create at concurrent.c line 35
103 <main invocation>
104
105 State 45 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
       pthread_lib.c line 96 function pthread_create thread 0
106 c::pthread_create at concurrent.c line 35
107 <main invocation>
108
109 State 46 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
       pthread_lib.c line 101 function pthread_create thread 0
110 c::pthread_create at concurrent.c line 35
111 <main invocation>
112
```

```
113 State 50 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 135 function pthread_join_switch thread 0
114 c::pthread_join_switch at concurrent.c line 36
115 <main invocation>
116
117 State 51 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 137 function pthread_join_switch thread 0
118 c::pthread_join_switch at concurrent.c line 36
119 <main invocation>
120
121 State 57 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 236 function pthread_mutex_lock_check thread 1
122 c::pthread_mutex_lock_check at concurrent.c line 9
123 c::threadA at /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 67
124
125 State 58 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 239 function pthread_mutex_lock_check thread 1
126 c::pthread_mutex_lock_check at concurrent.c line 9
127 c::threadA at /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 67
128
129 State 59 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 242 function pthread_mutex_lock_check thread 1
130 c::pthread_mutex_lock_check at concurrent.c line 9
131 c::threadA at /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 67
132
133 State 60 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 242 function pthread_mutex_lock_check thread 1
134 c::pthread_mutex_lock_check at concurrent.c line 9
135 c::threadA at /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 67
136
137 State 61 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 242 function pthread_mutex_lock_check thread 1
138 c::pthread_mutex_lock_check at concurrent.c line 9
```

```
139 c::threadA at /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 67
140
141 State 62 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 242 function pthread_mutex_lock_check thread 1
142 c::pthread_mutex_lock_check at concurrent.c line 9
143 c::threadA at /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 67
144
145 State 65 file concurrent.c line 10 function threadA thread 1
146 c::threadA at /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 67
147
148 State 69 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 236 function pthread_mutex_lock_check thread 1
149 c::pthread_mutex_lock_check at concurrent.c line 11
150 c::threadA at /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 67
151
152 State 70 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 239 function pthread_mutex_lock_check thread 1
153 c::pthread_mutex_lock_check at concurrent.c line 11
154 c::threadA at /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 67
155
156 State 71 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 242 function pthread_mutex_lock_check thread 1
157 c::pthread_mutex_lock_check at concurrent.c line 11
158 c::threadA at /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 67
159
160 State 72 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 242 function pthread_mutex_lock_check thread 1
161 c::pthread_mutex_lock_check at concurrent.c line 11
162 c::threadA at /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 67
163
```

```
164 State 73 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 242 function pthread_mutex_lock_check thread 1
165 c::pthread_mutex_lock_check at concurrent.c line 11
166 c::threadA at /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 67
167
168 State 74 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 242 function pthread_mutex_lock_check thread 1
169 c::pthread_mutex_lock_check at concurrent.c line 11
170 c::threadA at /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 67
171
172 State 78 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 266 function pthread_mutex_unlock_check thread 1
173 c::pthread_mutex_unlock_check at concurrent.c line 12
174 c::threadA at /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 67
175
176 State 79 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 266 function pthread_mutex_unlock_check thread 1
177 c::pthread_mutex_unlock_check at concurrent.c line 12
178 c::threadA at /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 67
179
180 State 80 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 266 function pthread_mutex_unlock_check thread 1
181 c::pthread_mutex_unlock_check at concurrent.c line 12
182 c::threadA at /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 67
183
184 State 81 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 266 function pthread_mutex_unlock_check thread 1
185 c::pthread_mutex_unlock_check at concurrent.c line 12
186 c::threadA at /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 67
187
188 State 87 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 236 function pthread_mutex_lock_check thread 2
```

```
189 c::pthread_mutex_lock_check at concurrent.c line 20
190 c::threadB at /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 67
191
192 State 88 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 239 function pthread_mutex_lock_check thread 2
193 c::pthread_mutex_lock_check at concurrent.c line 20
194 c::threadB at /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 67
195
196 State 89 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 242 function pthread_mutex_lock_check thread 2
197 c::pthread_mutex_lock_check at concurrent.c line 20
198 c::threadB at /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 67
199
200 State 90 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 242 function pthread_mutex_lock_check thread 2
201 c::pthread_mutex_lock_check at concurrent.c line 20
202 c::threadB at /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 67
203
204 State 91 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 242 function pthread_mutex_lock_check thread 2
205 c::pthread_mutex_lock_check at concurrent.c line 20
206 c::threadB at /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 67
207
208 State 92 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 242 function pthread_mutex_lock_check thread 2
209 c::pthread_mutex_lock_check at concurrent.c line 20
210 c::threadB at /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 67
211
212 State 95 file concurrent.c line 21 function threadB thread 2
213 c::threadB at /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 67
214
```

