VERIFICAÇÃO E COMPROVAÇÃO DE ERROS EM CÓDIGOS C USANDO BOUNDED MODEL CHECKER

HERBERT OLIVEIRA ROCHA

VERIFICAÇÃO E COMPROVAÇÃO DE ERROS EM CÓDIGOS C USANDO BOUNDED MODEL CHECKER

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal do Amazonas como requisito parcial para a obtenção do grau de Mestre em Informática.

ORIENTADOR: PROF. DR. RAIMUNDO DA SILVA BARRETO

Manaus

Fevereiro de 2011

Oliveira Rocha, Herbert
Verificação e Comprovação de Erros em Códigos C usando Bounded Model Checker / Herbert Oliveira Rocha. — Manaus, 2011
xx, 56 f. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal do Amazonas
Orientador: Prof. Dr. Raimundo da Silva Barreto

1. Model Checking. 2. Bounded Model Checker. 3. Propriedades de Segurança. 4. Contra-exemplos. 5. Casos de Teste. 6. Comprovação de Erros. I. Título.

Dedico este trabalho a minha mãe que me impulsionou nesta jornada, concedendo a mim o apoio para sempre persistir em busca de minhas metas.

Agradecimentos

Agradeço primeiramente a Deus que tem me abençoado com capacidade intelectual neste processo de aprendizagem contínua, onde em cada etapa em busca pelo conhecimento, me permitiu deslumbrar novos ambientes, pelo privilégio de compartilhar tamanha experiência, contribuindo assim na consolidação do conhecimento em diferentes vertentes da minha vida.

A minha querida mãe Atenilza que me incentiva a romper e superar desafios, pelo apoio e paciência em cada momento, por estar presente quando necessitei, mesmo que em alguns momentos longe pela distância entre as cidades (Boa Vista - Manaus), por ser um referencial de vida para min.

Ao professor Raimundo Barreto, meu orientador, por todo o seu apoio e presteza no auxílio às atividades e discussões sobre o andamento e normatização deste trabalho e principalmente por acreditar no desenvolvimento do meu trabalho, proporcionando-me novas experiências que contribuíram significativamente no meu processo de aprendizagem.

Agradeço também aos amigos, colegas de laboratório, colegas de classe pela espontaneidade e alegria na troca de informações e materiais numa rara demonstração de amizade e solidariedade. A Lucas Cordeiro pelo incentivo, simpatia e presteza no auxílio às atividades e esclarecimento nas discussões no decorrer deste trabalho.

Aos professores que tanto me ajudaram e incentivaram nesta longa jornada que agora conclui mais uma etapa e aos demais idealizadores, coordenadores e funcionários da Universidade Federal do Amazonas (UFAM), em especial ao grupo do departamento de pós-graduação em informática (PPGI). Ao CNPq pelo suporte no processo de pesquisa. Enfim, agradeço a todos que de alguma forma contribuíram para que eu conseguisse ultrapassar as dificuldades e alcançar meu objetivo.

"A História tem demonstrado que os mais notáveis vencedores normalmente encontraram obstáculos dolorosos antes de triunfarem. Eles venceram porque se recusaram a se tornarem desencorajados por suas derrotas."

(Bryan Forbes)

Resumo

A utilização de sistemas baseados em computador em diversos domínios aumentou significativamente nos últimos anos. Um dos principais desafios no desenvolvimento de *software* de sistemas críticos é a garantia da sua correção e confiabilidade. Desta forma, a verificação de *software* exerce um papel importante para assegurar a qualidade geral do produto, visando principalmente características como previsibilidade e confiabilidade. No contexto de verificação de *software*, os *Bounded Model Checkers* estão sendo utilizados para descobrir erros sutis em projetos de sistemas de *software* atuais, contribuindo eficazmente neste processo de verificação. O valor dos contra-exemplos e propriedades de segurança gerados pelo Bounded Model Checkers para criar casos de testes e para a depuração de sistemas é amplamente reconhecido. Quando um *Bounded Model Checking* (BMC) encontra um erro ele produz um contra-exemplo. Assim, o valor dos contra-exemplos para depuração de software é amplamente reconhecido no estado da prática. Entretanto, os BMCs frequentemente produzem contra-exemplos que são grandes ou difíceis de entender ou manipular, principalmente devido ao tamanho do *software* e valores escolhidos pelo solucionador de satisfabilidade. Neste trabalho visamos demonstrar e analisar o uso de método formal (através da técnica model checking) no processo de desenvolvimento de programas na linguagem C, explorando as características já providas pelo model checking como o contra-exemplo e a identificação e verificação de propriedades de segurança. Em face disto apresentamos duas abordagens: (i) descrevemos um método para integrar o *Bounded Model Checker* ESBMC como o *framework* de teste unitário CUnit, este método visa extrair as propriedades geradas pelo ESBMC para gerar automaticamente casos de teste usando o rico conjunto de assertivas providas pelo *framework* CUnit e (ii) um método que visa automatizar a coleta e manipulação dos contra-exemplos, de modo a instanciar o programa C analisado, para comprovar a causa raiz do erro identificado. Tais métodos podem ser vistos como um método complementar para a verificação efetuada pelos BMCs. Demonstramos a eficácia dos métodos propostos sobre *benchmarks* públicos de código C.

Palavras-chave: Model Checking, Bounded Model Checker, Propriedades de Segurança, Contra-exemplos, Casos de Teste, Comprovação de Erros.

Abstract

The use of computer-based systems in several domains has increased significantly over the last years, one of the main challenges in software development of these systems is to ensure the correctness and reliability of these. So that software verification now plays an important role in ensuring the overall product quality, aimed mainly the characteristics of predictability and reliability. In the context of software verification, with respect to the use of model checking technique, Bounded Model Checkers have already been applied to discover subtle errors in actual systems projects, contributing effectively in this verification process. The value of the counterexample and safety properties generated by Bounded Model Checkers to create test case and to debug these systems is widely recognized. When a Bounded Model Checking (BMC) finds an error it produces a counterexample. Thus, the value of counterexamples to debug software systems is widely recognized in the state-of-the-practice. However, BMCs often produce counterexamples that are either large or difficult to be understood and manipulated mainly because of both the software size and the values chosen by the respective solver. In this work we aim to demonstrate and analyze the use of formal methods (through using the model checking technique) in the process of developing programs in C language, exploring the features already provided by the model checking as the counterexample and the identification and verification of safety properties. In view of this we present two approaches: (i) we describe a method to integrate the bounded model checker ESBMC with the CUnit framework. This method aims to extract the safety properties generated by ESBMC to generate automatically test cases using the rich set of assertions provided by the CUnit framework and (ii) a method aims to automate the collection and manipulation of counterexamples in order to instantiate the analised C program for proving the root cause of the identified error. Such methods may be seen as a complementary technique for the verification performed by BMCs. We show the effectiveness of our proposed method over publicly available benchmarks of C programs.

Keywords: Model Checking, Bounded Model Checker, Safety Properties, Counterexamples, Test Cases, Proving Errors.

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

Atualmente, as aplicações de software precisam ser desenvolvidas rapidamente e atingir um alto nível de qualidade. Afim de se obter este objetivo, a execução do *software* deve ser controlada, e da mesma forma, deve-se buscar formas de garantir que as propriedades definidas sejam atingidas. Isto requer que esses tipos de aplicações sejam projetadas considerando as exigências dos requisitos de previsibilidade e confiabilidade, principalmente em aplicações de sistemas críticos. Sendo assim, faz se necessárias verificações e análises detalhadas, que comprovem que os comportamentos estão de acordo com as respectivas especificações destes *software*.

Outro fator que tem tornado-se um grande desafio no desenvolvimento de sistemas de *software* tem sido a verificação de grandes quantidades de LOCs (linhas de código), que pode variar de milhares até dezenas de milhões, de maneira que a reutilização de código se tornou mais comum [Wang et al., 2008]. Como conseqüência, os erros durante o desenvolvimento de *software* tornam-se mais comum.

Segundo Nagarakatte et al. [2010], a linguagem de programação C/C++ é de fato o padrão para a implementação de diversos tipos de *software*, incluindo *software* crítico. Baseado nestes fatos, diversas estratégias de verificação e de teste estão sendo pesquisadas e aplicadas para garantir a qualidade das aplicações de *software*. As técnicas tradicionais de validação de sistemas, como a simulação, já não são suficientes para verificação e exatidão de tais sistemas. Em primeiro lugar estes métodos podem explorar apenas uma pequena fração do comportamento do sistema, em segundo lugar os erros encontrados tardiamente durante a fase de prototipagem têm um impacto negativo no mercado e, por último, devido à forte dependência em sistemas computadorizados, uma falha pode conduzir a situações catastróficas.

O foguete Ariane 5[1] (projeto europeu), é um exemplo de *software* que não funcionou a contento porque poucos segundos após seu lançamento ele explodiu devido à falhas de programação. Um meio de lidar com este tipo de problema é construir modelos que venham prever um determinado comportamento de um artefato. Outra opção seria usar algum tipo de linguagem específica, ou ferramenta, para representar e analisar os problemas em um dado domínio, como por exemplo a linguagem Promela utilizada no *model checker* SPIN[Ben-Ari, 2008].

A verificação formal tem desempenhado um papel importante para assegurar a previsibilidade e a confiabilidade na concepção de aplicações críticas. Como exemplo, atualmente a indústria tem utilizado técnicas de verificação formal como o *model checking*, que é uma técnica para verificação de sistemas de estados finitos. Basicamente, o *model checking* usa uma busca exaustiva no espaço dos estados do sistema, para determinar se uma dada propriedade é verdadeira ou não [Clarke et al., 1999].

Model Checking tem sido uma técnica interessante de pesquisa na área de verificação formal por muitos anos. Nas últimas décadas, o uso de modelos tem sido adotado em diferentes categorias de *software*, tais como UML [Borges & Mota, 2007] Model-Driven Development [France & Rumpe, 2007], and Model-Driven Testing [Petrenko & Alexandre, 2001]. Este cenário também motivou os engenheiros de *software* a utilizar *model checking* [Beyer et al., 2007a]. *Model Checking* também é definido como uma estratégia para verificar a exatidão dos sistemas, verificando se o sistema atende as propriedades desejadas com base em sua representação em um modelo formal [Baier & Katoen, 2008].

Os principais desafios em *model checking* estão em como lidar com a explosão do espaço de estados do modelo, integração com outros ambientes de testes mais familiares aos projetistas e tratamento e análise de contra-exemplos. Neste trabalho visamos demonstrar e análisar o uso de método formais (através da técnica *model checking*) no processo de desenvolvimento de programas na linguagem de programação C. Em face disto apresentamos duas abordagens, a primeira que visa a integração da técnica *model checking* com o ambiente de teste unitário CUnit[2], objetivando a checagem de propriedades de segurança, e a segunda para análise e abstração de dados do contra-exemplo para a comprovação de erros.

De modo a lidar com o problema da integração do *model checking* com outras técnicas de teste ou verificação, a primeira abordagem tem como uma possível solução explorar as características já providas pelo *model checking* (por exemplo, a identificação das propriedades de segurança) para a geração de casos de teste que serão implementa-

[1] http://www.sbmac.org.br/bol/bol-2/artigos/ariane5.html
[2] Disponível em: http://cunit.sourceforge.net/

dos no ambiente de teste unitário. Segundo Baier & Katoen [2008], uma propriedade de segurança é frequentemente caracterizada como "*nada de ruim deve acontecer*". Como exemplo, a propriedade da exclusão mútua é uma típica propriedade de segurança. Deste modo, a violação de uma propriedade de segurança pode ser detectada pela monitoração da execução do sistema. Esse acompanhamento é geralmente feito de uma maneira controlada, tendo em conta tanto a entrada para o sistema, assim como os vários caminhos de sua execução.

Nesta primeira abordagem apresentamos um método para integrar o *model checker* ESBMC (*Efficient SMT-Based Bounded Model Checker*)[Cordeiro et al., 2009b] com o *framework* de testes CUnit. A integração destes dois ambientes visa garantir a qualidade do *software* através da exploração da verificação formal e testes sem a obtenção de uma explosão de mémoria. Basicamente, este método extrai as propriedades de segurança geradas automaticamente pelo ESBMC e gera casos de testes, usando o rico conjunto de assertivas providas pelo CUnit.

A segunda abordagem visa a aplicação da verificação formal de *software* para coletar dados (do contra-exemplo gerado) para comprovação de erros pela instanciação e execução dos mesmos. Os *Bounded Model Checking* (BMC) tem ganhado popularidade devido a sua habilidade de lidar com semânticas completas de linguagens de programação atuais, e suporte a verificação de um rico conjunto de propriedades. A aplicação de *model checking* garante que o sistema tem as propriedades desejadas ou não, e se não um contra-exemplo é gerado [Cordeiro et al., 2009b].

A motivação desta segunda abordagem tem como base que a coleta dos dados nos contra-exemplos gerados pelas ferramentas de verificação não é uma tarefa trival, geralmente são difíceis de ser entendidos ou manipulados, principalmente devido a quantidade de variáveis e valores que podem estar envolvidos. Outro fato seria os valores escolhidos pelos solucionadores de satisfabilidade, neste trabalho utilizamos BMC baseados em *Satisfiability Modulo Theories* (SMT). O método apresentado nesta abordagem visa contribuir como um complemento para a verificação executada pelo ESBMC, facilitando o processo de coleta e manipulação das informações geradas pela verificação da ferramenta, deste modo também provendo um meio de validar o contra-exemplo gerado pelo ESBMC, evitando assim falsos contra-exemplos.

Basicamente o método para esta segunda abordagem funciona da seguinte forma, a partir dos contra-exemplos gerados pelo BMC durante a fase de verificação, o método utiliza os dados providos neste contra-exemplo, tais como: (i) as variáveis envolvidas, (ii) os valores atribuídos a estas variáveis, e (iii) a linha onde a variável está localizada no código, para detectar a causa raiz do erro. Adicionalmente, o método proposto executa a instanciação do código analizado com os dados providos pela identificação do erro.

Demonstramos a eficácia dos métodos propostos através da aplicação dos mesmos sobre diversos *benchmarks* públicos de programas em C, que variam erros como violação de limites superiores ou inferiores de *arrays*, *buffer overflow* e *underflow*, segurança de ponteiros e alocação de mémoria dinâmica. Acreditamos que a automatização destes processos permitirá divulgar a aplicação de métodos formais, contribuindo com engenheiros e desenvolvedores não muito familiarizados com este assunto, e aprofundando a verificação de programas em código C.

Este Capítulo apresenta o contexto deste trabalho, descreve o problema a ser resolvido, apresenta a motivação e objetivos, explica como o problema está planejado para ser resolvido, resume as contribuições e finalmente, mostra a estrutura deste trabalho.

