UNIVERSIDADE FEDERAL DE PERNAMBUCO

GRADUAÇÃO EM ENGENHARIA DA COMPUTAÇÃO

CENTRO DE INFORMÁTICA

Model Checkers: Uma análise de ferramentas para a linguagem de programação C

TRABALHO DE GRADUAÇÃO

Aluno: Pedro Montenegro Rodrigues (pmr@cin.ufpe.br)
Orientador: Alexandre Cabral Mota (acm@cin.ufpe.br)
Co - Orientador: Augusto César Alves Sampaio (acas@cin.ufpe.br)

Recife, 27 de Agosto de 2007

UNIVERSIDADE FEDERAL DE PERNAMBUCO

GRADUAÇÃO EM ENGENHARIA DA COMPUTAÇÃO

CENTRO DE INFORMÁTICA

2007.1

Model Checkers: Uma análise de ferramentas para a linguagem de programação C

Monografia apresentada ao Centro de Informática da Universidade Federal de Pernambuco, como requisito parcial para obtenção do Grau de Engenheiro da Computação.

Aluno: Pedro Montenegro Rodrigues (pmr@cin.ufpe.br)
Orientador: Alexandre Cabral Mota (acm@cin.ufpe.br)
Co - Orientador: Augusto César Alves Sampaio (acas@cin.ufpe.br)

Recife, 27 de agosto de 2007

AGRADECIMENTOS

Eu quero agradecer a todos que, de algum modo, contribuíram direta ou indiretamente para a realização deste trabalho. Este trabalho é um resultado de dedicação e de sustentação daqueles a quem eu dou os meus agradecimentos:

- Primeiramente, eu agradeço a Deus por estar sempre ao meu lado, por ter me dado o dom da vida.
- Aos meus pais, Adilson e Tânia, pelo constante incentivo, carinho, compreensão, dedicação, amor e esforço para que eu pudesse chegar até aqui.
- Ao meu irmão, Rafael, pela amizade e compreensão.
- A minha namorada, Juliana, pelo término do trabalho, em detrimento de vê-la.
- Agradeço também a todos os meus amigos por estarem sempre comigo e compreenderem os meus momentos de ausência.
- Ao meu orientador, Alexandre Cabral Mota, e ao meu co-orientador, Augusto César Alves Sampaio, pelo incentivo, confiança e disponibilidade na elaboração do trabalho.

RESUMO

Sistemas críticos necessitam da garantia de funcionamento perfeito, uma vez que falhas podem levar a perdas financeiras ou de vidas humanas. Um dos pontos fundamentais para garantir esse alto nível de qualidade é a análise formal de propriedades desejáveis.

Uma das técnicas que vem obtendo sucesso nos últimos anos é a de verificação de modelos (model checking), uma técnica completamente automática para analisar sistemas críticos, na qual se verifica automaticamente a validade de propriedades em sistemas acerca do seu funcionamento.

Neste contexto, este trabalho tem como objetivo pesquisar verificadores de modelos para a linguagem de programação C e fazer uma análise das características de alguns dos mais utilizados, fazendo um estudo do que cada um oferece, da abordagem que utiliza, das suas vantagens e desvantagens e concluir apontando a ferramenta que apresentou os melhores resultados, baseado em suas características. O trabalho também tem o objetivo de aplicar uma das ferramentas analisadas a um componente do sistema crítico de um metrô, relatando os resultados obtidos.

Palavras chave: Engenharia de software, Linguagem C, Model Checkers, Métodos Formais

ABSTRACT

Critical systems need the guarantee of perfect functioning, a time that imperfections can take the financial losses or of lives human beings. One of the basic points to guarantee this high level of quality is the formal analysis of desirable properties.

One of the techniques that come getting success in recent years is of verification of models (model checking), one completely automatic technique to analyze critical systems, in which if it automatically verifies the validity of properties in systems concerning its functioning.

In this context, this work has as objective to search verifiers of models for the programming language C and to make an analysis of characteristics of some of the most used, making a study of what each one offers, of the boarding that it uses, of its advantages and disadvantages and to conclude pointing the tool that presented the best ones resulted, established in its characteristics. The work also has the objective to apply one of the tools analyzed to a component of the critical system of a subway, being told the gotten results.

Key words: Software Engineering, C Language, Model Checkers, Formal Methods.

SUMÁRIO

LISTA DE SIGLAS

BMC – Bounded Model Checking

CTL – Computation Tree Logic

LTL – Linear Temporal Logic

MEF – Maquina de Estados Finita

FSA – Autômato de Estados Finito

CFG – Gráfico do Fluxo de Controle

BDD – Diagrama de Decisão Binário (Binary Decision Diagram)

SAT – Refere-se ao problema de determinar se uma proposição pode ser satisfeita ou não. A abreviação vem do inglês satisfiability.

LISTA DE FIGURAS

LISTA DE TABELAS

1 - INTRODUÇÃO

Sistemas críticos necessitam da garantia de funcionamento perfeito, uma vez que falhas podem levar a perdas financeiras ou de vidas humanas. Um dos pontos fundamentais para garantir esse alto nível de qualidade é a análise formal de propriedades desejáveis. Para sistemas concorrentes, a análise ainda é mais complexa que para sistemas seqüenciais. Isto gera a necessidade de um suporte ferramental que de alguma forma, automatize o processo de verificação deste tipo de programa.

Uma das técnicas que vem obtendo sucesso nos últimos anos é a de verificação de modelos (model checking), uma técnica completamente automática para analisar sistemas, na qual se verifica automaticamente a validade de propriedades em sistemas acerca do seu funcionamento. Esta é uma técnica automática para analisar o espaço de estados finito de sistemas. Tradicionalmente, a aplicação da verificação de modelos ocorre através de três etapas: modelagem (construção de um modelo formal do sistema que contenha o comportamento do sistema), especificação dos comportamentos desejáveis e verificação (modelo e especificações são submetidos ao verificador de modelos). Na última etapa é obtida uma resposta do verificador: verdadeiro (as especificações são satisfeitas pelo modelo) ou falso mais contra-exemplo (um “trace” com os passos que levam a um estado onde as especificações não são verdadeiras).

1.1 - MOTIVAÇÃO

Hoje em dia, existem sistemas de controle aplicados a áreas tão distintas quanto ferrovias, aeronaves, caixas eletrônicos de banco, etc. Apesar dos diferentes graus de prejuízo que uma eventual falha em algum destes sistemas possa provocar, um alto grau de confiabilidade é exigido para quase todos eles. Sistemas de controle críticos como intertravamentos metroviários e controles automáticos de aeronaves podem colocar vidas humanas sob grave risco em caso de ocorrência de alguma falha. Mesmo os sistemas de controle que não ameaçam diretamente a integridade humana em caso de falha, devem

possuir um alto grau de confiabilidade. Esta afirmação se sustenta pelo fato que eventuais falhas podem resultar em graves transtornos econômicos para as empresas envolvidas. Assim, a garantia de correção da maior parte dos sistemas de controle é importante por um motivo ou por outro.

A necessidade de garantia de correção dos sistemas de controle, entretanto, 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.

Nos últimos anos as indústrias reconheceram os verificadores de modelos como uma ferramenta promissora para o desenvolvimento de sistemas embarcados. Muitos sistemas embarcados são usados dentro ambientes críticos de segurança. Testar completamente esses sistemas não é freqüentemente possível porque é demasiadamente caro ou consome muito tempo. De qualquer modo, erros nestes sistemas podem conduzir a problemas fatais. A maioria dos sistemas embarcados utilizam micro-controladores. Os softwares para micro-controladores são, na maioria dos casos, escritos na linguagem de programação C. Conseqüentemente, pode-se esperar que em quase todos os projetos de software embarcado, existe código C para micro-controladores. A verificação de modelos deve ser feita sem a necessidade de pré-processamento manual do programa. Para uma adaptação do modelo que se verifica na indústria, não é praticável que se use horas para preparar programas para verificação. Se a verificação de modelos dever ser usado na indústria, é essencial que os programas que são criados no desenvolvimento possam ser usados para a verificação sem preparação manual. Além disso, um pré-processamento manual poderia introduzir novos erros ou mascarar erros existentes.

Por todos estes fatores citados, esse estudo sobre ferramentas para a linguagem de programação C será muito importante.