```
215 State 99 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 236 function pthread_mutex_lock_check thread 2
216 c::pthread_mutex_lock_check at concurrent.c line 22
217 c::threadB at /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 67
218
219 State 100 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 239 function pthread_mutex_lock_check thread 2
220 c::pthread_mutex_lock_check at concurrent.c line 22
221 c::threadB at /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 67
222
223 State 101 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 245 function pthread_mutex_lock_check thread 2
224 c::pthread_mutex_lock_check at concurrent.c line 22
225 c::threadB at /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 67
226
227 State 104 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 236 function pthread_mutex_lock_check thread 1
228 c::pthread_mutex_lock_check at concurrent.c line 13
229 c::threadA at /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 67
230
231 State 105 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 239 function pthread_mutex_lock_check thread 1
232 c::pthread_mutex_lock_check at concurrent.c line 13
233 c::threadA at /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 67
234
235 State 106 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 245 function pthread_mutex_lock_check thread 1
236 c::pthread_mutex_lock_check at concurrent.c line 13
237 c::threadA at /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 67
238
239 State 107 file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 247 function pthread_mutex_lock_check thread 1
```

```
240 c::pthread_mutex_lock_check at concurrent.c line 13
241 c::threadA at /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/
      pthread_lib.c line 67
242
243
244 Violated property:
245   file /tmp/esbmc_release_n70Swf/buildrelease/ansi-c/library/pthread_lib.c
        line 247 function pthread_mutex_lock_check
246   Deadlocked state in pthread_mutex_lock
247   (unsigned int)blocked_threads_count != num_threads_running
248
249 VERIFICATION FAILED
```

Apêndice F

Contraexemplo simplificado para o código da Figura 4.5

```
1 ESBMC version 2.1.0 64-bit x86_64 linux
2 ...
3 Counterexample:
4
5 State 1 file main.c line 4 thread 0
6 ------------------------------------------------------------
7   x = 0 (00000000000000000000000000000000)
8
9 State 2 file main.c line 4 thread 0
10 ------------------------------------------------------------
11   y = 0 (00000000000000000000000000000000)
12
13 State 4 file pthread_lib.c line 71 function pthread_trampoline thread 1
14 ------------------------------------------------------------
15   threadid = 1 (00000000000000000000000000000001)
16
17 State 5 file pthread_lib.c line 72 function pthread_trampoline thread 1
18 ------------------------------------------------------------
19   startdata = { .func=&sum_x, .start_arg=NULL }
20
21 State 6 file main.c line 7 function sum_x thread 1
22 sum_x at pthread_lib.c line 75
23 ------------------------------------------------------------
```

```
24   x = 1 (00000000000000000000000000000001)
25
26 State 7 file pthread_lib.c line 75 function pthread_trampoline thread 1
27 ------------------------------------------------------------
28   exit_val = NULL
29
30 State 8 file pthread_lib.c line 79 function pthread_trampoline thread 1
31 ------------------------------------------------------------
32   __ESBMC_pthread_end_values[1] = NULL
33
34 State 9 file pthread_lib.c line 80 function pthread_trampoline thread 1
35 ------------------------------------------------------------
36   __ESBMC_pthread_thread_ended[1] = TRUE
37
38 State 11 file pthread_lib.c line 71 function pthread_trampoline thread 2
39 ------------------------------------------------------------
40   threadid = 2 (00000000000000000000000000000010)
41
42 State 12 file pthread_lib.c line 72 function pthread_trampoline thread 2
43 ------------------------------------------------------------
44   startdata = { .func=&sum_y, .start_arg=NULL }
45
46 State 13 file main.c line 12 function sum_y thread 2
47 sum_y at pthread_lib.c line 75
48 ------------------------------------------------------------
49   y = -1 (11111111111111111111111111111111)
50
51 State 15 file pthread_lib.c line 75 function pthread_trampoline thread 2
52 ------------------------------------------------------------
53   exit_val = NULL
54
55 State 16 file pthread_lib.c line 79 function pthread_trampoline thread 2
56 ------------------------------------------------------------
57   __ESBMC_pthread_end_values[2] = NULL
58
59 State 17 file pthread_lib.c line 80 function pthread_trampoline thread 2
60 ------------------------------------------------------------
61   __ESBMC_pthread_thread_ended[2] = TRUE
```

```
62
63 State 21 file main.c line 22 function main thread 0
64 main
65 ------------------------------------------------------------
66 Violated property:
67   file main.c line 22 function main
68   assertion
69   (_Bool)(x > 0 && y > 0)
70
71 VERIFICATION FAILED
```