Verificação de *Data Races* em Programas Concorrentes via *SAT Solver*

Matheus R. De Lima[1], **Igor P. Dos Santos**[1]

[1]Departamento de Ciência da Computação – Universidade Federal de Roraima (UFRR)
Boa Vista – RR – Brazil

mthsbusiness91@protonmail.com, igorpadilhads@gmail.com

Abstract. *This paper presents a static analysis tool for detecting data races in concurrent C programs. Using* `pycparser`*, the tool builds an Abstract Syntax Tree (AST) to identify global variables, synchronization primitives, and data accesses. It analyzes control flow to determine if accesses to shared variables are protected by mutexes. The problem is modeled as a boolean satisfiability (SAT) instance, generating a simplified Conjunctive Normal Form (CNF) formula. A simulated SAT solver verifies the formula, identifying potential data races. The tool successfully found a race in an unprotected test case while correctly reporting no races in a protected scenario, validating its effectiveness as a formal verification method.*

Resumo. *Este artigo apresenta uma ferramenta de análise estática para a detecção de data races em programas C. Utilizando a biblioteca* `pycparser`*, a ferramenta constrói uma Árvore de Sintaxe Abstrata (AST) para identificar variáveis globais e primitivas de sincronização. A análise de fluxo determina se os acessos a dados compartilhados estão protegidos. O problema é modelado como uma instância de satisfatibilidade booleana (SAT), utilizando uma fórmula simplificada em Forma Normal Conjuntiva (CNF). Um SAT solver é simulado para verificar a fórmula. A eficácia da ferramenta foi validada com sucesso em um caso de teste desprotegido e, corretamente, em um cenário protegido, demonstrando seu potencial como abordagem de verificação formal.*

1. Introdução

O desenvolvimento de software concorrente, alicerce para otimizar o desempenho do hardware atual, encontra-se frente a desafios complexos, como as condições de corrida (*data races*) se mostrando um dos problemas mais críticos. Entende-se a ocorrência de um *data race* quando múltiplas threads acessam simultaneamente a mesma variável, no qual ao menos uma dessas operações é de escrita, sem a sincronização feita de forma adequada. A natureza não determinística dos data races torna difícil sua identificação e reprodução em testes dinâmicos.

Este trabalho propõe uma abordagem baseada em verificação formal para identificar *data races* em programas C concorrentes. A metodologia se baseia na combinação da análise estática do código-fonte com a modelagem do problema utilizando lógica proposicional, que é resolvida por meio de um *SAT solver*. O artefato desenvolvido examina o código C, gera um modelo de transições, expressa as condições de *data races* em *CNF* e verifica a presença de falhas de concorrência. Este artigo descreve em detalhes a arquitetura do artefato, o método de análise e os resultados obtidos.

2. Metodologia e Ferramentas

O artefato de análise estática foi desenvolvido em Python, fez-se uso da biblioteca `pycparser` para o parseamento de código C. O processo de identificação de *data races* é segmentado em etapas, conforme será descrito abaixo.

2.1. Análise Estática do Código (AST)

A priori é feita a construção de uma Árvore de Sintaxe Abstrata (AST) do código C. Para que isto seja feito, fazemos uso do `pycparser`, no qual permite-nos percorrer a estrutura do programa de forma programática. Além disso, um `NodeVisitor` foi implementado para extrair as seguintes inforrmações:

- **Variáveis Globais:** Todas as variáveis declaradas fora de qualquer função são identificadas e seus nomes armazenados. Essas variáveis são os principais alvos para a verificação de *data races*.
- **Acessos a Variáveis:** O `visitor` rastreia todas as operações de leitura e escrita sobre as variáveis globais, diferenciando `read` (ex: `y = x;`) de `write` (ex: `x = y;`).
- **Chamadas de Sincronização:** A ferramenta identifica chamadas das funções `pthread_mutex_lock` e `pthread_mutex_unlock`, extraindo também o nome do mutex usado como argumento para rastrear seu estado.