1.2 - OBJETIVOS

O contexto deste trabalho é fruto de uma cooperação de pesquisa entre CIn/UFPE e uma empresa de São Paulo responsável por um sistema do complexo de metrô de Santiago no Chile, estando relacionado mais especificamente à um sistema de controle geral de portas (CGP) de um metrô. Este trabalho tem como objetivo procurar ferramentas Model Checkers (Verificadores de Modelos) para a linguagem de programação C, descrever o funcionamento de algumas delas e fazer uma análise das características de algumas das mais utilizadas, realizando um estudo do que cada uma oferece, das suas vantagens e desvantagens, uma análise das abordagens utilizadas e concluir apontando a ferramenta que apresentou os melhores resultados para o contexto do trabalho. Um estudo de caso será feito utilizando um verificador de modelos em um componente crítico de um sistema de metrô [20], analisando, por exemplo, a alcançabilidade de estados, que apesar de ser uma propriedade típica de Static Checkers, é possível verificá-la através de um Model Checker.

De forma resumida, as contribuições deste trabalho são:

- Pesquisar Verificadores de modelos para a linguagem C.
- Relatar abordagens utilizadas pelas ferramentas.
- Descrever o funcionamento das mais utilizadas.
- Analisar as características de algumas delas.
- Aplicar na prática uma ferramenta a um componente crítico do sistema de um metrô.

1.3 - ESTRUTURA DO TRABALHO

Além deste capítulo introdutório, o trabalho é composto por mais cinco capítulos. O capítulo 2 tem por objetivo uma introdução à verificação de modelos (model checking), discutindo os principais conceitos e abordagens. O capítulo 3 faz uma breve descrição das ferramentas de verificação de modelos (model checking) para a linguagem de programação C, relatando suas

características. O capítulo 4 faz uma análise de algumas das ferramentas descritas no capitulo 3, as mais utilizadas, além de especificar vantagens, desvantagens e também limitações de cada uma destas ferramentas. O capítulo 5 corresponde ao estudo de caso do componente do metrô, especificando o componente e relato dos resultados obtidos ao aplicar uma ferramenta ao componente. O capítulo 6 apresenta a conclusão, os trabalhos relacionados e propostas para trabalhos futuros.

2 - MODEL CHECKING

Verificação de modelos (Model Checking) provou ser uma tecnologia bem sucedida para verificar exigências e projetá-las para uma variedade de sistemas embarcados e para segurança de sistemas críticos de tempo real. Antes de iniciar o código em um projeto, se enfrenta o problema crônico do desenvolvimento do software: requisitos danificados do projeto. Faz sentido encontrar falhas logo no inicio, porque os requisitos danificados produzem os erros que terão que ser consertados mais tarde, elevando o custo do projeto.

Verificação de modelos é uma técnica totalmente automática que analisa o espaço de estados finito de sistemas críticos. Em métodos formais, a abordagem de verificação de modelos (model checking) vem obtendo bastante sucesso nos últimos anos, onde se verifica automaticamente a validade de propriedades acerca do comportamento de sistemas. Esta técnica verifica propriedades de um sistema através de enumeração exaustiva de todos os estados alcançáveis. Para realizar a validação das propriedades de sistemas seguem-se os três passos abaixo:

1. Modelagem: esta etapa consiste na construção de um modelo formal do sistema e extrair dele todos os comportamentos possíveis do sistema. A estrutura que contém todos os comportamentos possíveis é conhecida como espaço de estados do sistema.

2. Especificação: esta etapa consiste na especificação dos comportamentos desejados do sistema, ou seja, especificação das propriedades. Um comportamento que se deseja do sistema pode ser descrito formalmente através de lógicas temporais ou máquinas de estado.

3. Verificação: esta etapa consiste em submeter o modelo e as especificações das propriedades a uma ferramenta chamada de verificador de modelos (model checker). Esta ferramenta oferece como resultado um valor verdade que indica se a especificação é satisfeita ou não no modelo. Em caso negativo, o verificador fornece uma seqüência

de estados alcançáveis, chamada de contra-exemplo, que demonstra que a especificação não é válida no modelo.

A Figura 2.1 descreve a arquitetura de uma ferramenta de verificação de modelos (model checking), denominada *verificador (model checker).* O módulo *Verificador de Modelos* representa a ferramenta de verificação. Como entrada para a ferramenta, o *Espaço de Estados* na figura representa todos os comportamentos possíveis de um sistema. A *Especificação de Propriedade* representa a descrição de uma propriedade comportamental que se deseja checar sobre o espaço de estados. Como saída, o verificador responde *Sim*, caso a propriedade seja válida, ou *Não*, caso contrário. Quando a resposta é *Não*, um contra-exemplo é apresentado: o verificador exibe um caminho no espaço de estados em que a propriedade especificada não acontece.

Figura 2.1 - Arquitetura da técnica de verificação de modelos

Analisando o contra-exemplo, é possível localizar a fonte do erro no modelo, corrigir o modelo, e verificá-lo novamente. A idéia é que, assegurando-se de que o modelo satisfaça a várias propriedades do sistema, a confiança na exatidão do modelo aumenta. Os requisitos variam extremamente para sistemas em domínios diferentes da aplicação. Por exemplo, os requisitos de um sistema de operação bancária e de um sistema metroviário diferem no tamanho, na estrutura, na complexidade, na natureza de dados do sistema, e nas operações executadas.

A técnica de verificação de modelos utiliza as lógicas temporais para a especificação de propriedades, porque elas podem expressar relações de ordem, sem recorrer à noção explícita de tempo. Duas abordagens de lógica

temporal são utilizadas no contexto de verificação de modelos: LTL (Linear Temporal Logic) e CTL (Computation Tree Logic).

2.1 - LÓGICA TEMPORAL

A lógica Temporal suporta a formulação das afirmações sobre o comportamento de um sistema reativo enquanto ele evolui com o tempo. Sistemas reativos têm como características básicas estados e transições. Estado é a descrição do sistema em um dado instante de tempo, ou seja, os valores associados as suas variáveis naquele instante. Transição é uma relação entre dois estados. Tipicamente, as afirmações sobre o comportamento do sistema incluem as propriedades de segurança, definindo o que deve sempre ser verdadeiro de um sistema e um conjunto de propriedades de vivacidade (liveness), refletindo circunstâncias que um sistema deve eventualmente satisfazer. As propriedades a serem verificadas em um sistema reativo são expressas como fórmulas, que especificam os comportamentos desejados, de uma linguagem de lógica temporal. O principal objetivo de um verificador de modelos é verificar se um modelo satisfaz um conjunto de propriedades desejadas especificadas pelo usuário.

Essas propriedades podem ser definidas utilizando-se duas lógicas temporais: Lógica de Árvore de Computação, ou *Computation Tree Logic* (**CTL**); e Lógica Temporal Linear, ou *Linear Temporal Logic* (**LTL**). Tais lógicas se referem à noção de seqüências de estados, e não a valores de tempos, ou a intervalos de tempos, pois se deseja tratar de comportamentos de sistemas não-determinísticos que envolvem diferentes caminhos. Isto é, cada estado pode ter vários sucessores em termos de ramificação ou o comportamento é dado por um conjunto de caminhos que são lineares. LTL é uma lógica temporal de tempo linear que interpreta fórmulas sobre funcionamentos do sistema, sendo mais apropriada para especificar seqüências de teste. Já a CTL é uma lógica de tempo ramificado que interpreta fórmulas sobre árvores de computação, que permite raciocinar sobre propriedades estruturais do sistema e de considerar vários critérios empregados nos testes.

2.1.1 - LTL

A lógica LTL faz a caracterização de cada caminho linear proporcionados pelas MEFs, já as especificações na lógica CTL expressam propriedades sobre a árvore de computação das MEFs. As duas lógicas possuem um poder expressivo diferente, mas também possuem vários pontos em comum, os quais incluem a maioria das propriedades usadas na prática. Diferentemente de CTL, os operadores temporais de LTL não possuem quantificadores de caminho. De fato, fórmulas LTL são avaliadas sobre caminhos lineares, e uma fórmula somente é considerada verdadeira em um modelo se ela é verdadeira para todos os caminhos iniciando num dos estados iniciais daquele modelo.

2.1.2 - VERIFICANDO MODELOS NA LÓGICA LTL

A verificação de modelos é aplicada para validação de propriedades de sistemas reativos cujo número de estados é finito, um verificador de modelos pode ser comparado como sendo um autômato finito que aceita palavras infinitas, pois a execução de sistemas deste tipo não pára. Um autômato que se comporta dessa forma é denominado autômato de Büchi. Este tipo de autômato é diferente de um autômato finito de estados pelo fato de aceitar palavras de forma infinita. A condição de aceitação somente é satisfeita se um estado final é visitado de forma infinita.