1.1 Contexto

A Figura 1.1 mostra uma visão geral do trabalho. O contexto deste trabalho está situado no uso de metodologias e técnicas de verificação formal, focando principalmente em *Bounded Model Checking*. Este trabalho está interessado especificamente na parte de geração de casos de testes baseados em propriedades de segurança e na comprovação de erros pela utilização de contra-exemplo de *model checkers*. O escopo deste trabalho é restrito a análise e verificação de códigos na linguagem de programação C.

A verificação formal é dividida em duas partes, que é a verificação de *hardware* e a verificação de *software*, neste trabalho estaremos abordando a verificação de *software*. O núcleo deste trabalho está em explorar as características providas pelos *Bounded Model Checkers* em específico a identificação das propriedades de segurança e os contra-exemplos, de modo a prover uma verificação complementar a efetuada por estes, facilitando a utilização desta técnica de verificação formal por pessoas não familiarizadas com a mesma.

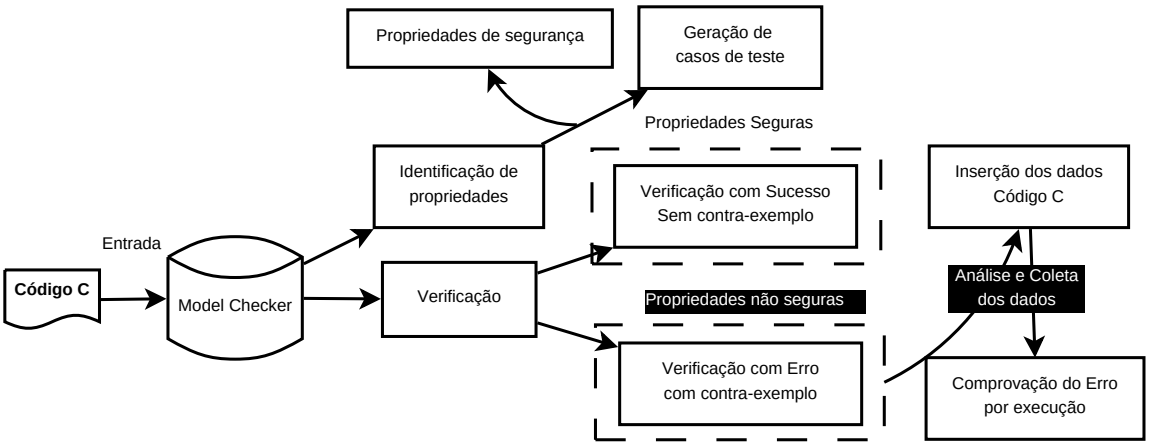


Figura 1.1. Visão geral do trabalho

1.2 Definição do Problema

Segundo Clarke [2008], *model checking* tem diversas vantagens se comparadas a outras técnicas, tais como, prova por teorema e verificação de provas. Segue uma lista parcial de algumas dessas vantagens: (i) o usuário de um *model checker* não necessita construir provas de corretude; (ii) o processo de verificação é automático; (iii) *model checking* é rápido se comparado a outros métodos, como por exemplo, o *proof checker*, no qual pode requerer meses de tempo de trabalho do usuário em modo interativo; (iv) diagnósticos de contra-exemplos, se uma especificação (propriedade) não é satisfeita o *model checker* produzirá um contra-exemplo do caminho da execução, demonstrando o porquê da especificação não está segura; e (v) lógicas temporais pode ser facilmente expressas.

Dentre as vantagens citadas, também observa-se algumas desvantagens do *Model Checking* [Clarke, 2008]: (i) o *model cheking* não comprova o erro identificado no programa, o que ajudaria no entendimento do mesmo; (ii) a escrita de especificações nem sempre é uma tarefa trivial; e (iii) o problema da explosão do espaço de estados, que é causado devido ao grande número global dos estados de um sistema concorrente ou devido a uma estrutura de dados, pode chegar a números exponenciais. Todos os *model checkers* sofrem destes problemas.

O problema considerado neste trabalho é expresso na seguinte questão: *Como utilizar as características providas pelos Bounded Model Checkers, como identificação das propriedades de segurança e contra-exemplos, para complementar a verificação efetuada em código C, de tal modo que as suas desvantagens possam ser contornadas e as propriedades verificadas possam ser validadas?*

1.3 Motivação/Justificativa

Atualmente existem inúmeros sistemas aplicados a áreas muito distintas como: ambientais, em aeronaves, militares, médicas etc. Apesar dos diferentes graus de prejuízo que uma eventual falha em algum destes sistemas possam provocar, um alto grau de confiabilidade é exigido para quase todos eles. Estas afirmações se sustentam pelo fato de que eventuais falhas podem resultar em graves transtornos econômicos para as empresas e instituições envolvidas, como por exemplo, no caso do processador da *Intel Pentium II*, no qual foi detectado um erro com relação a ponto flutuante[3] que contabilizou à empresa um prejuízo de aproximadamente US$ 450.000.000,00.

Um dos motivos desde trabalho está vinculado à necessidade de se garantir a corretude dos sistemas, que é acompanhada pelo aumento do grau de complexidade exigido para os novos sistemas, ao mesmo tempo em que, por pressão econômica, os prazos de entrega se mantêm os mesmos ou até menores. Tanto o aumento de complexidade quanto a diminuição no prazo de entrega, tornam bem mais difícil a tarefa da verificação do sistema a ser entregue. Isto estimula a pesquisa por alternativas automáticas de verificação e correção de *software*[D'Silva et al., 2008].

A otimização da técnica de *model checking* na linguagem C para o desenvolvimento de *software* é um fator importante para garantir a validade do comportamento e propriedades acerca do *software*. O desenvolvimento e otimização desta técnica contribuirá para a melhoria de qualidade no desenvolvimento de *software*, já que normalmente existe um alto custo envolvido na preparação, execução e gerenciamento dos testes e verificação destes[Sherer, 1991]. Estas afirmações se sustentam pelo fato que ainda há muito campo para pesquisas na área que consideram a criação de ferramentas e métodos com a aplicação em *model checking* ou mesmo a otimização de ferramentas e métodos já existentes[Clarke, 2008].

1.4 Objetivos

Ao se relacionar com o problema definido na Seção 1.2, o principal objetivo deste trabalho é *demonstrar e análisar uma metodologia para verificação de propriedades de segurança e comprovação de erros baseado em contra-exemplos gerados por Bounded Model Checkers com aplicação na linguagem de programação C.*

[3] http://pt.wikipedia.org/wiki/Defeito_de_ponto_flutuante

Os objetivos específicos são:

1. Demonstrar o método proposto para a geração de casos de teste, complementar a outros métodos, baseados em assertivas, utilizando as propriedades de segurança identificadas no código C por um *Bounded Model Checker*;

2. Demonstrar o método proposto para integrar *Bounded Model Checkers* com teste unitário para efetuar a verificação de propriedades de segurança;

3. Demonstrar o método proposto para instanciar os dados coletados do contra-exemplo (gerado por um *Bounded Model Checker*) no código C analisado, afim de comprovar o erro identificado;

4. Analisar a aplicação dos métodos sobre *benchmarks* públicos de programas em C, afim de examinar a sua eficácia e aplicabilidade.

1.5 Organização do Trabalho

No Capítulo 1, apresentamos a introdução composta pelo contexto, definição do problema, motivações e objetivos. No Capítulo 2, é discutido os conceitos e definições de Verificação Formal com *Model Checking*, Bounded Model Checking com ESBMC, Explorando Propriedades de Segurança em Estratégias de Teste de *Software*, Contra-Exemplos e Estratégias de Análise de Programas. No Capítulo 3 é analisado os principais trabalhos correlatos. No Capítulo 4 é explicitado o método proposto. No Capítulo 5, é apresentado os resultados experimentais. E por fim no Capítulo 6, é exposto as conclusões.

Capítulo 2

Conceitos e Definições

Este Capítulo introduz os conceitos necessários para o entendimento deste trabalho. Este Capítulo é dividido em quatro seções: Verificação Formal com *Model Checking*, Bounded Model Checking com ESBMC, Explorando Propriedades de Segurança em Estratégias de Teste de *Software*, Contra-Exemplos e Estratégias de Análise de Programas.

2.1 Verificação Formal com *Model Checking*

Sistemas de *software* e *hardware*, inevitavelmente crescem em dimensão e funcionalidade. Devido a este aumento, e consequentemente de sua complexidade, a probabilidade de erros é muito maior. Além disso, alguns destes erros podem causar perda de dinheiro, tempo, ou mesmo de vidas humanas. Um dos principais objetivos da engenharia de software é permitir que os desenvolvedores possam construir sistemas seguros, apesar da complexidade. Uma forma de alcançar este objetivo é por meio de métodos formais, que são linguagens matemáticas, onde por meios de técnicas e ferramentas é possível especificar e verificar tais sistemas [Clarke & Wing, 1996].

Model Checking é uma técnica baseada em métodos formais utilizada para a análise e validação de sistemas. Esta é uma técnica automática que gera uma busca exaustiva no espaço de estados do modelo, para determinar se uma dada propriedade é válida ou não [Baier & Katoen, 2008]. O uso de métodos formais colaboram significativamente para a nossa compreensão do sistema, revelando inconsistências, ambiguidades e incompletude que poderiam passar despercebidos [Clarke & Wing, 1996].

A verificação de um sistema é um aspecto fundamental para a aplicação de métodos formais. Esta verificação consiste em fazer um modelo do sistema que contenha suas propriedades mais relevantes [Fisteus, 2005]. Segundo Clarke et al. [1999], aplicação de testes em um modelo consiste basicamente em:

- **Modelagem**, esta é a primeira tarefa, a de converter um projeto em um formalismo aceito por uma ferramenta de verificação de modelos. Em muitos casos, isso é simplesmente uma tarefa de compilação, em outros casos, devido às limitações do tempo e da memória, a modelagem de um trabalho pode exigir a utilização da abstração de informações, para eliminar detalhes irrelevantes ou sem importância.

- **Especificação**, é necessária para determinar as propriedades que o modelo deve satisfazer, geralmente é dado em algum formalismo lógico. Para *hardware* e sistemas de *software*, é comum usar a lógica temporal, que pode valer como o comportamento do sistema.

- **Verificação**, o ideal seria que a verificação fosse completamente automática. No entanto, na prática, muitas vezes envolve assistência humana. Em caso de resultado negativo, muitas vezes é fornecido ao usuário o rastreamento do erro, este pode ser usado como um contra-exemplo para a propriedade testada e pode ajudar o projetista no rastreamento do erro. A análise do rastreamento do erro pode exigir uma modificação no sistema e a reaplicação do algoritmo de verificação de modelo.

O conceito de *model checking* engloba um conjunto de algoritmos eficientes que permitem verificar propriedades de modelos com um número finito de estados, que normalmente são expressos mediante lógica temporal. As lógicas temporais são formalismos mais freqüentemente utilizados para expressar as propriedades a serem verificadas em modelos baseados em estados e transições. A lógica de proposições permite expressar propriedades relativas ao estado do modelo, mediante proposições atômicas e conectores lógicos (negação, conjunção, disjunção, condicional e bicondicional) [Baier & Katoen, 2008].

A lógica temporal é suficiente para verificar os programas do ponto de vista da sua entrada e saída. Contudo, para os sistemas reativos ou concorrentes também é relevante descrever as transições entre estados, ou seja, detalhes internos sobre a evolução da computação. Isto é especialmente importante nos sistemas que funcionam continuamente, uma vez que nunca terminam [Fisteus, 2005].

Existem diferentes formas de lógicas temporais, dependendo do modelo de tempo subjacente, como por exemplo, LTL, PCTL e a lógica CTL *(Computation Tree Logic)*

[Baier & Katoen, 2008]. Segundo Clarke et al. [1999], a lógica CTL baseia-se na lógica proposicional de ramificação do tempo, ou seja, uma lógica onde o tempo pode evoluir para mais do que um possível futuro, usando um modelo de tempo discreto. As fórmulas em CTL são compostas por: proposições atômicas, conectivos booleanos e operadores temporais. Os operadores temporais consistem em operadores de tempo futuro [Clarke et al., 1999]:

- **G** (*always* ou *globally*), especifica que uma propriedade é válida em todos os estados no caminho;
- **F** (*eventually* ou *in the future*) é usada para afirmar que a propriedade estará válida em algum estado no caminho;
- **X** (*next time*) exige que a propriedade esteja válida no segundo estado no caminho;
- **U** (*until*) assegura se existe um estado ao longo do caminho onde a segunda propriedade está válida e em cada estado anterior no caminho, a primeira propriedade é assegurada;
- **R** (*release*) requer que a segunda propriedade seja válida ao longo do caminho até incluir o primeiro estado onde a primeira propriedade é válida.

O operadores de tempo são precedidos por quantificadores (**A** - todas as execuções, **E** - existe uma execução), conforme demonstrado na Figura 2.1, para os operadores **G**, **F** e **X** que são normalmente os mais utilizados. Em CTL, o tempo não é mencionado explicitamente. Os operadores temporais apenas permitem descrever propriedades em termos de "no próximo", "eventualmente" ou "sempre".

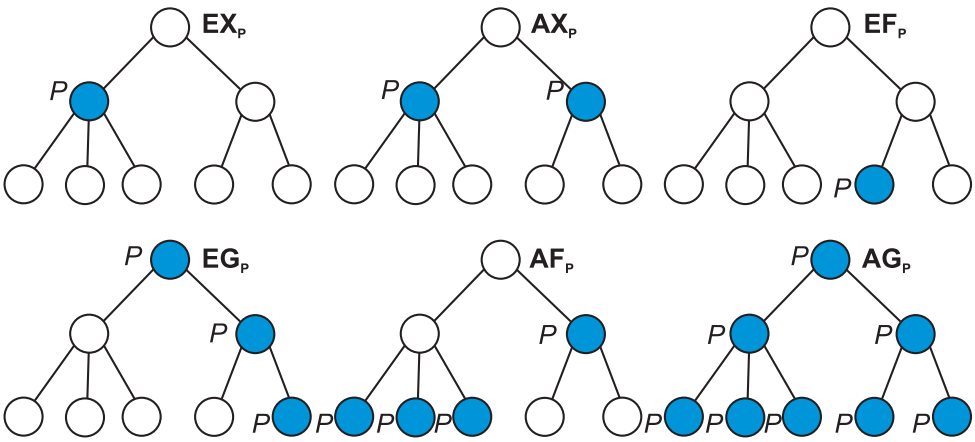


Figura 2.1. Operadores Temporais

2.2 *Bounded Model Checking* com ESBMC

Os *software* de verificação de modelos têm sido um vibrante ramo da pesquisa em métodos formais por muitos anos. No contexto de *model checking* existe um tipo especial de *model checking* denominado de *bounded model checking* (BMC) baseado em *Boolean Satisfiability* (SAT), no qual tem sido introduzido como uma técnica complementar para Diagramas de Decisão Binária (do inglês *Binary Decision Diagrams , BDD*) para aliviar o problema da explosão de estados.