2.2. Análise de Fluxo e Detecção de Proteção

Feita a extração dos dados da AST, o artefato realiza uma análise de fluxo para determinar se os acessos realizados nas variáveis globais estão protegidos. Para cada função, os eventos (locks, unlocks e acessos) são ordenados por linha. Após isso, o estado de cada mutex é localizado ao longo do fluxo da função (por exemplo, `locked` ou `unlocked`), permitindo à ferramenta marcar cada acesso como `protected` ou `unprotected`.

Um *data race* é detectado quando o artefato descobre um par de acessos para a mesma variável global em distintas funções, nas quais atendem as seguintes condições:

1. A variável é acessada por pelo menos duas funções distintas.
2. Pelo menos um desses acessos é de escrita.
3. Pelo menos um dos acessos não está protegido por um mutex.

2.3. Modelagem e Resolução com SAT

Para apresentar o uso de verificação formal, o problema de *data race* é modelado em uma fórmula de lógica proposicional. Assim, o problema é transformado em uma instância de satisfatibilidade booleana (*SAT*) na Forma Normal Conjuntiva (*CNF*).

- Se um ou mais *data races* forem detectados, a ferramenta gera uma *CNF* simplificada que é `SAT` (satisfatível).
- Se nenhum *data race* for encontrado, uma *CNF* que resulta em `UNSAT` (não satisfatível) é gerada.

O artefato, dessa forma, simula um *SAT solver* (como o MiniSAT), interpreta o *CNF* gerado e produz um resultado que indica se uma condição de corrida foi detectada. Em caso de `SAT`, um contraexemplo é extraído para auxiliar o desenvolvedor na depuração.

3. Resultados e Validação

O artefato foi testado com três arquivos de código C para validar sua eficácia. No primeiro cenário, que utiliza mutex no código `example.c`, demonstrou uma falha de sincronização. Os dois cenários seguintes foram construídos especificamente para validar, respectivamente, a identificação de um *data race* intencional e a ausência de *data race* (falso-negativo).

3.1. Cenário 1: Teste Inicial (`example.c`)

O teste inicial utilizou um código com uma variável global protegida por um mutex dentro da função `thread_behaviour`. O artefato detectou corretamente os acessos e as chamadas de `pthread_mutex_lock`/`pthread_mutex_unlock`. No entanto, a análise de fluxo revelou que o acesso de leitura à variável global na função `main` (`printf`) estava fora do escopo de qualquer *lock*, sendo classificado como `DESPROTEGIDO`.

Como as condições para um *data race* foram atendidas — acessos concorrentes, sendo pelo menos um de escrita, e um deles desprotegido — o artefato **corretamente detectou um potencial *data race***, gerou uma fórmula em CNF que resultou em `SAT` e forneceu um contraexemplo detalhado, apontando a falha no acesso desprotegido na função `main`.

3.2. Cenário 2: *Data Race* Intencional

Código com duas threads que incrementam uma variável global (`global_counter`) sem mutex. O artefato:

- Identificou a variável `global_counter`.
- Registrou acessos de `read` e `write` desprotegidos.
- Detectou *data race* entre `unprotected_increment` e `main`.
- Gerou uma fórmula *CNF* que resultou em `SAT`.
- Extração do contraexemplo mostrou linhas e funções envolvidas.

```
--- Sumario da Analise Bruta ---
Variaveis Globais Identificadas
  - _Value (Linha: 225, Coluna: 42)
  - global_counter (Linha: 4, Coluna: 5)

--- Analise Detalhada de Transicoes e Protecao por Locks ---

[ Funcao: unprotected_increment ]
  - Linha 9: Acesso 'write' a var 'global_counter'. Status: DESPROTEGIDO
  - Linha 9: Acesso 'read' a var 'global_counter'. Status: DESPROTEGIDO

[ Funcao: main ]
  - Linha 24: Acesso 'read' a var 'global_counter'. Status: DESPROTEGIDO

--- Detecção de Potenciais Data Races ---
Potenciais Data Races Encontrados:
  - Variável 'global_counter' acessada em 'unprotected_increment' e 'main'.
    (Pelo menos um acesso é escrita e pelo menos um é desprotegido)

--- Fórmula CNF Gerada (Simplificada para Demonstração) ---
p cnf 1 1
1 0


--- Simulação da Chamada ao MiniSAT ---
MiniSAT Output: SAT
v 1

Resultado do MiniSAT: SAT

--- Extração de Contraexemplo ---
Contraexemplo Identificado: Um potencial data race foi detectado!!
  Variavel: 'global_counter'
  Funcoes Envolvidas: 'unprotected_increment' e 'main'
  Detalhes dos Acessos Desprotegidos/Envolvidos:
    - Função 'unprotected_increment', Linha 9: write 'global_counter' (DESPROTEGIDO)
    - Função 'unprotected_increment', Linha 9: read 'global_counter' (DESPROTEGIDO)
    - Função 'main', Linha 24: read 'global_counter' (DESPROTEGIDO)
```

Figura 1. Exemplo de saída do artefato para o cenário com *Data Race*.

3.3. Cenário 3: Ausência de *Data Race*

Duas threads apenas leem a variável `readonly_var`:

- Apenas acessos de `read` foram detectados.
- Nenhum mutex foi necessário.
- A lógica de detecção foi satisfeita apenas parcialmente.
- A *CNF* gerada foi vazia, e o resultado foi `UNSAT`.

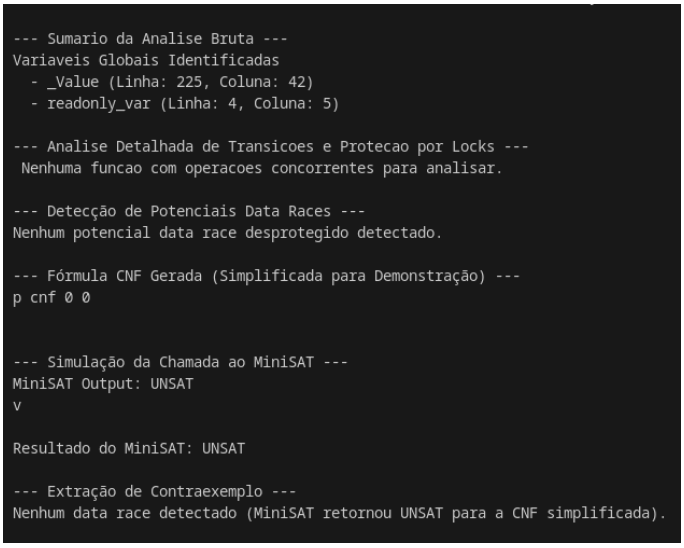

Figura 2. Exemplo de saída do artefato para o cenário sem *Data Race*.

4. Conclusão

O projeto proporcionou um artefato de análise estática funcional para a identificação de *data races* em código C. Apresentou-se uma abordagem que combina a robustez do `pycparser` com a expressividade da lógica proposicional via *SAT*, demonstrando-se eficaz. Além disso, a modelagem via *SAT solver* forneceu uma alternativa poderosa aos testes dinâmicos, com diagnósticos claros via contraexemplos.

Referências

Choi, J.-D. et al. (2002). Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM.

Gray, J. (1985). Why do computers stop and what can be about it. In *Operating Systems: An Advanced Course*, pages 128–145. Springer.

Lamport, L. (1997). How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Transactions on Computers*, 46(7):779–782.

Rabinovitz, I. and Grumberg, O. (2005). Bounded model checking of concurrent programs. In *Lecture Notes in Computer Science*, pages 82–97. Springer.

Sales, E., Inverso, O., and Tuosto, E. (2024). Accurate static data race detection for c. In *Lecture Notes in Computer Science*, pages 443–462. Springer.