Na lógica LTL, as propriedades podem ser formuladas através de um autômato de Büchi, mas, por causa da complexidade, é construído o autômato da negação da propriedade que se quer verificar, ou seja, verifica-se se uma propriedade indesejada é válida ou não. Para transferir a solução do problema para o domínio da teoria dos autômatos é necessário ter tanto o modelo do sistema quanto as propriedades formalizadas em forma de máquina de estados. Utilizando esta estratégia o problema se limita a determinar se a linguagem reconhecida pelo autômato da propriedade é uma sub-linguagem do autômato que modela o sistema. O processo é feito através da operação produto síncrono entre autômatos de palavras infinitas. Caso o resultado desta

operação seja vazio (autômato só reconhece palavras vazias), conclui-se que a propriedade se verifica para o modelo. De forma resumida, esta é uma visão geral do processo de verificação de modelos na lógica LTL.

2.1.3 - CTL

A lógica LTL considera que, há somente um único estado sucessor, ou seja, um único futuro possível, a cada momento do tempo. Uma nova lógica foi proposta capaz de considerar diferentes futuros possíveis, usando a noção de tempo ramificado. A idéia principal desta lógica é quantificar todas as possíveis execuções de um programa através da noção de caminhos que existem no espaço de estados do sistema. A partir disso, as propriedades podem ser avaliadas em relação a alguma execução ou então em relação a todas as execuções. Atualmente, existem várias formalizações para lógica temporal ramificada, cada uma com expressividade diferente. É uma lógica utilizada em vários verificadores de modelos.

Utilizando a lógica CTL é possível descrever a evolução de uma MEF como uma árvore infinita, onde os nós são os estados da MEF e as arestas correspondem às transições entre estados. A estrutura de árvore deve-se ao não-determinismo possível nas definições das transições. Os caminhos na árvore que iniciam em um determinado estado são as possíveis evoluções da MEF a partir daquele estado. Em CTL é possível expressar propriedades que devem ser satisfeitas para todos os caminhos iniciando num estado, bem como propriedades que devem ser satisfeitas apenas para algum dos caminhos.

2.1.4 - VERIFICANDO MODELOS NA LÓGICA CTL

Um verificador de modelos é uma ferramenta que automatiza a formulação acima. O processo de validação segue os três passos abaixo:

1. Especificar em CTL quais são as propriedades que o sistema deverá ter para que seja considerado correto. Por exemplo, podemos querer que o sistema nunca entre em deadlock, ou ainda, que ele sempre alcance um determinado estado.

2. O segundo passo é a construção do modelo formal do sistema, que é definido, geralmente, em uma linguagem de alto nível (a linguagem do verificador). O modelo deve capturar todas as propriedades essenciais do sistema para verificar a correção do mesmo, contudo, também deverá possuir abstrações de detalhes do sistema que não afetem a correção das propriedades a serem verificadas.

3. O terceiro e último passo é a própria execução do verificador de modelos para validar as propriedades especificadas do sistema. Neste passo, já temos as propriedades e o modelo. Assim, aplicamos o verificador e conseguimos garantir se o modelo do sistema possui ou não as propriedades desejadas. Caso todas as propriedades sejam verdadeiras, então o sistema está correto. Caso não obedeça a alguma propriedade, então é gerado um contra exemplo mostrando o porquê da não verificação da propriedade. Desta forma, podemos detectar o erro e realizar a correção do modelo. Esse processo deve ser feito até que o sistema obedeça todas as propriedades, realizando assim um ajuste na especificação.

2.1.5 - LTL x CTL

Existe uma discussão sobre a melhor lógica para expressar propriedades, LTL ou CTL. Contudo, as propriedades usualmente utilizadas na verificação de tais sistemas podem ser expressas nas duas lógicas.

O quantificador existencial (E) foi incluso na lógica CTL, mas isso não faz com que a mesma tenha um poder de expressividade maior do que a lógica LTL. Inclusive o poder de expressividade da lógica LTL não é um subconjunto do poder de expressividade da lógica CTL. As expressividades de LTL e CTL são incomparáveis. Na Figura 2.2, é ilustrada uma relação entre os poderes de expressividade de LTL e de CTL.

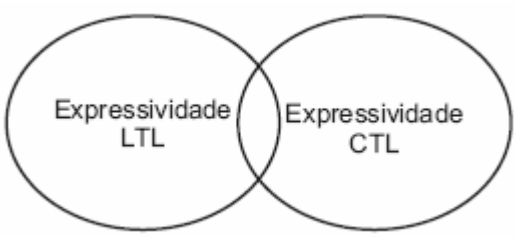

Figura 2.2 - Relação entre as lógicas LTL e CTL

.

Como ilustrado na Figura 2.2, propriedades podem ser descritas em uma lógica e não podem na outra, e vice-versa. Propriedades existenciais não podem ser expressas na lógica LTL. Este tipo de propriedade é muito útil na procura de possíveis deadlocks em um sistema. Já a lógica CTL, não é capaz de expressar algumas propriedades de razoabilidade (fairness).

Cada uma destas lógicas é usada em situações diferentes, pois o uso de uma lógica ou da outra depende do tipo de propriedade que se quer verificar.

2.2 - TIPOS DE PROPRIEDADES

Um número grande de requisitos pode ser associado a um determinado sistema que será submetido a uma verificação. Muitos destes requisitos são muito comuns, como especificar que um determinado sistema não execute uma determinada ação indesejada ou ainda que se consiga chegar a um determinado ponto em que o sistema não pode mais continuar a executar. O fato de alguns tipos de requisitos serem utilizados com bastante freqüência fez com que houvesse a necessidade de determinação de categorias que os definisse de uma forma precisa. Esta categorização é importante pelos seguintes motivos [11]:

- Dado um determinado modelo a ser verificado, é possível começar o estudo do problema com o questionamento de quais devem ser as especificações de segurança, atingibilidade, vivacidade e outras que devem ser consideradas. Com isso, melhores condições para organizar a verificação do modelo são criadas.

- Há diferentes técnicas que possuem indicações que podem variar de acordo com as diferentes categorias de especificações a serem verificadas. Uma divisão em diferentes categorias permite com que, se identifique de maneira mais fácil, as técnicas disponíveis de verificação para cada uma das especificações.

- A categorização das propriedades cria um padrão para a linguagem, facilitando a comunicação entre as pessoas envolvidas.

A seguir é apresentada a categorização de algumas das propriedades mais comuns a que pode estar submetido um sistema [11].

2.2.1 - PROPRIEDADES DE SEGURANÇA

Uma propriedade de segurança especifica que um determinado evento nunca deve ocorrer, dadas certas condições. Um exemplo de propriedade de segurança é:

- Um sistema de intertravamento metroviário jamais deve permitir que um trem entre em uma região que já se encontra ocupada por outro trem;

É possível fazer a associação de uma linguagem a uma determinada propriedade, bastando considerar o conjunto formado por todas as possíveis palavras infinitas que obedeçam as regras impostas pela propriedade em questão.

Em qualquer sistema, a violação de uma propriedade de segurança tem que permitir que a mesma possa ser observada imediatamente, independentemente do comportamento do sistema no futuro. Assim, é recomendado que as propriedades de segurança de um sistema possam ser definidas a partir dos primeiros estados do caminho seguido pelo sistema e não precise levar em consideração o caminho infinito que possa vir a ser seguido.

Na prática, a maior parte das ferramentas permite a definição de especificações que se valem apenas de operadores futuros, de modo que é

mais comum trabalhar apenas com este tipo de operadores. Propriedades de segurança podem ser representadas em lógica temporal CTL ou LTL [11].

2.2.2 - PROPRIEDADES DE VIVACIDADE

Uma propriedade de vivacidade especifica que algum evento desejado do modelo deve ocorrer, dadas certas condições. Exemplos de propriedades de vivacidade poderiam ser os seguintes:

- Um trem que solicita rota deve ser atendido;

Da mesma maneira que para as propriedades de segurança, é possível definir formalmente as características que devem ser seguidas por uma propriedade para que ela possa ser considerada uma propriedade de vivacidade. No caso da vivacidade, é importante relatar que se considera que o evento desejado sempre ocorrerá, independentemente da ocorrência de outros eventos associados ao modelo.

A grande maioria das propriedades de vivacidade podem ser caracterizadas facilmente como tais a partir da própria definição de propriedade de vivacidade [11].

2.2.3 - PROPRIEDADES DE ATINGIBILIDADE

Uma propriedade de atingibilidade (reachability) especifica que uma determinada situação pode ser atingida. Alguns exemplos são apresentados abaixo:

- Será possível fazer o retorno para voltar ao local de origem;

As propriedades de atingibilidade podem ser facilmente representadas através de fórmulas CTL. Para expressar que um acontecimento pode ocorrer é utilizada a combinação EF. Já a combinação EU indica que um acontecimento pode ocorrer, com o acréscimo da informação que uma outra condição é verdadeira enquanto o acontecimento esperado não ocorre.

Assim, a propriedade do exemplo dado acima pode ser escrita como EF retorno. A propriedade indica que, a partir de seu ponto de origem, o viajante pode escolher um determinado caminho que o leva ao retorno.