A idéia básica do BMC é verificar (a negação de) uma dada propriedade em uma dada profundide: dado um sistema de transições M, uma propriedade ϕ, e um limite (bound) k, o BMC desenrola o sistema k vezes e traduz ele em uma condição de verificação (CV) ψ tal que ψ é satisfeita se e somente se ϕ tem um contra-exemplo de profundidade menor ou igual a k. Os verificadores SAT padrões podem ser usados para verificar se ψ é satisfatível.

Afim de lidar com o complexo crescimento dos sistemas, existem os SMT (*Satisfiability Modulo Theories*) *solvers* que podem ser usados com base para resolver as CVs geradas a partir das instâncias do BMC. Segundo Cordeiro et al. [2009b] o SMT decide a satisfabilidade de fórmula de primeira ordem usando a combinação de diferentes teorias de bases e assim generalizar a satisfabilidade proposicional provendo suporte a funções não interpretadas, aritmética, *bit-vectors*, tuplas, *arrays* e outras teorias decidíveis de primeira ordem.

Os solucionadores de SMT funcionam da seguinte forma, dada um teoria τ e uma fórmula livre de quantificadores ϕ, dizemos que ϕ é uma τ satisfatível se e somente se existe uma estrutura que satisfaz tanto a fórmula com a sentença de τ, ou equivalentemente, se $\tau \cup \phi$ é satisfatível [Bradley & Manna, 2007]. O estado da arte dos solucionadores SMT não somente combinam diferentes teorias dividíveis, mas também a integração de solucionadores SAT (*Boolean Satisfiability*) de modo a melhorar o desempenho no processamento.

ESBMC[1] é um *bounded model checker* para softwares embarcados em ANSI-C baseado em *SMT solvers*, no qual permite: (i) verificar *software single* e *multithread* (com variáveis compartilhadas e alocadas); (ii) verificação aritimética de *underflow* e *overflow*, segurança de ponteiros, limites de *arrays*, atomicidade, *deadlock*, *data race* e assertivas providas pelo usuário; (iii) verificar programas que fazem o uso de *bit-level*, ponteiros, *structs*, *unions* e aritmética de ponto fixo. ESBMC utiliza uma versão modificada do CBMC[2] no *front-end* para analisar o código ANSI-C e para gerar as

[1] Available at http://users.ecs.soton.ac.uk/lcc08r/esbmc/
[2] Available at http://www.cprover.org/cbmc/

CVs através de execução simbólica.

ESBMC provê três abordagens para a verificação de modelos em *software multi-threaded*: (i) a abordagem *lazy*, onde ESBMC gera todas os possíveis *interleavings* e chama o *SMT solver* em cada uma delas individualmente, até ele encontrar o *bug*, ou ter sistematicamente explorado todos os *interleavings*; (ii) a abordagem *schedule recording*, ESBMC codifica todos os possíveis *interleavings* em uma única fórmula e então explora o desempenho dos *SMT solvers*; e (iii) a abordagem *underapproximation* e *widening (UW)*, ESBMC reduz o espaço dos estados pela abstração dos *interleavings* a partir da prova de não satisfabilidade gerada pelos *SMT solvers*.

A Figura 2.2 demonstra os principais componentes de software do ESBMC. As caixas brancas (exceto para o *SMT solver*) representa os components que são herdados do CBMC sem nenhuma modificação enquanto que as caixas cinzas com linhas tracejadas representam os componentes que são modificados de modo a melhorar a análise inicial no GFC (Grafo de Fluxo de Controle) do programa para determinar a melhor codificação e solucionador de satisfabilidade para um particular programa e dar suporte a verificação de *software multi-threaded*.

As caixas cinzas com linhas sólidas,na Figura 2.2, representam os novos componentes que são implementados para codificar as atribuições e propriedades dadas de um programa ANSI-C para um contexto lógico global, usando os conceitos e definições das teorias suportadas pelos *SMT solvers*. O ESBMC também implementa um novo componente para interpretar os contra-exemplos gerados pelos *SMT solvers* suportados. As caixas sombreadas indicam os componentes do software que devem ser implementados no *back-end* para suportar cada novo *SMT solver* [Cordeiro et al., 2009b].

Este trabalho visa explorar as características providas pelo ESBMC de modo a propor uma abordagem complementar para testes baseados em projeto, tendo em consideração as limitações do BMC, tal como a explosão nos espaços do estado para a verificação de satisfabilidade das CVs pelo uso de *SMT-solvers*.

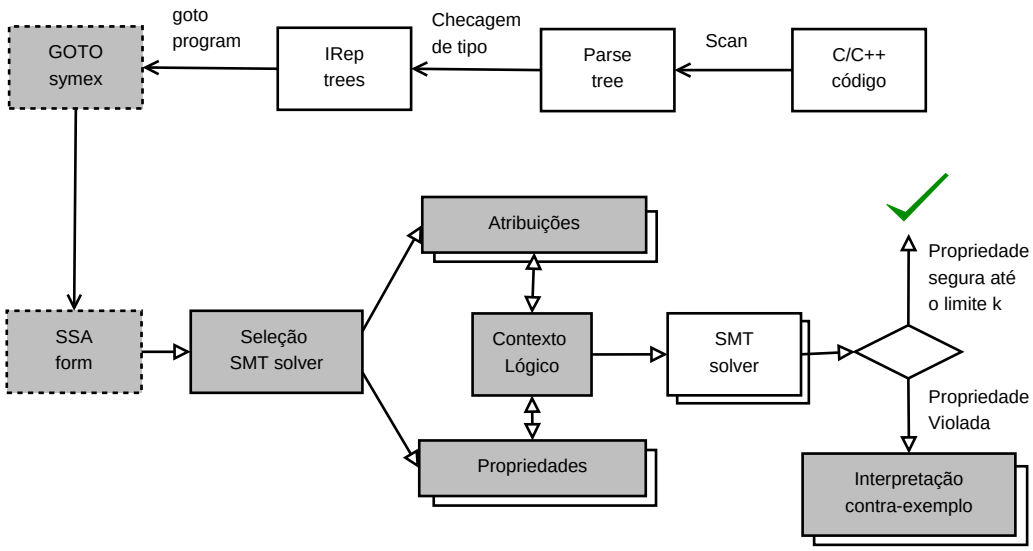


Figura 2.2. Visão geral da estrutura do ESBMC.

2.3 Explorando Propriedades de Segurança em Estratégias de Teste de *Software*

Segundo Myers & Sandler [2004], teste de *software* é o processo de execução de um programa com o objetivo de encontrar uma falta. Um teste de sucesso é aquele no qual se pode determinar os casos de teste para qual o programa sob teste falhou. Um caso de teste consiste de uma análise das informações associadas ao teste com um resultado esperado do processamento segundo as especificações do software. Tipicamente os testes unitários são escritos baseados em um conjunto de casos de teste para garantir que o código atende a sua concepção e se comporta como o esperado.

Existem diversas ferramentas para o desenvolvimento de testes unitários, tais como: GoogleTest [3], JUnit [4], PyUnit [5] e EmbeddedUnit [6]. Tais ferramentas permitem as engenheiros criar testes unitários de uma forma mais eficiente e simplificada, favorecendo uma melhor organização e reutilização de código. O CUnit [7] é um *framework* de teste unitário para programas em C que prove um conjunto de assertivas para testes de condição lógica. Algumas assertivas providas pelo CUnit são demonstradas na Tabela 2.1.

[3] http://code.google.com/p/googletest/
[4] http://junit.org/
[5] http://pyunit.sourceforge.net/
[6] http://embunit.sourceforge.net/
[7] http://cunit.sourceforge.net/

Tabela 2.1. Algumas assertivas do CUnit.

Asserts	**Parâmetros**
CU_ASSERT and CU_ASSERT_FATAL	(expressão com valor inteiro)
CU_ASSERT_TRUE CU_ASSERT_TRUE_FATAL	(valor da expressão)
CU_ASSERT_PTR_EQUAL CU_ASSERT_PTR_EQUAL_FATAL	(valor atual, valor esperado)
CU_ASSERT_DOUBLE_EQUAL CU_ASSERT_DOUBLE_EQUAL_FATAL	(valor atual, valor esperado, granularidade)
CU_PASS	(mensagem) - Registra uma mensagem especificada.

As etapas de configuração dos teste com o CUnit normalmente seguem a seguinte forma [8]:

- (a) Escrita de funções para teste (e `init/cleanup`, se necessário);
- (b) Inicialização dos registros do teste - `CU_initialize_registry()`;
- (c) Adicionar suites de teste - `CU_add_suite()`;
- (d) Adicionar testes - `CU_add_test()`;
- (e) Execução dos testes usando uma interface definida, por exemplo, `CU_console_run_tests`;
- (f) Limpeza do registro de teste - `CU_cleanup_registry`.

O sucesso ou falha das assertivas providas nos testes são controladas pelo CUnit, e podem ser visualizadas quando o teste é executado por completo. Cada assertiva é um teste de condição lógica, e se ele falhar a condição é avaliada como *FALSE*. Neste trabalho, apresentamos uma abordagem para criar casos de teste guiados pelas propriedades de segurança que são geradas automaticamente pelo ESBMC.

Uma propriedade de segurança (do inglês *safety property*[Baier & Katoen, 2008]) é definida segundo Clarke et al. [2003] como: dado um sistema de transições $ST = (S, S_0, E)$, seja um conjunto $B \subset S$ que especifica um conjunto de *maus estados* tais que $S_0 \cap B = \emptyset$, podemos dizer que TS *é seguro com relação a* B, denotada por $TS \models \mathbf{AG}\neg B$ se não existe um caminho no sistema transição do estado inicial S_0 até o *mau estado* B. De outro modo dizemos que TS não é seguro, denotado por $TS \not\models \mathbf{AG}\neg B$.

Informalmente podemos dizer que uma propriedade em tempo linear especifica o admissível (ou desejado) comportamento de um sistema [Baier & Katoen, 2008], por

[8] http://cunit.sourceforge.net/doc/writing/tests.html

exemplo, a temperatura de um reator nuclear nunca deve exceder a 100^0C. Se um sistema falha ao satisfazer uma propriedade de segurança, então existe uma execução finita que revela esta fato. Assim a verificação da corretude do sistema com respeito as propriedades de segurança é um meio para validar o comportamento do sistema.

O ESBMC gera condições de verificação para as seguintes propriedades:

1. CVs para a checagem aritmética de *underflow* e *overflow* seguindo o padrão ANSI-C, ou seja, a codificação aritmética para um *overflow* do tipo *unsigned integer* (ou seja, *unsigned int, unsigned long int*) é verificada sobre a sua condição de contorno sobre a adição, subtração, multiplicação, divisão e operações de negação;

2. CVs para os limites da indexação dos *arrays*, ou seja, checa se a operação com os índices do *array* estão fora do limite definido;

3. CVs para segurança de ponteiros, ou seja, verifica se o ponteiro [9] faz referências a um objeto correto (representado por $SAME\_OBJECT$) e também checa se o ponteiro é `NULL` ou um objeto inválido (representado por $INVALID\_POINTER$);

4. CVs para alocação de memória dinâmica, o ESBMC verifica os argumentos para qualquer *malloc, free* ou operação de *dereferencing* é um objeto dinâmico (representado por $IS\_DYNAMIC\_OBJECT$) e se o argumento para qualquer *free* ou operação de *dereferencing* é ainda um objeto válido ($VALID\_OBJECT$).

A abordagem demonstrada neste trabalho para a geração de casos de teste transforma cada propriedade identificada pelo ESBMC em uma assertiva no CUnit, de modo a obter-se como resultado final, uma nova instância do código analisado, contendo os casos de teste diretamente ligados as propriedades de segurança analisadas.

2.4 Contra-Exemplos

Um contra-exemplo é uma sequência de eventos que demonstra que uma dada propriedade não é segura no modelo [Baier & Katoen, 2008]. A verificação com a técnica *model checking*, como mencionado anteriormente, verifica se todas as propriedades desejadas estão seguras no modelo, então o sistema sob teste pode ser considerado seguro; de outro modo um contra-exemplo é gerado demonstrando o porque da propriedade não está segura no modelo. A partir do contra-exemplo, o usuário tem informações

[9] Nota-se que a linguagem ANSI-C oferece dois operadores diferenciados o $*p$ e $p[i]$, onde p representa um ponteiro (ou *array*) e i representa um índice inteiro, ambos são atendidos.

que lhe permitem: (i) analisar a falha; (ii) entender as origens do erro (causa raiz do erro); e (iii) corrigir as especificações ou o modelo.

Os modelos analisados pelo *model checking* são representados por um sistema de transições (ST) que é definido por Clarke et al. [2003], como uma 3-tupla $ST = (S, S_0, E)$ com um conjunto de estados S, um conjunto inicial $S_0 \subset S$, e um conjunto de transições $E \subset S \times S$. Baseado nesta definição de sistema de transições, podemos definir mais formalmente um contra-exemplo, como um caminho $\sigma = (s_0, s_1, ..., s_m)$ de $ST = (S, S_0, E)$, onde um conjunto $B \subset S$ que especifica um conjunto de *maus estados* tais que $S_0 \cap B = \emptyset$, logo $s_m \in B$, é chamado de um contra-exemplo de TS com relação a propriedade de segurança $TS \models \mathbf{AG}\neg B$.

Segundo Clarke et al. [2003], ainda podemos definir que dado um sistema de transição concreto C, um sistema de transição abstrata A, e um contra-exemplo σ em C, podemos dizer que $\hat{\sigma} = \{\hat{s}_0, \hat{s}_1, \hat{s}_2, ..., \hat{s}_m\}$ é correspondente ao contra-exemplo abstrato do sistema abstrato de A, se $\hat{s}_i = \alpha(\hat{s}_i)$ assegura para todo $i \in \{0, ..., m\}$. Deste modo dado um contra-exemplo de $\hat{\sigma}$ de A, σ é chamado de um contra-exemplo concreto correspondente, se $\hat{s}_i = \alpha(\hat{s}_i)$ e $(s_i, s_{i+1}) \in E$. Agora se o contra-exemplo de $\hat{\sigma}$ de A não tem um contra-exemplo concreto correspondente para C, então $\hat{\sigma}$ é denominado como um falso contra-exemplo.

Segundo Riacs et al. [2002], a explicação de um erro é especialmente importante no contexto de *model checking*. Por exemplo, quando a verificação de modelo está sendo feita sob a orientação dos projetistas e desenvolvedores do programa, esta verificação mostrará a maioria dos erros menos sutis, mas quaisquer erros remanescentes são suscetíveis a ter um alto grau de complexidade (desde a descoberta da falha rara, mas catastrófica em certo sentido), logo a explicação de um erro servirá de base importante para a correção do programa.

Infelizmente, os *bounded model checkers* freqüentemente produzem contra-exemplos que são tanto grande ou difíceis de serem entendidos devido aos valores escolhidos pelos *SAT/SMT solvers*. Como resultado, os valores que são escolhidos para as variáveis do programa dependem, fundamentalmente, da propriedade avaliada e da quantidade de variáveis envolvidas, o que torna o processo de coleta de informações para corrigir o programa uma tarefa complexa. Na Figura 2.3 é demonstrado um exemplo de um contra-exemplo gerado pelo ESBMC.

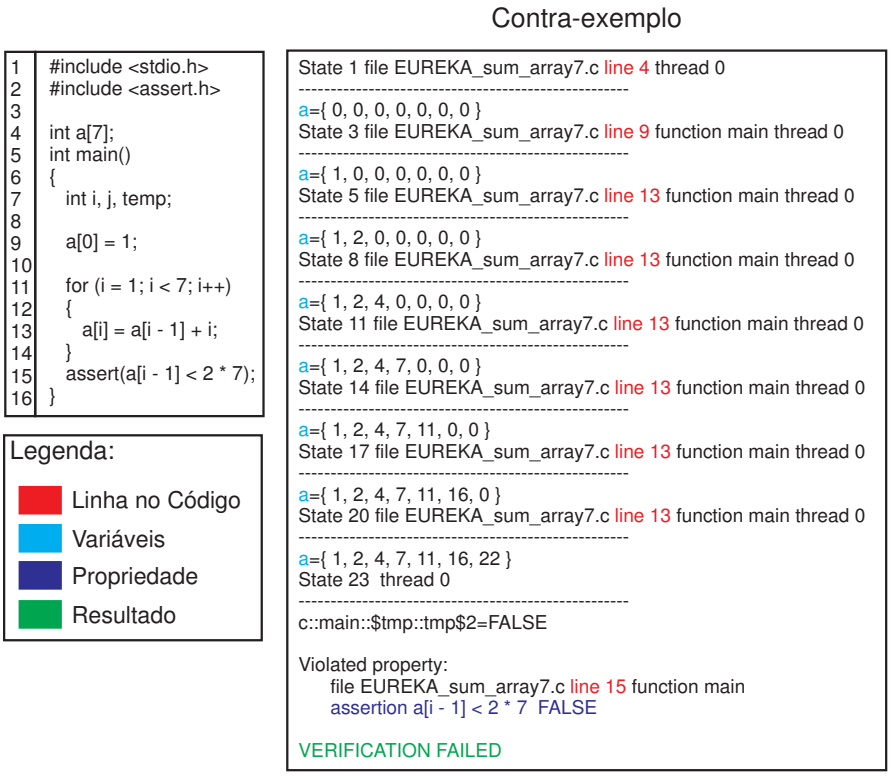


Figura 2.3. Exemplo de contra-exemplo gerado pelo ESBMC

2.5 Estratégias de Análise de Programas

Os sistemas de software e hardware têm tornado-se cada vez mais complexos e, com isso, os programadores precisam de mais ajuda do que nunca. Deste modo, a análise destes sistemas deve ser aprofundada ao ponto de abstrair e análisar as especificações requeridas para estes, de modo a identificar-se possíveis falhas.

Segundo Nethercote [2004], a análise de programas pode ser categorizada em quatro grupos, os dois primeiros são análise estática e análise dinâmica:

- (i) **Análise estática** que envolve a análise de código fonte de um programa ou código de máquina, sem executá-lo. Muitas ferramentas realizam análise estática, em particular compiladores; exemplos de análise estática utilizada por compiladores incluem a análise de corretude, tais como a verificação de tipos, e análise para otimização. As ferramentas que realizam a análise estática somente necessitam ler o programa de modo a análisa-lo [Nethercote, 2004].

- (ii) **Análise Dinâmica** que segundo Ball [1999], examina a execução do programa para derivar as propriedades seguras para uma ou mais execuções, de tal modo que se pode detectar a violação das propriedades. Muitas ferramentas

utilizam análise dinâmica, por exemplo, *profilers*, *model checkers* e *execution visualisers* [Nethercote, 2004]. As ferramentas que executam a análise dinâmica deve instrumentar [10] o programa com análises de código.

Estas duas abordagens são complementares, segundo Ernst [2003], análise estática pode ser uma boa opção, se considerarmos que ela executa todos os caminhos no programa, quando análise dinâmica somente considera um único caminho de execução. Todavia, a análise dinâmica é tipicamente mais precisa que a análise estática devida a ela trabalhar com valores reais em tempo de execução [Desoli et al., 2002].

A análise dos programas também podem ser categorizadas em outros dois grupos, de acordo com o tipo de código a ser análisado.

1. **Análise do código fonte**, segundo Nethercote [2004], envolve a análise de programas no nível de código fonte. Muitas ferramentas executam a análise de código fonte, os compiladores são um bom exemplo. Esta categoria de análise de código fonte do programa é realizada em representações que são derivados diretamente do código fonte, tal como gráficos de controle de fluxo. Análise do código fonte são geralmente feitas em termos de estruturas da linguagem de programação, tais como, funções, declarações, expressões e variáveis.

2. **Análise Binária**, segundo Nethercote [2004], envolve a análise de programas no nível de código de máquina, armazenado tanto como código objeto (*pre-linking*) ou código executável (*post-linking*). Normalmente a análise binária é feita em termos nas propriedades de código de máquina, tais como procedimentos, instruções, registros e em posições de mémoria.

Estas outras duas catégorias também são complementares, a principal diferença entre elas é que a análise de código fonte é independente de plataforma (arquitetura e sistema operacional), mas dependente de uma linguagem específica, enquanto a análise binária é independente de linguagem, mas dependente de plataforma [Nethercote, 2004]. Uma das principais vantagens da análise de código fonte é que ele tem acesso ao alto nível da informação, no qual pode ser mais poderoso, se comparado a análise binária que acessa ao baixo nível (tais como os resultados das posições de mémoria). Este trabalho foca principalmente na análise código fonte, visando a manipulação do código, para se efetuar uma análise mais aprofundada do mesmo, assim coletando e gerando novas informações sobre o mesmo.

[10] O verbo instrumentar é altamente utilizado na literatura para se referir a ação de adicionar código extra ao programa.

Capítulo 3

Trabalhos Correlatos

Neste Capítulo serão apresentados e discutidos os principais trabalhos existentes na literatura que se relacionam com os tópicos, que formam a base para o desenvolvimento deste trabalho, com ênfase em: geração de casos de teste, contra-exemplos de model checkers e depuração e análise de código.

3.1 Geração de Casos de Teste

Na literatura existem diversos métodos e ferramentas para a geração de casos de teste. O GAteL [Marre & Arnould, 2000] é um método que tem como base um SUT (*System Under Test*) ou uma completa especificação do SUT, uma descrição do ambiente, um objetivo do teste, quais tipos de propriedades verificar (*safety* ou *liveness*), a fim de gerar os casos de teste. Todos os elementos devem ser fornecidos tendo como base a linguagem Lustre [Caspi et al., 1987]. Outros métodos e ferramentas que contribuem neste ambiente de geração de casos de teste são o AsmL [Barnett et al., 2003], TorX [Tretmans & Brinksma, 2003], TestComposer e AutoLink [Schmitt et al., 2000] que são fortemente ligadas a um domínio específico ou uma linguagem.

Segundo Barnett et al. [2003], basicamente o AsmL (Abstract State Machine Language) é uma linguagem de modelagem executável que está plenamente integrado ao framework `.NET` e ferramentas de desenvolvimento da Microsoft. O AsmL suporta meta-modelagem baseada em reflexão computacional, que permite uma exploração sistemática não-deterministica no modelo. O TorX integra a geração automática de teste, execução de teste e análise de prova. TorX é baseado na teoria bem definida, ou seja, a teoria de teste **ioco**, que tem suas origens na teoria dos testes e recusa de equivalência para sistemas de transição[Tretmans & Brinksma, 2003]. Segundo Schmitt et al. [2000], o Autolink e TestComposer são totalmente integrados em seus correspondentes

ambientes de desenvolvimento. O TestComposer é construído em cima do simulador ObjectGeode[Kerbrat et al., 1999]; Autolink é parte do validador Tau[1]. O simulador, assim como o validador são usados para localizar erros e inconsistências dinâmicas nas especificações SDL (*Specification and Description Language*).

Este trabalho visa complementar com este conjunto de soluções já existentes, tendo como domínio específico a linguagem de programação C no qual, na maioria das vezes, não é explorada ou contemplada. Deste modo a metodologia proposta utiliza o próprio código como base para a geração dos casos de teste, ou seja, sem a necessidade de uma modelagem extra.

No trabalho de Cadar & Engler [2005] é apresentado uma técnica que usa o código para gerar automaticamente os seus próprios casos de teste em tempo de execução pela utilização da combinação de execução simbólica e concreta (ou seja, regular). A sua principal contribuição é a análise de observação no código, para automaticamente gerar os seus próprios e potênciais casos de teste. Instanciando e executando o código com entradas concretas, construídas manualmente, esta técnica executa a sua própria entrada simbólica. Esta observação diz quais são os valores válidos (ou intervalos de valores) para a entrada no código, e assim gerar os casos de teste pela solução dessas restrições para valores concretos. Estes testes são chamados de *execution generated testing* (EGT).

O trabalho aqui apresentado, assim como o trabalho de Cadar & Engler [2005], visa explorar o código detectando possíveis propriedades a serem testadas, utilizando técnicas e métodos de análise e verificação de código. Todavia, fazendo uma análise de comparação com o trabalho de Cadar & Engler [2005], não visamos a utilização de entradas simbólicas como é tratado nesta aboradagem, pois não é levado em conta os fatores de limitação do solucionador de satisfabilidade e mémoria utilizada. Para o trabalho desenvolvido utilizamos *Bounded Model Checkers* para identificar as propriedades e a partir destas propriedades gerar os casos de testes.

3.2 Contra-exemplos de *Model Checkers*

Existem diferentes aspectos para entender ou analisar um contra-exemplo. Segundo Beer et al. [2009], nos últimos anos, o processo de encontrar um erro na fonte de origem tem atraído uma atenção significativa. Muitos trabalhos têm abordado este problema tais como, Copty et al. [2001], Jin et al. [2002], Riacs et al. [2002], Dong et al. [2003], Ball et al. [2003], Groce [2004], Groce & Kroening [2005], Chechik & Gurfinkel [2005],

[1]http://www.itm.mu-luebeck.de/

Shen et al. [2005], C.Wang et al. [2006], Griesmayer & and [2007], Staber & Bloem [2007], Sülflow et al. [2008], abordando a questão de encontrar a causa central da falha no modelo, e propondo meios automáticos de extrair mais informações sobre o modelo, facilitando o processo de depuração.

Os contra-exemplo gerados pelos *model checkers* geralmente possuem um tamanho significativo, o que torna o processo de análise do mesmo uma tarefa muitas vezes complexa. O trabalho de Groce & Kroening [2005] apresenta uma técnica que pode ser usada para minimizar o tamanho do contra-exemplo gerados por BMC, mas centra-se na minimização semântica, no que diz respeito ao sistema de tipos da linguagem (ANSI C). Basicamente, essa abordagem minimiza os valores das variáveis do contra-exemplo. Com relação ao trabalho aqui apresentado visamos contribuir, assim como em Groce & Kroening [2005], no processo de tratamento de contra-exemplo ao usuário final, em nosso caso já apresentando os dados abstraídos do mesmo.

No Trabalho de Beer et al. [2009], é abordado o problema da análise de contra-exemplo de um determinado erro que ele representa. Usando a noção de causalidade, introduzido por Halpern & Pearl [2000], objetiva explicar o erro ocorrido, definindo um conjunto de causas para a falha da especificação. Estas causas são marcadas como "ponto vermelhos" e apresentados ao usuário como uma explicação visual da falha. Tendo em vista o trabalho de Beer et al. [2009], visamos complementar nesta área de entendimento do erro, todavia por meio da confirmação do erro, provendo ao usuário final o resultado do erro diagnosticado, onde o usuário, tendo como base a consequência do erro e as variáveis envolvidas, o mesmo seja capaz de identificar soluções para evitar o erro.

Como efetuado em [Groce & Kroening, 2005; Beer et al., 2009], visamos também contribuir com a análise de contra-exemplos gerados na verificação de código pelos BMC, de modo que se o BMC gerar um falso contra-exemplo, possamos validar este através da execução do código com os valores nele apresentados.

3.3 Depuração e Análise de Código

Existem diversos meios e métodos existentes na literatura para se efetuar a análise e depuração de códigos com o objetivo de identificar e comprovar erros. Nesta seção estaremos abordando itens como, instanciação, instrumentação e análise dinâmica de código.

No trabalho de Calimeri et al. [2008] é implementado uma estratégia para a instanciação paralela, permitindo melhorar tanto o desempenho e escalabilidade de sistemas ASP[2] (*Answer Set Programming*), explorando o poder dos computadores com múltiplos processadores. A técnica explora algumas propriedades estruturais da entrada do programa de modo a detectar subprogramas de `P` (dado qualquer programa `P`) que podem ser avaliados em paralelo, minimizando a utilização dos mecanismos de controle de concorrência nas principais estruturas, e assim minimizar o "overhead paralelo". Deste modo, assim como no trabalho [Calimeri et al., 2008] a instanciação foi utilizada como um meio intermediário para um procedimento específico de análise do código, deste mesmo modo visamos utilizar a instanciação de dados para a comprovação de erros.

Groce & Joshi [2008] demonstram um combinação de baixa sobrecarga de instrumentação de código, *model checking*, e análise dinâmica. Utilizando o *framework* CIL [3] (*C Intermediate Language*) para instrumentar o código a fim de permitir a verificação de propriedades com o *model checker* SPIN [4], redefinindo a simulação do sistema, e usando as informações de cobertura local para guiar a busca do *model checking*. Outros métodos, tais como, ATOM [Srivastava & Eustace, 1994] e Pin [Luk et al., 2005] são ferramentas para instrumentação de código também, mas eles são usados principalmente para análise de desempenho e coleta de estatísticas sobre os programas. Assim como demonstrado no trabalho de Groce & Joshi [2008] objetivamos a combinação de análise dinâmica, análise estática e *model checking*.

O trabalho de Ji et al. [2008] apresenta um depurador (*debugger*) de *software*, para um micro-processador de 32-bit, baseado no *debugger* do GNU (GDB), onde este é utilizado para encontrar erros nos programas. Com relação ao método proposto em nosso trabalho para a comprovação de erros, nós utilizamos a verificação efetuada com o *model checker*, para encontrar erros, que é um importante aspecto neste método proposto, pois o *model checker* testa o sistema exaustivamente (explorando todos os estados) para verificar que uma dada propriedade é parte do modelo.