Pela definição desta propriedade, chega-se à conclusão que a lógica LTL não possui meios para expressar propriedades de atingibilidade, pois não consegue exprimir os diferentes futuros possíveis a partir de um estado inicial. Entretanto, a lógica LTL possui a capacidade de exprimir que algum determinado evento nunca ocorre, o que é justamente a negação de uma propriedade do tipo EFevento. Dispondo-se de um verificador de modelos que aceita apenas fórmulas LTL e desejando-se verificar alguma propriedade deste formato, pode-se fornecer o seu complemento como especificação de entrada. Se a propriedade original for válida, o verificador de modelos vai chegar à conclusão oposta e vice-versa [11].

2.2.4 - PROPRIEDADES DE AUSÊNCIA DE DEADLOCK

Deadlock é uma situação indesejada em qualquer tipo de sistema, sendo representado por um estado no qual nenhum progresso é possível de acontecer, uma situação em que o sistema trava e acontece um bloqueio permanente.

Através da lógica CTL, existe a possibilidade de especificar a propriedade de ausência de deadlock utilizando uma fórmula AGEX. Esta fórmula especifica que, independentemente do estado em que o sistema se encontra, é possível encontrar um caminho tal que haja alguma evolução, que não acontece o bloqueio. Já a lógica LTL não é capaz de especificar propriedades deste tipo de um modo genérico como a CTL [11].

2.2.5 - PROPRIEDADES DE RAZOABILIDADE (FAIRNESS)

Uma propriedade de razoabilidade indica que um evento deve ocorrer um número infinito de vezes, dadas certas condições. Um exemplo deste tipo de propriedade é:

- Um transmissor que tenta enviar suas mensagens um número infinito de vezes consegue sucesso na transmissão em um número infinito de vezes.

A propriedade de razoabilidade é assumida quando se deseja realizar alguma verificação. Por exemplo, no caso da transmissão de mensagens não seria possível verificar o envio de uma mensagem do lado do transmissor até o lado do receptor se a propriedade do exemplo não estivesse sendo assumida, já que sempre existiria um contra-exemplo em que o canal de comunicação estivesse constantemente impedindo a transmissão.

A propriedade do exemplo citado acima indica que um número infinito de envios será acompanhado por um número infinito de recebimentos. Propriedades deste tipo são normalmente denominadas de propriedades fortes de razoabilidade. Não é possível exprimir este tipo de propriedade em lógica CTL [11].

2.3 - VERIFICAÇÃO DE MODELOS NA PRÁTICA

Verificação de modelos provou ser uma tecnologia extremamente bem sucedida para verificar requisitos e projetá-los para uma grande variedade de sistemas, particularmente em sistemas embarcados e para segurança de sistemas críticos de tempo real. Entretanto, há algumas questões a serem consideradas ao usar na prática a verificação de modelos, entre elas algumas dificuldades e algumas estratégias.

As principais dificuldades encontradas na utilização na técnica de verificação de modelos são:

- Escolher uma linguagem de especificação e a utilização de ferramentas de suporte
    - Existem muitas dificuldades de se obter ferramentas que ofereçam suporte na utilização de algumas destas linguagens.
    - A geração do espaço de estados é permitida pela utilização de linguagens formais baseadas em máquinas de estados.
    - Um exemplo é a linguagem de especificação CSP.

- Em um certo nível de abstração, o código-fonte é considerado como uma especificação do programa. Cada vez mais, a técnica de verificação de modelos tem sido aplicada diretamente sobre código-fonte em função de problemas com uso de linguagens de especificação.
- Os verificadores mais usados possuem uma linguagem própria para descrição de modelos.

- Explosão do espaço de estados
  - Problema bastante conhecido na técnica de verificação de modelos. O espaço de estados dos sistemas pode ser muito grande ou até mesmo infinito
  - Registrar todos os comportamentos possíveis de um sistema complexo pode esgotar os recursos de memória de uma máquina, mesmo que o número de estados alcançados pelo sistema seja finito.
  - Muitos trabalhos de pesquisa têm sido realizados neste contexto e há, atualmente, um número considerável de estratégias para tratar deste problema. Algumas estratégias são: representação simbólica do espaço de estados, redução por ordem parcial, etc.
  - A representação simbólica nem sempre pode ser adequadamente aplicada, pois sistemas de software são mais complexos.
  - O problema é agravado com a aplicação da técnica de verificação diretamente sobre o código-fonte

- Especificação de propriedades
  - As duas lógicas mais utilizadas para especificação de propriedades são CTL e LTL
  - Existe uma grande dificuldade de se escrever fórmulas nas lógicas CTL e LTL.
  - Alguns padrões para especificação de propriedades tem sido adotados como uma alternativa

Por causa das dificuldades na utilização da técnica de verificação de modelos, estratégias estão sendo adotadas em diversas situações e com diferentes objetivos:

- Verificação de modelos em código-fonte de programas
- Utilizando linguagens de especificação

As principais estratégias são:

- Verificação de modelos em código-fonte de programas
    - Motivação
        - É uma estratégia bastante interessante para processos que não se preocupam muito com a manutenção de modelos.
        - Identificação de erros que não são inseridos na fase de projeto, mas sim na fase de implementação.
        - A linguagem Java, por exemplo, possui projeto de pesquisa consolidado.
    - Aspectos teóricos
        - Qualquer detalhe do sistema é considerado, pois a verificação é aplicada sobre código-fonte de programas. Isso faz com que a questão da explosão de espaço de estados fique mais complexa e difícil.
        - Técnicas são utilizadas para tentar minimizar os problemas da explosão de estados
    - Problemas
        - Explosão de estados
        - Erros são descobertos tardiamente, pois somente depois que o software está pronto, é que é possível encontrá-los.

- Utilizando linguagens de especificação
    - Motivação
        - Utilização de linguagens com grande poder de expressividade para oferecer cobertura a características como paralelismo e concorrência.

    - Aspectos teóricos
        - Utilizar estas linguagens é utilizar ferramentas disponíveis, pois a geração de espaço de estados é direta do modelo.

    - Problemas
        - Uma quantidade pequena de ferramentas comerciais está disponível.
        - Existe dificuldade em adotar estas linguagens no processo de desenvolvimento
            - O conhecimento destas linguagens é pouco.
            - Existe a necessidade de maior conhecimento teórico por parte dos desenvolvedores que utilizam estas linguagens.

2.4 - ABORDAGENS PARA VERIFICAÇÃO DE MODELOS EM C

Existem maneiras diferentes de se verificar modelos em código C. Abaixo é feita uma descrição breve das quatro maneiras freqüentemente mais usadas. Estas quatro maneiras são usadas para segurar o espaço infinito de estados do programa C. Em todas estas abordagens o programa está de alguma maneira transformado em um programa abstraído que tenha um espaço finito de estados. Isto é requerido porque o algoritmo verificador de modelos tem que visitar todos os estados.São elas:

- Bounded Model Checker (BMC)
- Verificação de modelos com abstração de predicados usando provador de teorema ou um resolvente SAT

- Tradução do código de C em um modelo de um verificador de modelo padrão existente

2.4.1 - BOUNDED MODEL CHECKING

A técnica chamada Bounded Model Checker (BMC), foi proposta primeiramente por Biere em 1999. A motivação original de BMC era o sucesso dos resolventes SAT em resolver fórmulas booleanas para verificação de modelos. Durante os últimos anos houve um aumento tremendo no poder de raciocínio dos resolventes SAT. Agora podem segurar exemplos com centenas de milhares de variáveis e de milhões de cláusulas. Não resolve o problema da complexidade da verificação de modelos, pois ainda confia em um procedimento exponencial e por isso tem capacidade limitada. BMC tem também a desvantagem de não poder provar a ausência de erros, na maioria dos casos reais. Conseqüentemente BMC se junta ao arsenal de ferramentas automáticas de verificação, mas não substitui algumas delas. A idéia básica no BMC é procurar por um contra-exemplo nas execuções, cujo comprimento é limitado por algum inteiro K. Por esta razão é chamado Bounded Model Checking. Se nenhum erro for encontrado então k é incrementado em um até que um erro seja encontrado, o problema torne-se intratável, ou algum limite superior preestabelecido seja alcançado. O problema de BMC pode eficientemente ser reduzido a um problema de satisfatibilidade da proposição, e pode conseqüentemente ser resolvido por métodos SAT. Os resolventes SAT modernos podem segurar problemas da satisfatibilidade da proposição com centenas de milhares de variáveis ou mais. Um exemplo de ferramenta Bounded Model Checker para a linguagem de programação C é o CBMC.