[2] ASP é uma abordagem de programação declarativa para propostas na área de raciocínio e lógica de programação não monotônica.

[3] http://cil.sourceforge.net/

[4] http://spinroot.com/spin/whatispin.html

Adicionalmente, BMCs executam o código simbolicamente, ou seja, ele não somente testa o programa com os valores fixos de entrada, mas também cria um modelo matemático do programa Baier & Katoen [2008]. Outros métodos, tais como os *debuggers* Matloff & Salzman [2008], somente implementam a execução dos caminhos que foram definidos de acordo com as entradas das variáveis. Assim, um *debugger* não irá testar exaustivamente os espaços do estado do código analisado.

Taghdiri [2004] apresenta um método de análise estática de programa para verificar as propriedades estruturais do código proposto. O método basicamente funciona da seguinte forma, uma abstração inicial do código é computada para obter a aproximação do efeito da chamada das funções. Quando o algoritmo proposto finaliza, os contra-exemplos gerados são checados afim de verificar se não são falsos, confirmando a verificação efetuada, todavia a ausência de um contra-exemplo não constitui uma prova. Deste modo, nossa contribuição é colocar junto as abordagens de *model checking* e análise de programa, ou seja, usar a verificação com BMC, utilizando os dados contidos no contra-exemplo gerado para instânciar o código analisado e por meio da execução do mesmo comprovar o erro.

Capítulo 4

Verificação e Comprovação de Erros em Código C

Este Capítulo descreve o método proposto por este trabalho, que visa uma metodologia para verificação formal de propriedades de segurança e comprovação de erros em códigos escritos na linguagem C. A proposta é dividida em duas vertentes: (i) Explorando Propriedades de Segurança em BMC para a Geração de Casos de Teste em Programas em C e (ii) Instanciação de Programas em C pela Utilização de Contra-Exemplos de BMC. Estas etapas servirão de base para a solução do problema definido.

4.1 Explorando Propriedades de Segurança em BMC para a Geração de Casos de Teste em Programas em C

Esta Seção descreve as principais etapas do método proposto, denominado de **FORTES** (*FORmal unit TESt generation*), que visa explorar as propriedades de segurança geradas pelo ESBMC, objetivando uma verificação complementar à efetuada pelo ESBMC. O método consiste das seguintes etapas:

- *(i)* Identificação das propriedades de segurança;
- *(ii)* Coleta de informações das propriedades de segurança;
- *(iii)* Inclusão de assertivas;
- *(iv)* Implementação de teste unitário com Framework CUnit; e

- *(v)* Execução de testes unitários com Framework CUnit.

A Figura 4.1 demonstra uma visão geral do método FORTES. De modo a explicar as principais etapas do método proposto nós utilizamos o código demonstrado na Figura 4.2.

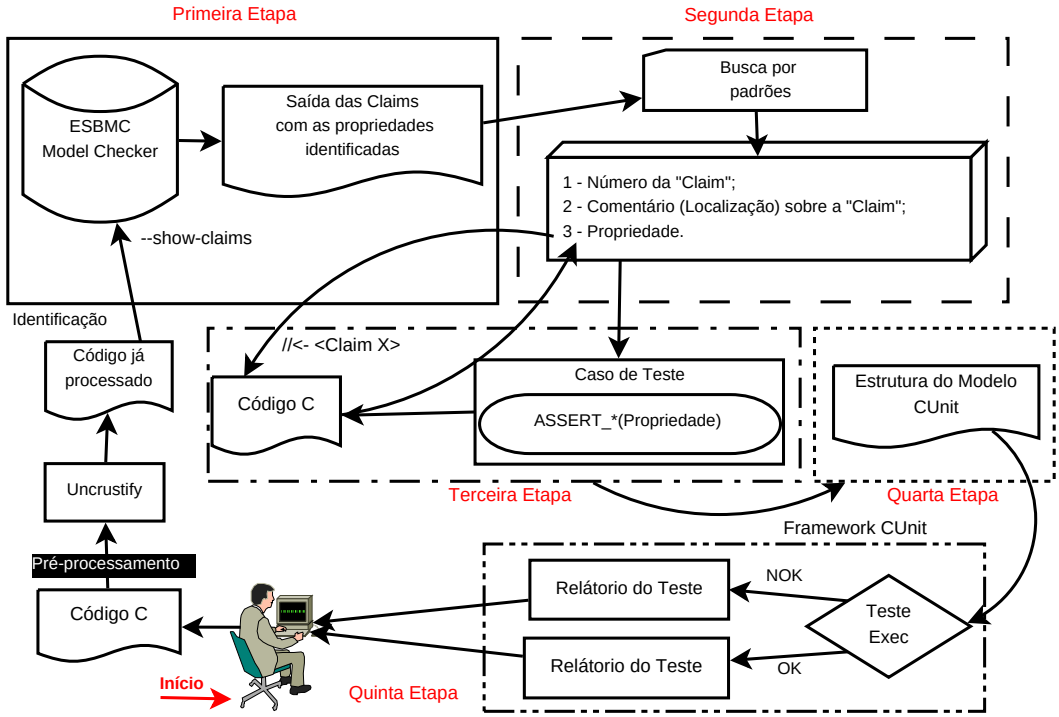


Figura 4.1. Fluxo da estrutura do método proposto.

```
1 #include <stdio.h>
2 int a[5];
3 int b[6];
4
5 int main(){
6  int i, j, temp;
7  a[0]=1;
8  for(i=1; i<5; i++) {
9     a[i]= a[i-1] + i;
10    b[0] = 1;
11    temp = a[i]*(i+1);
12    for(j=1;j<temp;j++) {
13     b[j] = b[i-1]+(temp*2);
14    }
15  }
16 }
```

Figura 4.2. Código C sum_array.c, para a descrição das etapas do método proposto.

4.1.1 Primeira Etapa: Identificação das propriedades de segurança

Nesta primeira etapa o código a ser analisado é pré-processado usando a ferramenta *UNCRUSTIFY*[1] que irá processar o código, de modo a definir um padrão de formatação, envolvendo itens como: indentação, delimitadores de bloco e um comando por linha. Como dito anteriormente, utilizamos o ESBMC para efetuar a identificação das propriedades de segurança. O ESBMC recebe como entrada o programa C que será analisado e usa a opção `--show-claims`, que mostra as propriedades de segurança que o ESBMC identifica como aquelas que podem vir a ser violadas. No contexto de *bounded model checking*, uma `claim` é o mesmo que uma propriedade. Exemplos de propriedades de segurança são: *buffer overflow*, divisão por zero, *overflow* aritmético e assim por diante.

A necessidade de verificação destas *claims* tem como principal argumento que elas podem ser violada em alguma execução dos caminhos possíveis do programa. É interessante notar que as *claims* geradas automaticamente não necessariamente correspondem a *bugs* (erros ou violação das propriedades), mas elas são apenas faltas em potencial. Se uma dessas *claims* correspondem a um *bug*, este *bug* deve ser determinado por uma análise mais aprofundada. Esta análise consiste da tradução desta

[1]Disponível em http://uncrustify.sourceforge.net

propriedade em um caso de teste e da respectiva execução deste. Cada propriedade de segurança (ou *claim*) gerada pelo ESBMC contém as seguintes informações: a propriedade identificada, localização e a assertiva a ser checada.

A Figura 4.3 demonstra um exemplo de uma *claim* gerada automaticamente. Neste exemplo, a *claim 2* apresenta um potencial *upper bound* do *array* `a` no código `sum_array.c` (Figura 4.2) na linha 9 na função *main*. Todas as *claims* identificadas pelo ESBMC no código C analisado são armazenadas para tratamento posterior nas próximas etapas do método.

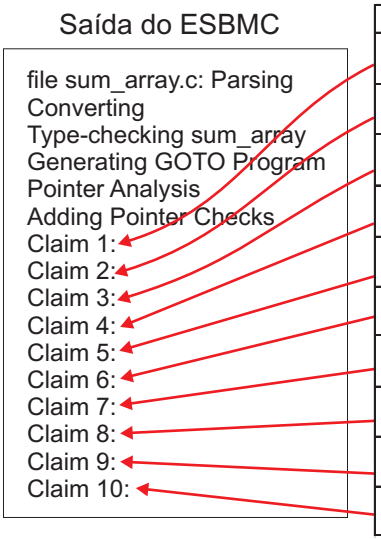
```
herbert@nbh /media/$ esbmc sum_array.c --show-claims
file sum_array.c: Parsing
Converting
Type-checking sum_array
Generating GOTO Program
Pointer Analysis
Adding Pointer Checks
Claim 2:
  file sum_array.c line 9 function main
  array `a' upper bound
  i < 5
```

Figura 4.3. Primeira etapa do método proposto.

4.1.2 Segunda Etapa: Coleta de Informações das Propriedades de Segurança

A segunda etapa do método proposto faz um tratamento no resultado da etapa 1 para simplesmente coletar, e deixar mais acessível, informações importantes e necessárias para as etapas seguintes. As informações a serem coletadas são: (i) a *claim*, (ii) os comentários sobre a *claim*, (iii) o número da linha no código onde a *claim* ocorreu e (iv) a propriedade identificada pela *claim*.

O resultado da aplicação da segunda etapa no código exemplo da Figura 4.2 pode ser visto na Figura 4.4, onde apresenta-se diversas informações sobre as claims (propriedades) encontradas.



Claims	Comentários	Linha do Código	Propriedade
Claim 1	file sum_array.c line 9 function main array `a' lower bound	9	i >= 0
Claim 2	file sum_array.c line 9 function main array `a' upper bound	9	i < 5
Claim 3	file sum_array.c line 9 function main array `a' lower bound	9	-1 + i >= 0
Claim 4	file sum_array.c line 9 function main array `a' upper bound	9	-1 + i < 5
Claim 5	file sum_array.c line 11 function main array `a' lower bound	11	i >= 0
Claim 6	file sum_array.c line 11 function main array `a' upper bound	11	i < 5
Claim 7	file sum_array.c line 13 function main array `b' lower bound	13	j >= 0
Claim 8	file sum_array.c line 13 function main array `b' upper bound	13	j < 6
Claim 9	file sum_array.c line 13 function main array `b' lower bound	13	-1 + i >= 0
Claim 10	file sum_array.c line 13 function main array `b' upper bound	13	-1 + i < 6

Figura 4.4. Aplicação da segunda etapa.

4.1.3 Terceira Etapa: Inclusão de Assertivas

Esta etapa visa criar os casos de teste, baseados em assertivas, os quais serão incluidos no código com a sua respectiva propriedade de segurança gerada pelo ESBMC. Esta etapa adiciona uma assertiva contendo a (talvez violada) propriedade de segurança para cada *claim* identificada na segunda etapa.

Usando o resultado da segunda etapa, esta terceira etapa identifica a linha de código de cada propriedade identificada, para adicionar uma assertiva contendo a respectiva propriedade de segurança em uma linha anterior a linha no código identificada com a propriedade a ser verificada. A Figura 4.5 apresenta um trecho da aplicação da terceira etapa no código C da Figura 4.2.

```
1 //Claim 1: file sum_array.c line 6 //array 'a' lower bound
2 CU_ASSERT(i >= 0);
3 //Claim 2: file sum_array.c line 6 //array 'a' upper bound
4 CU_ASSERT(i < 5);
5 a[i]= a[i-1] + i ; b[0] = 1 ;//<- <Claim 1> <Claim 2>
```

Figura 4.5. Código C com as assertivas inseridas na terceira etapa.

4.1.4 Quarta Etapa: Implementação de Teste Unitário com Framework CUnit

Esta etapa tem como objetivo gerar a entrada adaptada para a estrutura do *framework CUnit*. Esta geração é feita usando um *template* que contém os seguintes itens: (i) os *includes* específicos para o CUnit e os do código C analisado; (ii) As funções de configuração do CUnit; (iii) As funções que contêm os casos de teste que serão testados; e (iv) Uma nova função *main* que executará o CUnit. A Figura 4.6 mostra os itens do template e seus respectivos códigos.

Os *includes* do CUnit são obtidos diretamente do arquivo de *template*. Os *includes* do arquivo de código C são copiados diretamente do código fonte C analisado. As funções de configuração do CUnit são obtidas diretamente do arquivo de *template*. As funções que serão testadas são preenchidas pela função *main* do código fonte C alterando o nome da função *"main"* para *"testClaims"*. A nova função *main* e seu respectivo conteúdo é obtido do arquivo de *template*. O resultado é um novo arquivo de código que está pronto a ser testado e executado pelo *framework* CUnit.

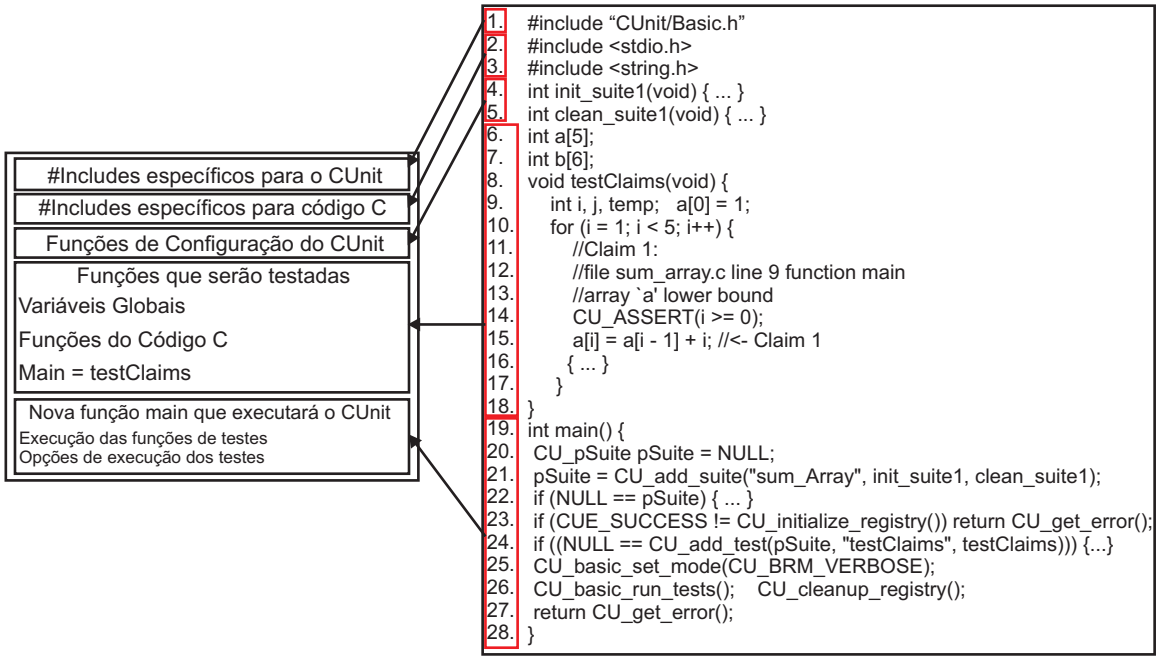


Figura 4.6. Código de Teste para a execução do testes unitários utilizando o CUnit.

4.1.5 Quinta Etapa: Execução de Testes Unitários com Framework CUnit

Nesta última etapa, o código obtido da quarta etapa é submetido a execução com o CUnit. Neste caso, o CUnit executa os testes no código que contém os casos de teste (validando cada assertiva) que foram gerados durante a aplicação do método, pela abstração das propriedades de segurança. Basicamente, os casos de teste são analisados durante a execução do código, onde cada um pode ser aprovado ou falhar. Cada falha do teste é relatada pelo *framework* ao final da execução.

A Figura 4.7 demonstra o resultado da aplicação do método proposto no código C mostrado anteriormente na Figura 4.2. Como podemos ver, houve um teste onde uma assertiva foi executada 404 vezes durante a execução do código e foi detectado que por 77 vezes ela falhou, assim resultando na violação da propriedade.

```
Suite: sum_Array

  Test: testClaims ... FAILED
    sum_Array.c: 86 - j < 6

--Run Summary:   Type      Total     Ran     Passed     Failed
                 suites        1       1        n/a          0
                 tests         1       1          0          1
                 asserts     404     404        327         77
```

Figura 4.7. Saída resultante da execução do Cunit.

4.2 Instanciação de Programas em C pela Utilização de Contra-Exemplos de BMC

Esta Seção descreve as principais etapas do método proposto, denominado de **EZProofC**, para explorar os contra-exemplos gerados pelo ESBMC para comprovação de erros pela utilização da instanciação de código. O método proposto consiste das seguintes etapas:

- (i) Pré-processamento de código;
- (ii) Verificação formal com ESBMC;
- (iii) Instanciação de código; e
- (iv) Execução de código e confirmação de erros.

A Figura 4.8 apresenta uma visão geral do método EZProofC. De modo a explicar as principais etapas do método proposto, utilizaremos o código *tT-flag_arr_two_loops_bad.c* do *benchmark* Verisec [2] do programa Sendmail [3] que é um servidor padrão de e-mail (SMTP) Unix.

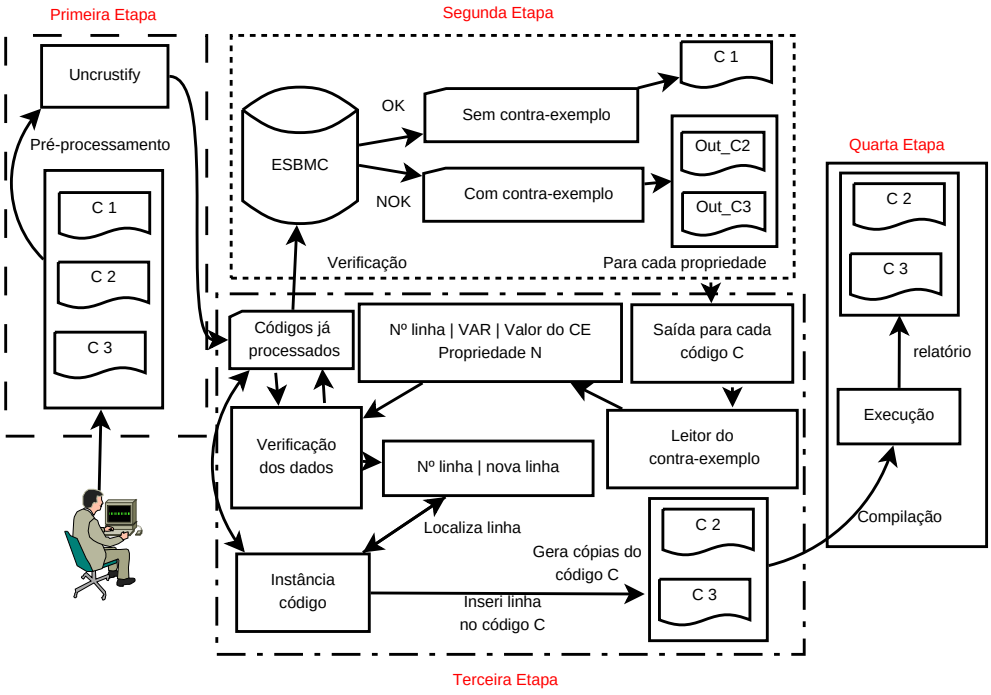


Figura 4.8. Estrutura do fluxo do método proposto.

[2] http://se.cs.toronto.edu/index.php/Verisec_Suite
[3] http://www.sendmail.org

4.2.1 Primeira Etapa: Pré-processamento de Código

Nesta primeira etapa o código analisado é pré-processado usando a ferramenta *UNCRUSTIFY*[4] que irá processar o código, de modo a definir um padrão de formatação, envolvendo itens como: indentação, delimitadores de bloco, um comando por linha, definição de estruturas e outros aspectos de formatação, como apresentado no código exemplo da Figura 4.9. Esta etapa de pré-processamento permite uma melhor definição das estruturas contida no código, facilitando a sua manipulação para a aplicação das próximas etapas.

```
1  /*accumulate last(int) from in (char[])*/
2  c = in[idx_in];
3  if (c == '-')
4  {
5   i=0;
6   idx_in++;
7   c = in[idx_in];
8   while (('0' <= c) && (c <= '9'))
9   {
10     j = c - '0';
11     i = i * 10 + j;
12     idx_in++;
13     c = in[idx_in];
14    }
15 }
```

Figura 4.9. C code already pre-processed.

4.2.2 Segunda Etapa: Verificação Formal com ESBMC

Nesta segunda etapa, o código pré-processado é verificado usando a ferramenta ESBMC. Esta verificação é realizada principalmente em dois modos, podendo-se optar entre eles. O primeiro modo efetua a identificação da propriedade (neste contexto chamada de *claim*) do código. Esta identificação é efetuada para que se possa verificar cada propriedade de forma individual. Esta é realizada pela execução do ESBMC passando como parâmetros o código C com a opção `--show-claims`, e depois é feita a verificação da propriedade identificada, usando o ESBMC com as opções `--unwind <n> --claim <x> <file.c>`, onde `<n>` significa o número de vezes que o código será desenrolado, e `<x>` é o número da respectiva *claim* que será verificada. Esta opção

[4]Disponível em http://uncrustify.sourceforge.net

fornece uma verificação mais ampla do código, delimitada apenas pela propriedade especificada, todavia ela não é muito usual para código (programas) grandes, devido a possibilidade de grande utilização de memória.

No segundo modo, é efetuado uma verificação modular no código, com base nas funções do código, esta é realizada pela execução do ESBMC passando como parâmetros o código C com as opções `--function <f> --unwind <n>`, onde `<f>` é a função a ser verificada. Nesta opção a verificação é efetuada no código para uma função específica onde, neste caso, o escopo é reduzido. Esta opção é usual para códigos grandes. É importante ressaltar que em ambos os modos é permitido a inclusão de outras opções disponíveis pelo ESBMC.

O resultado da verificação pode ser classificado de suas maneiras: (i) o código foi verificado é não há contra-exemplo, ou seja, a propriedade foi verificada mas não houve erro; e (ii) o código foi verificado e existe um contra-exemplo, ou seja, a propriedade foi violada. A Figura 4.10 mostra um exemplo de violação de propriedades. A verificação deste código foi feito com base no primeiro modo. Neste segundo caso (quando existe um contra-exemplo), o método proposto armazena este contra-exemplo que será utilizado nas próximas etapas.

```
$ esbmc --show-claims ccode.pre/pre_tTflag_arr_two_loops_bad.c

Claim 10:
  file ccode.pre/pre_tTflag_arr_two_loops_bad.c line 43 function main
  array `in' upper bound
  idx_in < 15

$ esbmc --unwind 15 --claim 10 ccode.pre/pre_tTflag_arr_two_loops_bad.c

Counterexample:

State 96 file ccode.pre/pre_tTflag_arr_two_loops_bad.c line 40 function main thread 0
------------------------------------------------------
 pre_tTflag_arr_two_loops_bad::main::1::j=3 (00000000000000000000000000000011)

State 97 file ccode.pre/pre_tTflag_arr_two_loops_bad.c line 41 function main thread 0
------------------------------------------------------
 pre_tTflag_arr_two_loops_bad::main::1::i=33 (00000000000000000000000000100001)

State 98 file ccode.pre/pre_tTflag_arr_two_loops_bad.c line 42 function main thread 0
------------------------------------------------------
 pre_tTflag_arr_two_loops_bad::main::1::idx_in=15 (00000000000000000000000000001111)

Violated property:
 file ccode.pre/pre_tTflag_arr_two_loops_bad.c line 43 function main
 array `in' upper bound
 idx_in < 15

VERIFICATION FAILED
```

Figura 4.10. Contra-exemplo do código verificado.

4.2.3 Terceira Etapa: Instanciação de Código

A terceira etapa visa a instanciação do código com os dados gerados pelo contra-exemplo do ESBMC, onde esta etapa é dividida em duas fases: (i) análise do contra-exemplo produzido na etapa 2; e (ii) a geração de um novo código instanciado com os dados do contra-exemplo gerado. Na primeira fase, os contra-exemplos produzidos na etapa 2 são analisados para a coleta de diversas informações, tais como: (1) as variáveis envolvidas na violação da propriedade verificada; (2) o número da linha correspondente a cada variável envolvida na violação da propriedade; e (3) o valor identificado no contra-exemplo para cada variável.

As informações coletadas do contra-exemplo serão checadas no código C analisado, para comparação e validação da informações abstraídas, de tal forma que a partir destas informações o método gere uma nova linha do código, onde para cada variável identificada no código, esta receberá o valor abstraído do contra-exemplo, como apresentado na Figura 4.11.

```
-----------------------------------------------------
File: pre_tTflag_arr_two_loops_bad.c
-----------------------------------------------------
Linha  | Variável | Valor
  40   |    j     |   6
Nova Linha -> j = 6;
  41   |    i     |   1191976950
Nova Linha -> i = 1191976950;
  42   | idx_in   |   15
Nova Linha -> idx_in = 15;
-----------------------------------------------------
```

Figura 4.11. Lista dos dados coletados do contra-exemplo.

A coleta de informações no contra-exemplo em situações específicas podem requerer uma abordagem a parte, por exemplo:

- (i) quando é identificado a violação de uma propriedade (um *bug*) e não há dados no contra-exemplo, isto pode ocorrer devido a aplicação de um função que efetua cortes no contra-exemplo gerado pelo ESBMC, deste modo algumas vezes (em particular código pequenos) é necessário usar a opção `--no-slice`, a qual não remove equações não utilizadas para gerar o contra-exemplo. Outro modo de diversificar os valores na verificação e consequentemente o resultado no contra-exemplo é inicializamos as variáveis do código com valores randômicos, ou seja, não determinísticos (por exemplo, `a[0]=nondet_int()`);

- (ii) Algumas estruturas na linguagem C como *Structs* e *arrays* serão demonstradas no contra-exemplo como listas, por exemplo, os *arrays* são demonstrados

através de uma lista com o seus respectivos valores para cada estado de sua execução, identificados no *trace* do contra-exemplo, como demonstrado na Figura 4.12, ou seja, para efetuar a comprovação do erro utilizando os dados destas estruturas, é necessário a manipulação das mesmas, de maneira a atribuir a cada posição o seu respectivo valor;

- (iii) Com relação a análise das propriedades para segurança de ponteiros e alocação de memória, o método proposto verifica: (1) se o ponteiro referência um objeto correto (representado por *SAME_OBJECT*); (2) se o ponteiro é `NULL` ou se referencia um objeto inválido (representado por *INVALID_POINTER*); (3) se o objeto é considerado um objeto dinâmico (representado por *IS_DYNAMIC_OBJECT*); e (4) se o argumento para liberação (*free*) é uma operação ainda válida sobre um objeto dinâmico (*VALID_OBJECT*). O objetivo desta análise é obter uma assertiva da propriedade identificada, onde o lugar para a inserção da assertiva é obtido do contra-exemplo, deste modo a assertiva é inserida uma linha antes da linha onde ocorreu a violação da propriedade.

```
State 17 file ccdoe.pre/EUREKA_sum_array7.c line 13
-------------------------------------------------
   a={ 1, 2, 4, 7, 11, 16, 0 }

State 20 file ccdoe.pre/EUREKA_sum_array7.c line 13
-------------------------------------------------
   a={ 1, 2, 4, 7, 11, 16, 22 }
```

Figura 4.12. Exemplo de um *array* em um contra-exemplo.

A segunda fase é baseada na geração de uma nova instância do código, neste momento já com o erro instanciado. O método somente efetua uma cópia do código original e substitui a atribuição feita a variável usando a linha do código e os valores identificados na fase 1 desta terceira etapa. É importante ressaltar que a verificação que ocorre na próxima etapa, para propriedades como `UPPER BOUND` ou `LOWER BOUND`, o método utiliza assertivas para comprovar o erro, estas assertivas contêm as propriedades identificadas no contra-exemplo gerado, as quais serão inseridas no código, no local onde foi identificado no contra-exemplo a violação da propriedade.

O resultado final da aplicação da segunda fase para cada propriedade examinada é um novo código C para a sua respectiva propriedade analisada, neste momento já com o código instanciado com os valores obtidos do contra-exemplo. O resultado da aplicação desta segunda fase é demonstrado na Figura 4.13.

```
1  /*accumulate last(int) from in (char[])*/
2  c =45 ;
3  if (c == '-')
4  {
5   i =0 ;
6   idx_in = 3 ;
7   c =56 ;
8   while (('0' <= c) && (c <= '9'))
9   {
10     j =6 ;
11     i =1191976950 ;
12     idx_in = 15 ;
13     assert(idx_in<15);
14     c =54 ;
15  }
16 }
```

Figura 4.13. C code already instantiated.

4.2.4 Quarta Etapa: Execução de Código e Confirmação de Erros

As novas instâncias dos códigos gerados na terceira etapa são compilados e executados um por um. Cada execução tem o seu resultado armazenado, assim como o resultado da compilação, *warnings*, de cada código.

A Tabela 4.1 demonstra o resulta da execução do código da Figura 4.9, comprovando o erro apontado e demonstrado pelo contra-exemplo do ESBMC, com base na aplicação das etapas do método proposto.

Tabela 4.1. Resultado da execução do código com método aplicado.