2.4.2 - ABSTRAÇÃO DE PREDICADOS

A eficácia da verificação de modelos de sistemas é comprometida severamente pelo problema da explosão do espaço de estados, e muita da pesquisa nesta área objetiva reduzir o espaço de estados do modelo usado para a verificação. Um método principal utilizado para redução do espaço do estado de sistemas de software é abstração. As técnicas de abstração

reduzem o espaço de estados do programa traçando o conjunto dos estados do sistema a um sumário e de uma maneira que preserve os comportamentos relevantes do sistema. As abstrações são executadas freqüentemente de uma maneira mais informal, manual, e requerem um cuidado considerável. A abstração de predicados é um dos métodos mais populares e mais aplicados para a abstração sistemática dos programas. Abstrai dados mantendo-se um par de determinados predicados nos dados. Cada predicado está representado por uma variável booleana no programa abstrato, quando as variáveis originais dos dados forem eliminadas. A verificação de um sistema de software com abstração de predicado consiste em construir e avaliar um sistema de estados finito, que seja uma abstração do sistema original com respeito a um conjunto de predicados.

O programa abstrato é criado usando abstração existencial. Este método define a relação de transição do programa abstrato de modo que se garanta ser uma aproximação conservadora do programa original, com respeito ao conjunto de predicados dados. O uso de uma abstração conservadora, ao contrário de uma abstração exata, produz reduções consideráveis no espaço de estados. O inconveniente da abstração conservadora é que, quando a verificação de modelo do programa abstrato falha, pode-se produzir um contra-exemplo que não corresponde a um contra-exemplo concreto, ou seja, um contra-exemplo falso. Isto é chamado geralmente um contra-exemplo “spurious”. Quando um contra-exemplo “spurious” é encontrado, o refinamento é executado ajustando o conjunto dos predicados de uma maneira que elimine este contra-exemplo. O processo de refinamento da abstração foi automatizado pelo paradigma do refinamento da abstração do contra-exemplo guiado ou CEGAR de forma abreviada.

As etapas são descritas abaixo no contexto da abstração de predicados.

1. Abstração do programa. Dado um conjunto de predicados, um modelo de estados finito é extraído do código de um sistema de software e o sistema abstrato da transição é construído.

2. Verificação. Um algoritmo de verificação de modelos é rodado a fim de verificar se o modelo criado aplicando-se a abstração de predicados

satisfaz o comportamento desejado. Se sim, o verificador de modelos relata sucesso e o “loop” do CEGAR termina. Se não, o verificador de modelos extrai um contra-exemplo e a computação prossegue à etapa seguinte.

3. Validação do Contra-exemplo. O contra-exemplo é examinado para determinar se ele é “spurious”. Isto é feito simulando o programa usando o contra-exemplo abstrato como um guia, para saber se o contra-exemplo representa um comportamento real do programa. Se este for o caso, o erro está relatado e o “loop” do CEGAR termina. Se não, o “loop” do CEGAR prossegue à etapa seguinte.

4. Refinamento de predicado. O conjunto dos predicados é mudado a fim de eliminar o contra-exemplo “spurious” detectado, e possivelmente outros comportamentos “spurious” introduzidos pela abstração de predicado. Dado o conjunto atualizado dos predicados, o “loop” do CEGAR prossegue a etapa 1.

A eficiência deste processo é dependente da eficiência dos procedimentos do refinamento da abstração e do predicado do programa. Quando a abstração do programa focalizar em construir a relação da transição do programa abstrato, o foco do refinamento do predicado é definir técnicas eficientes para escolher o conjunto dos predicados de uma maneira que elimine contra-exemplos “spurious”. Em ambas as áreas de pesquisa, o custo computacional baixo é um fator chave desde que permita a aplicação da verificação de modelos em programas reais.

A diferença entre a verificação de modelos com abstração de predicado usando um provador de teorema e usando um resolvente SAT é a maneira com que o programa booleano é construído. Um verificador de modelos que usa um provador de teorema constrói o programa booleano chamando um provador de teorema repetidamente. Já o que usa um resolvente SAT faz somente uma chamada ao resolvente SAT. Nesta chamada o resolvente SAT computa a abstração das relações concretas da transição.

A vantagem de se utilizar um resolvente SAT é que o número exponencial de chamadas do provador de teorema está eliminado, e depois de

instanciadas, as atribuições possíveis aos valores dos predicados são procuradas pelo resolvente SAT. Os resolventes SAT modernos são altamente eficientes e permitem um grande número de variáveis. Isto permite verificar muitas mais atribuições possíveis, tendo por resultado uma relação abstrata mais precisa da transição e a eliminação de contra-exemplos “spurious” redundantes.

Uma outra vantagem é que a maioria das construções de C podem ser codificadas usando FNC (Forma Normal Conjuntiva), que permite uma escala mais larga dos programas. Os operadores de inteiro são codificados usando operadores do vetor de bit, isto é, faz exame no cliente do excesso aritmético. Assim, não há nenhuma resposta positiva falsa devido à suposição inexata que a escala dos valores das variáveis é infinita. Além disso, as construções da manipulação do ponteiro, incluindo a aritmética de ponteiro, podem também ser suportadas. A única limitação é que a recursão e a alocação de memória dinâmica não estão permitidas. Esta limitação não pode ser evitada, desde que o programa booleano requer ser finito.

A geração de programas booleanos abstratos de um programa em C e um conjunto de predicados sofrem de alguns problemas:

1. A geração do programa booleano é feita chamando um provador de teoremas para cada atribuição aos predicados atuais e seguintes do estado. Para uma relação mais precisa da transição, isto requer um número exponencial das chamadas do provador de teorema. Diversas heurísticas são usadas para reduzir este número. Algumas ferramentas existentes evitam este número grande de chamadas do provador de teoremas. Depois que este número especificado é alcançado, a ferramenta adiciona todas as transições restantes para que a chamada do provador de teorema seja saltada, rendendo um número grande de contra-exemplos “spurious” desnecessários.

2. Trabalhos existentes confiam em provadores de teorema de uso geral. As variáveis do programa são modeladas como os valores ilimitados do inteiro, negligenciando o excesso aritmético possível no programa C. Isto pode resultar em respostas positivas falsas da ferramenta.

3. As ferramentas existentes suportam somente uma escala muito limitada dos operadores, como, operadores booleanos, adição/subtração, igualdade e operadores relacionais. Outros operadores de C, como a multiplicação e a divisão, os operadores bit-wise, os tipos de operadores de conversão, e os operadores de deslocamento são modelados por meio das funções não interpretadas. Isto limita o conjunto dos programas e das propriedades que podem ser verificadas.

4. As ferramentas existentes fornecem somente uma sustentação limitada para operações do ponteiro. No detalhe, a aritmética do ponteiro não é suportada.

Os exemplos para os verificadores de modelos que usam um provador de teorema são: BLAST, SLAM, MOPS e MAGIC. Um exemplo para um verificador de modelos que usa um resolvente SAT é o SatAbs.

2.4.3 - MIGRAÇÃO DE C PARA OUTRO MODELO

Nesta abordagem, o código de C é transformado em um modelo usado por um verificador modelo de uso geral. Isto tem a vantagem que estes verificadores de modelos estão extensamente espalhados e possuem algoritmos eficientes. Mas tem a desvantagem que todo o conhecimento especial do código em C e de hardware tem que ser usado no processo de abstração, porque estes verificadores de modelo não estão cientes desta informação. Os diferentes verificadores de modelo, que usam esta abordagem, fazem o uso de todas as abstrações de C para gerar os modelos que são finitos. FeaVer e FocusCheck são dois verificadores de modelo que usam esta abordagem. FeaVer transforma o código C no código de Promela. Este código de Promela é verificado então com o verificador de modelo Spin. FocusCheck transforma o código C no Prolog de XSB. Do Prolog de XSB um sistema de transição é gerado e o modelo é verificado. As ferramentas que utilizam esta abordagem não serão detalhadas, porque não fazem parte do contexto deste trabalho.

3 - FERRAMENTAS PARA A LINGUAGEM C

Ferramentas verificadoras de modelos, a partir de código-fonte, são bastante úteis para o desenvolvimento de software, pois sistemas críticos necessitam de garantia de funcionamento perfeito, mas os prazos de entrega estão cada vez menores devido à pressão econômica. Então, para tornar mais fácil à tarefa de verificação de modelos de sistemas, essas ferramentas estão, cada vez mais, sendo utilizadas, pois elas conseguem analisar sistemas sem a necessidade de migrar a implementação para uma linguagem de especificação.

Por tudo isso, novas ferramentas estão surgindo, as existentes estão sendo melhoradas e várias pesquisas na área estão sendo realizadas. O funcionamento de algumas das principais ferramentas de verificação de modelos para a linguagem de programação C é descrito abaixo, tendo algumas delas uma descrição mais detalhada e outras menos devido à disponibilidade de documentação das mesmas.