Módulo	Resultado da Execução
new_inst_tTflag_arr_two_loops_bad.c	44:main: Assertion 'idx_in<15' failed. Aborted

Capítulo 5

Resultados Experimentais

Neste Capítulo são apresentados os resultados experimentais da aplicação do método proposto no Capítulo 4. O objetivo da experimentação é a análise da aplicação prática da solução proposta. Dessa forma os métodos e técnicas já desenvolvidos neste trabalho foram verificados experimentalmente com a aplicação para verificação de *benchmarks* ANSI-C. Nossos experimentos foram conduzidos em um Intel Core 2 Duo CPU, 2Ghz, 3GB RAM com OS Linux.

5.1 Resultados Experimentais em Exploração de Propriedades de Segurança em BMC para a Geração de Casos de Teste em Programas em C

As etapas do método FORTES foram implementadas usando o ESBMC v.1.11, e o *framework* CUnit v2.1. A Tabela 5.1 detalha os *benchmarks* ANSI-C usados nestes experimentos. A primeira coluna, contém o número das aplicações; a segunda coluna descreve o nome do módulo composto pelo respectivo *benchmark*, programa em C, e a entrada (por exemplo, determinística ou não determinística); a terceira coluna demonstra o número de linhas do código (LOC); a quarta coluna provê o total do número de propriedades identificadas, o qual é equivalente ao número de casos de testes que foram criados; a quinta coluna apresenta o número de propriedades violadas utilizando o ESBMC; e por fim, a sexta coluna apresenta o número de propriedades violadas utilizando o método FORTES.

Tabela 5.1. Detalhes relacionados aos *benchmarks* dos códigos C usados.

Nº	Módulo	LOC	Identificado	#V ESBMC	#V FORTES
1	EUREKA_bf20_det.c	49	33	0	0
2	EUREKA_Prim_4_det.c	78	30	0	0
3	SNU_bs_nondet.c	120	7	0	0
4	SNU_crc_det.c	125	15	0	0
5	SNU_insertsort_nondet.c	94	14	6	0
6	SNU_qurt_det.c	164	6	0	0
7	SNU_qsort-exam_det.c	134	49	-	0
8	SNU_select_det.c	122	39	6	6
9	WCET_cnt_nondet.c	139	16	0	0
10	Oximeter_log_det.c	177	4	2	2

Os *benchmarks* utilizados foram EUREKA[1], SNU[2], WCET[3], e um fragmento de código extraído de um programa para um dispositivo de oxímetro de pulso [Cordeiro et al., 2009a]. Os resultados da aplicação do método FORTES estão disponíveis em https://sites.google.com/site/fortesmethod/.

Como apresentado anteriormente, o método FORTES utiliza o ESBMC para encontrar as propriedades (*claims*) que podem ser violadas. Quando o ESBMC é usado com a opção `--show-claims` ele somente identifica as possíveis violações de propriedades (neste contexto chamadas de claims). Neste caso, as propriedades identificadas pelo ESBMC podem ou não corresponder a *bugs* no programa. De qualquer forma, todas as propriedades identificadas pelo ESBMC foram implementadas pelo método proposto usando o CUnit. Entretanto, com o objetivo de validar e verificar a usabilidade prática do método proposto, utilizamos o ESBMC também com a opção de checar se a propriedade identificada foi realmente violada. Para tanto, a opção do ESBMC utilizada foi `--clam <number>`, onde `<number>` é o número da *claim* que será verificada. Portanto, em outras palavras, nós comparamos os resultados da aplicação do método, que usa CUnit, com a verificação de propriedades do próprio ESBMC.

Na Tabela 5.1, nas linhas de 1 a 4, 6 e 9 os respectivos códigos não tiveram nenhuma propriedade violada tanto com a aplicação do método FORTES quanto com a verificação efetuada somente com o ESBMC. Nas linhas 8 e 10 foram identificadas a violação de 6 e 2 propriedades, respectivamente identificadas pelo método FORTES e confirmadas pelo ESBMC. Houve dentre estes, dois casos especiais, (i) na linha

[1]http://www.ai-lab.it/eureka/bmc.html
[2]http://archi.snu.ac.kr/realtime/benchmark
[3]http://www.mrtc.mdh.se/projects/wcet/benchmarks.html

7, o método não identificou nenhum erro, todavia completou a verificação, porém utilizando o ESBMC ele tanto ultrapassou o limite de tempo estipulado (uma hora) quanto durante a verificação houve falta de memória; (ii) na linha 5, o método FORTES não identificou nenhum erro, porém somente com ESBMC foi identificado a violação de 6 propriedades. Este segundo caso em particular será abordado como trabalho futuro, onde uma possível solução para este caso seria a utilização de técnicas como, por exemplo, a instrumentação de código ou execução simbólica. A idéia é que estas venham explorar os casos de testes com uma quantidade mais abrangente de dados,além dos que são obtidos no próprio código, explorando assim um número maior de execuções do código e por sua vez dos casos de teste.

O método FORTES, proposto para identificação e verificação de erros, considerando as propriedades de segurança, tem demonstrado que esta não é uma tarefa trivial de ser implementada sem a utilização de uma metodologia sistemática, principalmente por que somente a identificação das propriedades nem sempre é suficiente para garantir que uma determinada propriedade seja violada.

De modo a detalhar ainda mais este ponto nos utilizamos o código "Oximeter_log_det.c" da implementação do dispositivo de oxímetro de pulso [4] [Cordeiro et al., 2009a] demonstrado na linha 10 da Tabela 5.1. Com o objetivo de demonstrar a dificuldade para a análise da verificação e identificação do erro. Neste exemplo, podemos notar que o ESBMC identificou quatro propriedades, mas nós iremos focar em um fragmento de código específico como demonstrado na Figura 5.1, onde o ESBMC identificou as seguintes propriedades: "`next` $<$ `6400`" na linha 2, e "`(unsigned int)buffer_size != 0`" na linha 3.

A identificação destas duas propriedades nos revela que esse fragmento de código tem propriedades que podem ser violadas, e que resultaram em problemas de *buffer overflow* e divisão por zero. Estes problemas podem ocorrer, devido ao fato que as variáveis "`buffer_size`" e "`next`" são instanciadas por uma outra função, neste caso a função "`void initLog(Data8 max)`". Portanto, se não for definido (ou atribuído) os valores de "`buffer_size`" e "`next`", por meio da chamada da função `initLog()`, pode ocorrer que durante a execução do código seja atribuído um valor não determinístico as variáveis "`buffer_size`" e "`next`", por exemplo, zero para "`buffer_size`" (no qual poderia causar uma divisão por zero) ou um valor maior que "`BUFFER_MAX`", onde esta define a constante para o tamanho do *array*, para "`next`" (no qual poderia ocasionar um *buffer overflow*). Assim para a identificação destes tipos de erros, algumas vezes é necessário que se explore outras partes do código analisado, onde dependendo do

[4] O oxímetro de pulso é responsável por medir a saturação de oxigênio (SpO_2) e freqüência cardíaca (FC) no sistema arterial utilizando um método não-invasivo.

código analisado, esta pode torna-se uma tarefa muito difícil.

```
1 void insertLogElement(Data8 b){
2         buffer[next] = b;
3         next = (next+1)%buffer_size;
4 }
```

Figura 5.1. O fragmento do código Oximeter_log_det.c.

As atribuições de valores para a execução dos testes com a aplicação do método foram feitas com base nos possíveis problemas acima identificados, isso é: (i) a função "`initLog()`" não é chamada; e (ii) a função "`insertElementLog`" é executada diversas vezes até que a variável "`next`" se torne maior que "`BUFFER_MAX`". Os resultados desta verificação são demonstrados na Figura 5.2, onde podemos notar que para a primeira situação, o *assert* atribuído foi executado 2 vezes, antes que a propriedade contida na assertiva fosse violada, enquanto para a segunda situação, o *assert* atribuído foi executado 12801 vezes antes que a propriedade (ou *assert*) fosse violada.

(i) Primeira Situação

```
Suite: log
  Test: testClaims ... FAILED
    1. log_CUnit.c:188 - (unsigned int)buffer_size != 0
--Run Summary: Type      Total      Ran      Passed      Failed
                suites      1        1         n/a          0
                 tests      1        1           0          1
                asserts     2        2           1          1
```

(ii) Segunda Situação

```
Suite: log
  Test: testClaims ... FAILED
    1. log_CUnit.c:133 - next < 6400
--Run Summary: Type      Total      Ran      Passed      Failed
                suites      1        1         n/a          0
                tests       1        1           0          1
                asserts  12801    12801       12800         1
```

Figura 5.2. Resultado da verificação executada com o método no código Oximeter_log_det.c

A abstração dos dados relevantes às propriedades geralmente não requer um grande esforço. Entretanto, se o número de LOCs é alto ou a quantidade de propriedades é grande (durante os nossos experimentos identificamos códigos com mais de 93 propriedades), então está tarefa pode torna-se muito difícil. A implementação do método proposto não somente serve como base para a abstração das propriedades de segurança geradas pelos *model checkers*, mas também como modelo de base para os

engenheiros de testes que já estão familiarizados com as ferramentas e frameworks de teste unitário. A automatização desse método pode ser assim uma grande vantagem.

5.2 Resultados Experimentais em Instanciação de Programas em C pela Utilização de Contra-Exemplos de BMC

As etapas do método EZProofC foram implementadas usando o ESBMC v.1.11. A Tabela 5.2 contém os detalhes sobre os *benchmarks* dos programas em ANSI-C usados em nossos experimentos.

A primeira coluna da Tabela 5.2, contém o número das aplicações; a segunda coluna descreve o nome de cada módulo composto pelo respectivo *benchmark*, programa em C, e a entrada (por exemplo, determinística ou não determinística); a terceira coluna demonstra o número de linhas do código (LOC); a quarta coluna provê o total do número de propriedades identificadas (`#P`); e finalmente, a quinta coluna contém o número de propriedades violadas e que foram provadas utilizando o método EZProofC (`#P EZProofC`).

Os *benchmarks* utilizados foram EUREKA[5], SNU[6], WCET[7], NEC[8], SIR[9] [Do et al., 2005] e alguns código baseados na documentação do CBMC em D_CBMC[10]. Os resultados da aplicação do método EZProofC, bem como a primeira versão da ferramenta que contém a automatização do método estão disponíveis em https://sites.google.com/site/ezproofc/.

O método EZProofC utiliza o ESBMC para executar a verificação de cada propriedade identificada no código. Adicionalmente, checando a viabilidade do método para identificar falsos contra-exemplos que possam ser gerados pelo ESBMC. Assim poderemos explorar os *model checkers*, de modo a verificar programas maiores, quebrando o modelo global (que contém todo o programa) em modelos locais (contendo apenas as funções em teste).

[5]http://www.ai-lab.it/eureka/bmc.html
[6]http://archi.snu.ac.kr/realtime/benchmark
[7]http://www.mrtc.mdh.se/projects/wcet/benchmarks.html
[8]http://www.nec-labs.com/research/system/systems_SAV-website/benchmarks.php
[9]http://sir.unl.edu/portal/index.html
[10]http://www.cprover.org/cbmc/doc/manual.pdf

Tabela 5.2. Detalhes relacionados aos *benchmarks* de códigos C utilizados.

Nº	Módulo	LOC	#P	#P EZProofC
1	EUREKA_no_init_sel_sort5.c	40	17	4
2	EUREKA_strcmp7.c	15	6	4
3	EUREKA_sum_array.c	16	7	1
4	SNU_select_det.c	122	39	3
5	SNU_crc_det.c	125	15	0
6	WCET_cnt_nondet.c	139	16	0
7	NEC_ex26.c	30	8	2
8	SIR_print_tokens2.c	508	51	1
9	D_CBMC_assert_unsafy.c	20	1	1
10	D_CBMC_division_by_zero.c	32	1	1
11	D_CBMC_bound_array.c	15	10	1

Na Tabela 5.2, nas linhas de 1 a 3, o número de propriedades violadas que foram identificadas são respectivamente 4, 4 e 1. Em geral, estas propriedades estavam relacionadas à segurança de assertivas já inseridas no código, exceto o código na linha 2, onde ele estava relacionada a segurança dos limites (`UPPER BOUND`) dos *arrays*.

Nas linhas de 4 a 6 na Tabela 5.2, o número de propriedades que foram identificadas são respectivamente 3, 0 e 0. No código da linha 4 houve uma situação específica com relação ao contra-exemplo, na etapa de instanciação e execução do código instanciado, percebemos que para comprovar a violação da propriedade `ir < 20` que resultaria em um `UPPER BOUND`, não necessitávamos instanciar todos os dados identificados no contra-exemplo, como demonstrado na Figura 5.3. O contra-exemplo continha os dados gerados a partir do *trace* para cada estado, no qual resultaria na violação da propriedade.

```
Counterexample:

State 4 file SNU_select_det.c line 68 function select thread 0
------------------------------------------------------------
SNU_select_det::select::1::ir=20 (00000000000000000000000000010100)

State 11 file SNU_select_det.c line 79 function select thread 0
------------------------------------------------------------
arr={ 5f, 4f, 10f, 1.100006103515625f, 5.6999969482421875f, 100f, 231f, 111f, 49.5f,
      99f, 10.3000030517578125f, 150f, 222.220001220703125f, 101f, 77f, 44f, 35f,
      20.5399932861328125f, 99.9900054931640625f, 888.8800048828125f }

Violated property:
  file SNU_select_det.c line 80 function select
  array `arr' upper bound  ir < 20

VERIFICATION FAILED
```

Figura 5.3. Estados e dados utilizados do contra-exemplo

A análise do *trace* nos revelou que no `State 4` a variável `ir` recebia como valor de entrada o número `20`, onde esta variável é o índice do *array* `arr` no qual estava definido no código como um *array* com capacidade para `20` elementos, todavia ocorria que era atribuído ao *array* `arr` um 21º elemento, assim está atribuição resultaria na violação da propriedade. Logo, nós podemos comprovar o erro sem a necessidade de implementar as outras informações. Este item é discutido como um item específico para trabalhos futuros.

Nas linhas de 7 a 11 na Tabela 5.2, o número de propriedades violadas que foram identificadas são respectivamente 2, 1, 1, 1 e 1. Estes códigos basicamente tratam sobre segurança de ponteiros, segurança de assertivas providas pelo usuário, divisão por zero e `UPPER` ou `LOWER` *bound* dos índices dos *arrays*.

O código na linha 7 em especial diz respeito a propriedades como `POINTER_OFFSET` e `SAME-OBJECT` voltada para a análise de segurança de ponteiros e alocação de memória dinâmica, neste caso o método foi implementado como mencionado na Seção 4.2.3 em relação a análise da propriedade em um contexto geral, obtendo uma assertiva da propriedade identificada, neste caso a propriedade 8 do código `!(2 * y + POINTER_OFFSET(x) >= 200) || !(SAME-OBJECT(x, &b[0]))`, no qual o erro foi identificado como um *UPPER BOUND*, a partir desta propriedade, obtemos a seguinte assertiva (2*y + x >= 200).

A verificação em especial do código na linha 8 da Tabela 5.2 foi feita com base nas funções (Seção 4.2.2), em especifico a função *"get_ token"*, pois como o código é grande, verificar o mesmo como um todo não é trivial, demandando assim uma quantidade de memória significativa, em muitos casos podendo gerar falta de memória durante a verificação. O erro identificado neste código trata-se da violação do limite superior (*upper bound*) do *array* `buffer` no qual é declarado com um limite superior de 80, todavia com base na comprovação do erro nota-se que o índice deste *array* o `i`, ultrapassa o limite superior, ocasionando a violação da propriedade ‘‘`i<81`’’.

Nossos experimentos demonstram que a abstração e manipulação da informação no contra-exemplo nem sempre é uma tarefa trivial (por exemplo, durante os nossos experimentos nos obtivemos contra-exemplos com aproximadamente 3.200 linhas). Todavia, a aplicação do método EZProofC diminui substancialmente a complexidade desta tarefa além de validar possíveis falsos positivos gerados durante a verificação com o *model checker*.

Capítulo 6

Conclusões e Trabalhos Futuros

Neste trabalho apresentamos duas abordagens que visam complementar a verificação efetuada pelos *bounded model checkers* e contribuir com familiarização da técnica de verificação formal *model checking*. O problema resolvido neste trabalho focou em utilizar as características providas pelos *bounded model checkers* para efetuar a verificação de propriedades de segurança, integrado ao ambiente de teste unitário, sem a obtenção de falta de memória durante a verificação e a comprovação de erros de programas identificado por estes.

A primeira abordagem demonstra um novo método, denominado de **FORTES** (*FORmal unit TESt generation*), para integrar o *model checker* ESBMC com o *framework* de teste unitário CUnit, de modo a criar e executar casos de testes em programas sequênciais em C. Basicamente, este método visa extrair as propriedades de segurança geradas automaticamente pelo ESBMC para criar casos de teste usando um rico conjunto de assertivas providas pelo *framework* de teste CUnit. A segunda abordagem apresenta um método, denominado de **EZProofC**, que visa automatizar a coleta e manipulação de dados gerados pelos *bounded model checkers* de modo a comprovar a causa raiz do erro identificado.

Os resultados experimentais para a primeira abordagem, demonstrou ser muito eficaz sobre os *benchmarks* públicos disponíveis, se comparando a utilização somente do *model checker*, tomando como base que esta abordagem obteve um gasto menor de memória e tempo de execução que a utilizada pelo *model checker*, não deixando a verificação incompleta devido a falta de mémoria. Nota-se também que com o conjunto de testes executados, fomos capazes de identificar alguns pontos que serão tratados em trabalhos futuros, com relação ao tratamento e abstração de informações geradas pelo *bounded model checker* (e respectivamente as propriedades de segurança), objetivando a melhoria destas, tais como, a estrutura do código, a análise de propriedades

relacionadas a ponteiros e alocação de memória dinâmica.

Para trabalhos futuros também, pretendemos investigar a aplicação de técnicas de verificação com instrumentação de código [Nethercote & Seward, 2007] e outras abordagens, tais como a execução de código com entradas simbólicas [Cadar & Engler, 2005], injeção de falhas [Arlat et al., 1990] e testes mutantes [Jia & Harman, 2010] que nos permitirá melhorar o método proposto. Assim como verificamos durante os experimentos em alguns casos, somente a execução do fluxo do código com os seus valores pode não ser suficiente para explorar e identificar um determinado erro, desta forma a inclusão destas técnicas de verificação nós permitirá a geração e execução de casos de teste com maior poder de exploração em uma ampla gama de propriedades.

Os resultados experimentais para a segunda abordagem, também demonstram-se muito eficaz sobre diversos *benchmarks* públicos, uma vez que foi possível utilizar e obter-se o resultado esperado em todos os casos demonstrados com a abordagem proposta. Nota-se que com o conjunto de testes executados, nós fomos capazes de identificar algumas melhorias que podem ser implementadas nesta abordagem, com relação ao tratamento e abstração de informações geradas pelo *bounded model checker* (e propriedades de segurança respectivamente), tais como, a análise de contra-exemplos relacionados a *arrays*, ponteiros e alocação de memória dinâmica, no qual necessitam de testes mais extensivos de modo a identificar outras situações que requerem tratamentos específicos. Assim como identificamos em um caso durante os experimentos, no qual para comprovar o erro não era necessario a utilização de todas as informações do contra-exemplo, deste modo como trabalhos futuros pretendemos criar heurísticas para análise de contra-exemplos.

Para trabalhos futuros, também com relação a esta segunda abordagem, pretendemos investigar como podemos adaptar o método proposto para se utilizar por outros *model checkers* com os seus respectivos contra-exemplos, tais como o Blast [Beyer et al., 2007b] e Java PathFinder [Havelund, 1999], de modo a permitir que os dados identificados e gerados nos contra-exemplos possam ser explorados e manipulados, onde este processo tem como entrada principal de informação para o método proposto as expressões regulares que caracterizam os dados gerados pelo contra-exemplo do seu respectivo *model checker*, assim tornando o método proposto flexível a outros *model checkers*.

Referências Bibliográficas

Arlat, J.; Aguera, M.; Amat, L.; Crouzet, Y.; Fabre, J.-C.; Laprie, J.-C.; Martins, E. & Powell, D. (1990). Fault injection for dependability validation: A methodology and some applications. *IEEE Transactions on Software Engineering*, 16:166–182.

Baier, C. & Katoen, J.-P. (2008). *Principles of Model Checking*. MIT Press.

Ball, T. (1999). The Concept of Dynamic Analysis. In *Proc. of the 7th European Software Engineering Conference (ESE'99)*, pp. 216--234. Springer-Verlag.

Ball, T.; Naik, M. & and, S. R. (2003). From symptom to cause: Localizing errors in counterexample traces. *In Proc. of POPL*.

Barnett, M.; Grieskamp, W.; Gurevich, Y.; Schulte, W.; Tillmann, N. & Veanes, M. (2003). Scenario-oriented modeling in asml and its instrumentation for testing. In *SCESM 2003*. Springer.

Beer, I.; Ben-David, S.; Chockler, H.; Orni, A. & Trefler, R. (2009). Explaining counterexamples using causality. In *Computer Aided Verification*, volume Volume 5643/2009, pp. 94–108. Springer Berlin / Heidelberg.

Ben-Ari, M. (2008). *Principles of the Spin Model Checker*. 1 edição.

Beyer, D.; Henzinger, T.; Jhala, R. & Majumdar, R. (2007a). The software model checker blast: Applications to software engineering. In *International Journal on Software Tools for Technology Transfer (STTT)*, pp. 505--525.

Beyer, D.; Henzinger, T. A.; Jhala, R. & Majumdar, R. (2007b). The software model checker blast - applications to software engineering. *Springer-Verlag*.

Borges, Rafael Magalh a. & Mota, A. C. (2007). Integrating uml and formal methods. In *Electron. Notes Theor. Comput. Sci.*, pp. 97--112.

Bradley, A. R. & Manna, Z. (2007). *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Cadar, C. & Engler, D. (2005). Execution generated test cases: How to make systems code crash itself. In *SPIN*, pp. 2–23.

Calimeri, F.; Perri, S. & Ricca, F. (2008). Experimenting with parallelism for the instantiation of asp programs. *J. Algorithms*, 63:34--54.

Caspi, P.; Pilaud, D.; Halbwachs, N. & Plaice, J. A. (1987). Lustre: a declarative language for real-time programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL'87, pp. 178--188, New York, NY, USA. ACM.

Chechik, M. & Gurfinkel, A. (2005). A framework for counterexample generation and exploration. *In Proc. of FASE*.

Clarke, E.; Fehnker, A.; Han, Z.; Krogh, B.; Stursberg, O. & Theobald, M. (2003). Verification of hybrid systems based on counterexample-guided abstraction refinement. In *Proceedings of the 9th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'03, pp. 192--207, Berlin, Heidelberg. Springer-Verlag.

Clarke, E.; Grumberg, O. & Peled, D. (1999). *Model Checking*. MIT Press.

Clarke, E. M. (2008). 25 years of model checking. chapter The Birth of Model Checking, pp. 1--26. Springer-Verlag, Berlin, Heidelberg.

Clarke, E. M. & Wing, J. M. (1996). Formal methods: State of the art and future directions. volume 28, pp. 626--643. ACM Computing Surveys.

Copty, F.; Irron, A.; Weissberg, O.; Kropp, N. & Kamhi, G. (2001). Efficient debugging in a formal verification environment. *In Proc. of CHARME*.

Cordeiro, L.; Fischer, B.; Chen, H. & Marques-Silva, J. (2009a). Semiformal verification of embedded software in medical devices considering stringent hardware constraints. In *International Conference on Embedded Software and Systems*, pp. 396--403.

Cordeiro, L.; Fischer, B. & Marques-Silva, J. (2009b). SMT-based bounded model checking for embedded ANSI-C software. In *Automated Software Engineering*, pp. 137--148.

C.Wang; Yang, Z.; Ivancic, F. & Gupta, A. (2006). Whodunit? causal analysis for counterexamples. *In Proc. of ATVA*.

Desoli, G.; Mateev, N.; Duesterwald, E.; Faraboschi, P. & Fisher, J. A. (2002). Deli: a new run-time control point. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 35, pp. 257--268, Los Alamitos, CA, USA. IEEE Computer Society Press.

Do, H.; Elbaum, S. G. & Rothermel, G. (2005). Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405--435.

Dong, Y.; Ramakrishnan, C. R. & Smolka, S. A. (2003). Model checking and evidence exploration. *In Proc. of ECBS*.

D'Silva, V.; Kroening, D. & Weissenbacher, G. (2008). A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 27(7):1165–1178.

Ernst, M. D. (2003). Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pp. 24--27, Portland, OR.

Fisteus, J. A. (2005). *Definición de Un Modelo Para La Verificación Formal de Procesos de Negocio*. PhD thesis, Universidad Carlos III de Madrid, Departamento De Ingeniería Telemática.

France, R. & Rumpe, B. (2007). Model-driven development of complex software: A research roadmap. In *FOSE '07: 2007 Future of Software Engineering*, pp. 37--54.

Griesmayer, A. & and, S. S. (2007). Automated fault localization for c programs. *ENTCS*.

Groce, A. (2004). Error explanation with distance metrics. *In Proc. of TACAS*.

Groce, A. & Joshi, R. (2008). Extending model checking with dynamic analysis. In *VMCAI'08: Proceedings of the 9th international conference on Verification, model checking, and abstract interpretation*, pp. 142--156, Berlin, Heidelberg. Springer-Verlag.

Groce, A. & Kroening, D. (2005). Making the most of bmc counterexamples. *Proceedings of the 2nd International Workshop on Bounded Model Checking (BMC 2004)*, Volume 119.

Halpern, J. Y. & Pearl, J. (2000). Causes and explanations: A structural-model approach, part i: Causes. *CoRR*, cs.AI/0011012.

Havelund, K. (1999). Java pathfinder, a translator from java to promela. In *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*, p. 152, London, UK. Springer-Verlag.

Ji, J. H.; Woo, G.; Park, H. B. & Park, J. S. (2008). Design and implementation of retargetable software debugger based on GDB. In *Proceedings of the 2008 Third International Conference on Convergence and Hybrid Information Technology - Volume 01*, pp. 737--740.

Jia, Y. & Harman, M. (2010). An analysis and survey of the development of mutation testing. In *IEEE Transactions on Software Engineering*.

Jin, H.; Ravi, K. & Somenzi, F. (2002). Fate and free will in error traces. *In Proc. of TACAS*.

Kerbrat, A.; Jéron, T. & Groz, R. (1999). Automated test generation from sdl specifications. In *SDL Forum'99*, pp. 135–152.

Luk, C. K.; Cohn, R.; Muth, R.; Patil, H.; Klauser, A.; Lowney, G.; Wallace, S.; Janapa, V. & Hazelwood, R. K. (2005). Pin: Building customized program analysis tools with dynamic instrumentation. In *In Programming Language Design and Implementation*, pp. 190--200. ACM Press.

Marre, B. & Arnould, A. (2000). Test sequences generation from lustre descriptions: Gatel. In *15th IEEE ICASE*.

Matloff, N. & Salzman, P. J. (2008). *The Art of Debugging with GDB, DDD, and Eclipse*. No Starch Press, San Francisco, CA, USA.

Myers, G. J. & Sandler, C. (2004). *The Art of Software Testing*. John Wiley & Sons.

Nagarakatte, S.; Zhao, J.; Martin, M. M. & Zdancewic, S. (2010). Cets: compiler enforced temporal safety for c. *SIGPLAN Not.*, 45(8):31--40.

Nethercote, N. (2004). Dynamic binary analysis and instrumentation. Technical Report UCAM-CL-TR-606, University of Cambridge, Computer Laboratory.

Nethercote, N. & Seward, J. (2007). Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89--100.

Petrenko & Alexandre (2001). Fault model-driven test derivation from finite state models: Annotated bibliography. In *Modeling and Verification of Parallel Processes*, pp. 196--205.

Riacs, W. V.; Groce, A.; Groce, A.; Visser, W. & Visser, W. (2002). What went wrong: Explaining counterexamples. In *In SPIN Workshop on Model Checking of Software*, pp. 121–135. Springer-Verlag.

Schmitt, M.; Ebner, M. & Grabowski, J. (2000). Test Generation with Autolink and TestComposer. In *Proceedings of the 2nd Workshop of the SDL Forum Society on SDL and MSC (SAM'2000)*, pp. 26--28.

Shen, S.; Qin, Y. & Li, S. (2005). A faster counterexample minization algorithm based on refutation analysis. *In Proc. of DATE*.

Sherer, S. A. (1991). A cost-effective approach to testing. *IEEE Software*, 8:34–40.

Srivastava, A. & Eustace, A. (1994). Atom: a system for building customized program analysis tools. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pp. 196--205, New York, NY, USA. ACM.

Staber, S. & Bloem, R. (2007). Fault localization and correction with qbf. *In Proc. of SAT*.

Sülflow, A.; Fey, G.; Bloem, R. & Drechsler, R. (2008). Using unsatisfiable cores to debug multiple design errors. *In Proc. of Symp on VLSI*.

Taghdiri, M. (2004). Inferring specifications to detect errors in code. *Automated Software Engineering*, 0:144–153.

Tretmans, G. J. & Brinksma, H. (2003). Torx: Automated model-based testing. In Hartman, A. & Dussa-Ziegler, K., editores, *ECMDSE*, pp. 31--43.

Wang, L.; Zhang, Q. & Zhao, P. (2008). Automated detection of code vulnerabilities based on program analysis and model checking. *Source Code Analysis and Manipulation, IEEE International Workshop on*, 0:165–173.