3.1 - SLAM

As ferramentas verificadoras de modelos foram estudadas academicamente algumas vezes, mas não foram consideradas uma solução praticável pela indústria de software. O SLAM gerou muito interesse da indústria de software quando aplicou com sucesso estes conceitos aos projetos industriais. Além disso, desde que o SLAM foi desenvolvido pela Microsoft Research, que é agora uma parte de um produto comercial (SDV (Static Driver Verifier) do Windows) [19], a indústria começou a mostrar alguma confiança em técnicas formais de verificação de software. Entretanto, por ser uma parte do SDV, o SLAM, no momento, não está disponível publicamente. Assim toda a análise feita é baseada na literatura disponível [1].

O objetivo principal do SLAM é verificar programas em C para propriedades temporais de segurança usando verificação de modelos. Seu principal domínio da aplicação tem sido Device Drivers do Windows. O SLAM tem três componentes principais: c2bp (que avalia um abstração booleana do

programa), bebop (que executa a análise da atingibilidade de programas booleanos) e newton (que verifica a praticidade dos caminhos de erros). O SLAM usa o zapato como seu provador de teoremas.

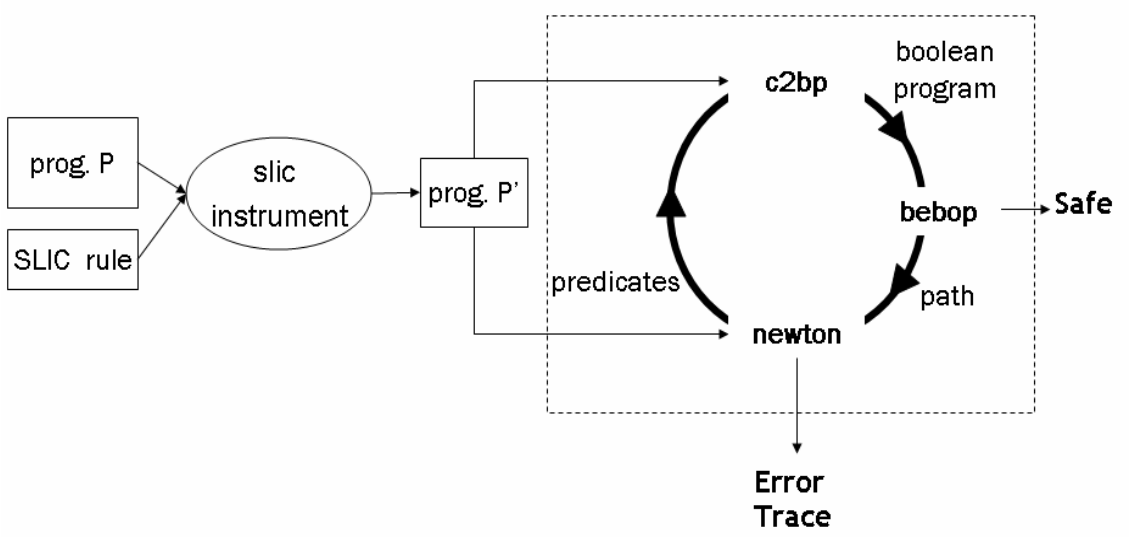

Figura 3.1 - Funcionamento do SLAM

A língua da especificação usada no SLAM é SLIC (língua da especificação para verificação de interface). As propriedades de segurança a serem verificadas são especificadas no SLIC. As especificações são descritas como máquinas de estados com um conjunto estático de variáveis de estados e um conjunto de eventos e transições de estados nos eventos. O “slic instrument” funde o programa P e o código em C para a especificação SLIC para formar um único programa P'. Isto é mostrado na Figura 3.1. Então, a análise é feita no programa instrumentado P'. Todo o caminho do erro alcançável em P' é também alcançável em P.

A Figura 3.1 mostra os diferentes componentes do SLAM e como eles interagem. C2bp faz o exame de uma entrada com um programa em C(o programa instrumentado P') e um conjunto de predicados (as especificações SLIC para a primeira iteração) e convertem-nos a um programa booleano. Um programa booleano é um programa em que todas as variáveis são do tipo booleano. Em seguida, o bebop executa uma análise da atingibilidade (reachability) no programa booleano criado por c2bp para verificar se o nome do erro esta alcançável em P'. Se tal caminho for encontrado, o newton verifica

para ver se este caminho é praticável no programa C original. Se o newton encontrar que o caminho é praticável, um erro foi encontrado e o SLAM dá o caminho do erro como saída. Se não, o newton encontra os predicados adicionais que explica a não praticidade. O newton usa as mesmas interfaces para os provadores de teoremas que o c2bp usa para sua análise.

Computar uma expressão booleana precisamente (c2bp) é muito caro. Depende do número de chamadas ao provador de teorema e é linear no tamanho do programa P e exponencial no tamanho do conjunto dos predicados E. O algoritmo do c2bp executa a abstração modular do programa quando cria um programa booleano. Abstrai cada procedimento e dentro de cada procedimento, abstrai cada indicação. Assim não há nenhuma necessidade do c2bp fazer algum controle no fluxo da análise.

Bebop é o verificador de modelos para programas booleanos. É componente do SLAM. Os estados são representados como vetores de bits, assim o conjunto de estados alcançáveis é representado como um conjunto de vetores de bits. Este conjunto de vetores de bits é computado para cada estado. Os estados são representados implicitamente através de BDDs.

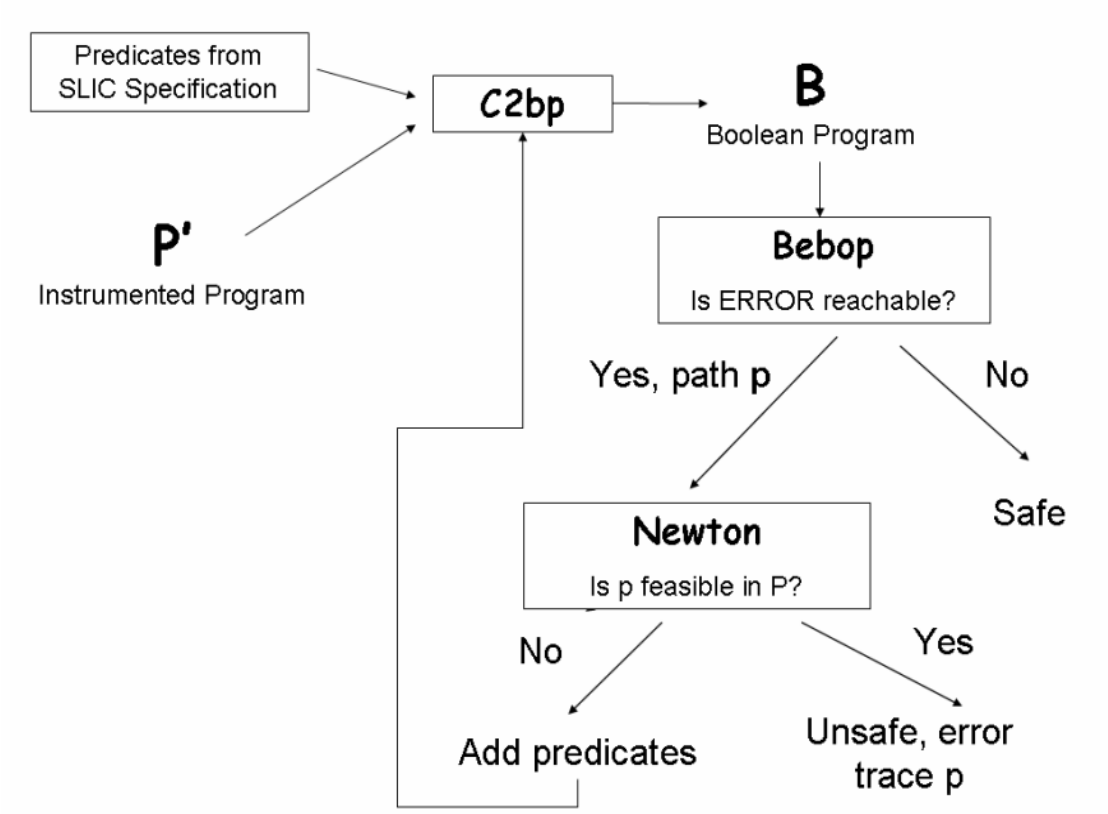

Figura 3.2 - Componentes do SLAM

Quando um caminho p do erro é fornecido pelo bebop, o SLAM usa primeiramente o newton para simular simbolicamente o caminho inteiro e determinar se é “spurious”. Se for “spurious”, então o newton procurará pelos predicados adicionais que poderiam eliminar o caminho em uma abstração refinada. Se nenhum predicado novo for encontrado, o SLAM conclui que o caminho “spurious” foi causado pela abstração imprecisa feita pelo c2bp. Invoca então um outro método de refinamento, chamado “constrain”. “Constrain” examina simbolicamente cada etapa do caminho dentro da isolação e faz tentativas para refinar a relação abstrata da transição a fim melhorar a exatidão da abstração usando os predicados que estão disponíveis. O newton e o constrain usam o zapato como provador de teorema [1].

3.2 - BLAST

O BLAST (ferramenta de verificação de software de abstração “preguiçosa”) foi desenvolvido na universidade da Califórnia, Berkeley. O conceito básico por trás do BLAST é uma abordagem com abstração, verificação e refinamento que é seguida também no SLAM. Adicionalmente, o BLAST usa conceitos de abstração “preguiçosa” e descoberta de predicados baseado em interpolação.

A metodologia é similar ao SLAM como mostrado na Figura 3.3. As entradas do BLAST são programas em C e a especificação da propriedade a ser verificada. spec.opt une as entradas e dá forma ao programa instrumentado que contém um nome para o estado do erro. Este programa instrumentado alimenta então o pblast.opt que é o verificador de modelos do BLAST. Se não houver nenhum caminho ao nome especificado do erro, o BLAST considera o sistema seguro e gera uma prova. Se não, verifica se este caminho é praticável usando a execução simbólica do programa. Se o caminho for praticável, gera um caminho como saída. Se não, o modelo é refinado.

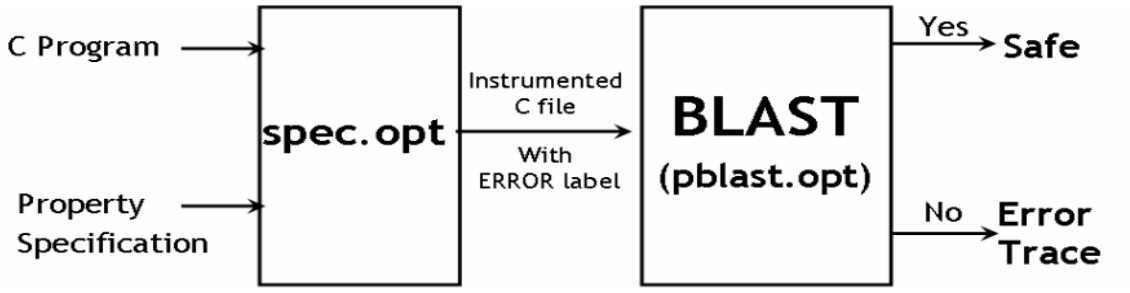

Figura 3.3 - Funcionamento do BLAST

A língua da especificação do BLAST tem muito código em C na sintaxe. Isto o faz mais fácil de aprender, especialmente para programadores em linguagem C, isso comparando com aprender uma nova linguagem de especificação.

A arquitetura do BLAST é mostrada na Figura 3.4. A ferramenta é escrita em OCaml. Usa CIL (CCured) para verificar o tipo de segurança do programa em C. CCured introduz verificações necessárias em tempo de execução para impedir todas as violações de segurança da memória. O programa é representado internamente como fluxo de controle dos autômatos (CFA). Um CFA é um gráfico dirigido, seus vértices correspondem aos pontos de controle do programa e suas bordas correspondem às operações do programa. Cada borda é nomeada por um bloco das instruções que são executadas ao longo dessa borda, ou por um suposto predicado. O suposto predicado representa a circunstância que deve prender para que a transição ocorra. A abstração é construída durante a execução e representada como uma árvore abstrata de Atingibilidade(Reachability). O BLAST usa “Simplify” e “vampyre” como provadores de teorema.

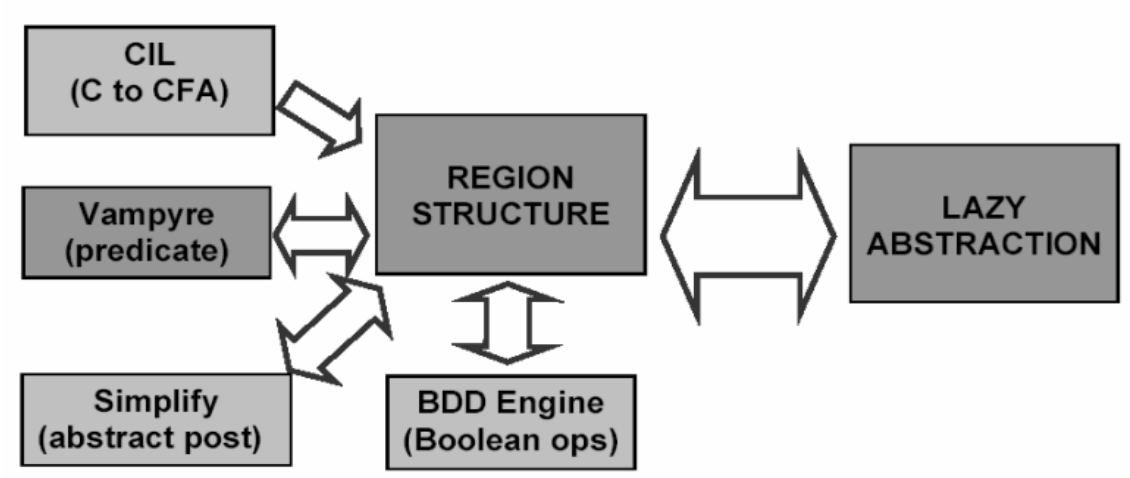

Figura 3.4 - Arquitetura do BLAST

A abordagem com abstração, verificação e refinamento (usada também no SLAM) tem três fases. Na fase “abstrata”, um conjunto de predicados é escolhido para criar uma abstração do programa. Cada estado abstraído pode ser representado pelo conjunto de atribuições verdades dos predicados. Na fase de “verificação”, esta abstração é usada para verificar a propriedade de segurança. Se o modelo abstraído for seguro, o programa original é considerado seguro. Se não, um caminho do erro é gerado e verifica-se se este caminho é praticável no programa original. Se o caminho do erro corresponder a um erro “spurious”, na fase de “refinamento”, o modelo abstraído é refinado adicionando mais predicados. No BLAST, as três fases são integradas firmemente usando a abstração “preguiçosa” como mostrado na Figura 3.5. A fase de verificação dirige a fase abstrata.

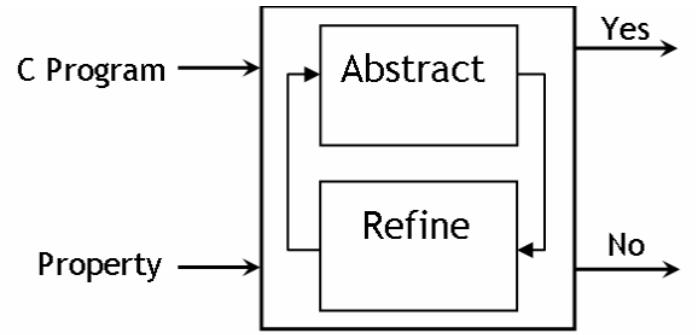

Figura 3.5 - Funcionamento do BLAST

Há dois princípios envolvidos na abstração “preguiçosa”: abstração precisa e refinamento sob-demanda. A abstração precisa é baseada na observação de que ter o mesmo nível de precisão não é aconselhável para todas as regiões no espaço de estados enquanto algumas regiões podem ser inalcançáveis. Na abstração “preguiçosa”, em vez de abstrair o modelo abstrato inteiro na fase abstrata, as regiões são abstraídas somente quando são necessitadas pela fase de verificação. O refinamento sob-demanda é baseado na observação que após o refinamento, o modelo usado na iteração precedente pode ser reusado. No SLAM, o modelo inteiro é construído após o refinamento. Na abstração “preguiçosa”, as regiões no espaço de estados que têm sido provadas serem seguras não são refinadas outra vez. Na fase de refinamento, o mesmo é feito começando pelo estado mais adiantado, em que

o ponto de controle no caminho do erro (baseado na abstração) não tem um ponto correspondente no programa original. Este estado é chamado de estado pivô [2].

O algoritmo da abstração “preguiçosa” é composto de duas fases: a fase de busca e de análise do contra-exemplo. Uma árvore abstrata de atingibilidade (reachability) é construída durante a execução. Representa uma parcela alcançável do espaço de estados abstrato do programa. Cada ponto de controle no programa pode ser representado como um estado. Um conjunto de estados pode ser abstraído como uma região. Se um caminho à região do erro for encontrado, pode ser o caso de a região ter alguns estados de erro, mas que não são alcançáveis no programa original. A abstração é refinada então para que essa região encontre como saída se o caminho do erro é praticável no programa original.

Na fase de busca, a árvore abstrata de atingibilidade (reachability) é construída de forma incremental. Cada caminho na árvore corresponde a um caminho no CFA. Cada nó da árvore é nomeado por um vértice do CFA e de uma fórmula, chamada de região alcançável. Cada borda é nomeada por um bloco das instruções ou por um suposto predicado. A região alcançável é uma combinação booleana dos predicados da abstração. Ela representa o que se sabe sobre o estado do programa nos termos dos predicados sob a consideração, após ter executado as instruções do nó da raiz ao nó atual. A região alcançável de um nó é obtida da região alcançável do nó pai e das instruções na borda do nó pai ao nó atual. Se um nó do erro for alcançável na árvore, então a etapa seguinte é a análise do contra-exemplo.

Na análise do contra-exemplo, um provador de teorema é chamado, para verificar se o erro é real ou resulta de uma abstração grosseira. Se for um erro “spurious”, o provador de teorema sugere os predicados novos da abstração. O programa é refinado localmente adicionando os predicados novos da abstração somente na sub-árvore menor que contém o erro “spurious”. A busca continua do ponto que é refinado (o estado pivô). Iterando sobre as duas fases de busca e da análise do contra-exemplo, parcelas diferentes da árvore de atingibilidade(reachability) vão usando conjuntos diferentes de predicados da abstração [2].

Quando um caminho do erro é gerado, se faz uma simulação para verificar se o caminho é praticável no programa original. O BLAST usa a descoberta de predicados baseada em interpolação para descobrir predicados novos. Os predicados em todo o nó (ponto de controle no CFA), podem ser computados interpolando o nó precedente e o nó seguinte no CFA entre os predicados. A descoberta de predicados baseada em interpolação encontra o conjunto mínimo dos predicados necessários em um nó (ponto de controle no CFA) para ir do nó precedente ao nó seguinte. A vantagem de se usar este método é que ele reduz a exigência de armazenamento enquanto menos predicados são armazenados em cada nó.

3.3 - MOPS

O MOPS (MOdel checking Programs for Security properties) é uma ferramenta de análise estática (tempo de compilação) verificadora de modelos para software com aplicações críticas de segurança. Dado um programa e uma propriedade de segurança, o MOPS verifica se o programa pode violar a propriedade de segurança. As propriedades de segurança que o MOPS verifica são propriedades de segurança temporais, isto é, propriedades que requerem que os programas executem determinadas operações relevantes de segurança em determinadas ordens [7]. O usuário do MOPS descreve uma propriedade de segurança por um autômato de estados finito (FSA). Um FSA que descreva uma propriedade de segurança pode ser chamado de um modelo de segurança. O programa é modelado como um autômato Pushdown (PDA). O MOPS verifica se um programa pode violar uma propriedade de segurança temporal usando um verificador de modelos Pushdown. Verificação de modelos Pushdown permite procurar caminhos interprocessuais. A verificação de modelos é usada para determinar se determinados estados que representam a violação da propriedade de segurança no FSA são alcançáveis no PDA. Se o MOPS encontrar violações, relata “traces” do erro, ou seja, o caminho do programa que causou tais violações [10].

Por se importar somente com todos os caminhos praticáveis em um programa e nas declarações executadas nestes caminhos, a execução do programa pode ser modelada por um ponteiro e por uma pilha. O ponteiro

aponta à posição do programa da declaração seguinte a ser executada, e aos registros da pilha os endereços do retorno de todas as ligações de controle não terminadas. Conseqüentemente, o valor do ponteiro e os valores na pilha identificam excepcionalmente um instante do programa na execução. Se nós fundirmos o ponteiro e a pilha considerando o ponteiro como o elemento superior na pilha, é obtido um autômato Pushdown (PDA). O fluxo do controle no programa determina as transições no PDA. Uma vez que um FSA descreve uma propriedade de segurança e um PDA representa um programa, o objetivo é verificar se qualquer estado de risco no FSA é alcançável em algum ponto do programa no PDA. Para responder a esta pergunta, o MOPS compõem o FSA M com o PDA P. Isto resulta em um novo PDA, chamado PDA composto. A configuração inicial do PDA composto representa o instante que o programa começa, onde o estado do PDA está no estado inicial do modelo de segurança e da pilha do PDA que contém somente o ponto de entrada do programa. O MOPS pode determinar se qualquer estado de risco é alcançável dentro do PDA composto. Se este for o caso, o MOPS tem encontrado uma violação potencial de segurança e então fornece como saída um caminho da execução no programa que causou esta violação. Além disso, o MOPS pode determinar, para cada declaração em um programa, todos os estados em um FSA que a declaração pode ser executada [10].

Uma característica importante do MOPS é que permite que as propriedades complexas de segurança sejam decompostas em modelos mais simples de segurança que são mais fáceis de descrever. O MOPS pode combinar estes modelos mais simples em um modelo complexo.

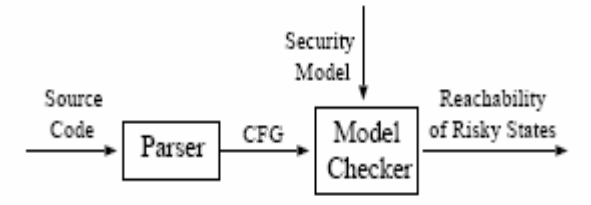

Figura 3.6 - Funcionamento do MOPS

O MOPS consiste em um “parser” e em um verificador de modelos, como mostrado na Figura 3.6. O MOPS verifica se um código-fonte satisfaz a uma propriedade de segurança pelas seguintes etapas: Primeiramente, o

parser constrói um gráfico do fluxo de controle (CFG) do programa e então, o verificador de modelos constrói um PDA do CFG e verifica se o PDA vai de encontro à propriedade de segurança.

- Parser - O “parser” constrói um CFG de um programa fonte. Cada borda no CFG representa uma declaração no programa por uma árvore de sintaxe abstrata (AST), e cada nó no CFG representa um ponto do programa. O parser é baseado no GCC. Usando um “parser” derivado do GCC, o MOPS pode analisar gramaticalmente todo o programa fonte que o GCC analisar gramaticalmente. Se o programa consistir em múltiplos arquivos fontes, o MOPS faz a união dos múltiplos CFGs, cada qual é gerado de um arquivo fonte em um único CFG. Para programas fonte grandes, os tamanhos de seus CFGs aumentam rapidamente. A maioria dos verificadores de modelos não pode suportar um CFG tão grande. Conseqüentemente, existe uma maneira de comprimir drasticamente CFGs grandes. Comprimir um CFG não deve introduzir nenhuma imprecisão à análise.

- Verificador de modelos - Fazendo exame de um CFG gerado pelo “parser” e de um modelo de segurança representado por um FSA, o verificador de modelos decide se o programa do CFG pode violar a propriedade de segurança em quatro etapas. Primeiramente, constrói um PDA para o CFG adicionando uma transição ao PDA para cada borda no CFG. Em segundo, o verificador de modelos faz a computação da interseção do modelo de segurança com o programa PDA fazendo exame de sua composição paralela, que cria um PDA novo (chamado o PDA composto), onde os estados vêm do FSA e os símbolos de entrada e símbolos da pilha vêm do PDA. Em terceiro lugar, o verificador de modelos computa todas as configurações alcançáveis da configuração inicial do PDA composto. O conjunto de configurações alcançáveis poderia ser muito grande, ou mesmo infinito, por isso, representá-lo é um desafio.

Quando o verificador de modelos determina que um programa pode violar uma propriedade de segurança, é útil identificar um caminho do programa em que a violação ocorre. Intuitivamente, desde que o símbolo de entrada em uma borda no PA representa um ponto do programa, a lista dos predecessores da borda grava todos os pontos do programa que podem imediatamente preceder o atual em um “trace” do erro. Conseqüentemente, uma vez encontrada uma configuração alcançável do PDA que viole a propriedade de segurança, pode-se recuperar todas suas configurações precedentes uma por uma [10].

3.4 - CBMC

CBMC é um Bounded Model Checker (verificador de modelos limitado) usado para a análise de programas em C. Traduz programas de C para a lógica proposicional. Loops e recursão são suportados. CBMC suporta a aritmética de ponteiro, operadores de inteiro, casts de tipo, chamadas de funções, chamadas através de ponteiros de função, não-determinismo, suposições, afirmações, estruturas, uniões nomeadas e memória dinâmica. As propriedades verificadas incluem também a segurança do ponteiro, limites de arrays e afirmações fornecidas pelo usuário. CBMC permite que os programas que empregam a alocação dinâmica de memória, por exemplo, definam o tamanho de um array dinamicamente ou estruturas de dados como listas ou gráficos. Conseqüentemente esta ferramenta é uma boa escolha para analisar o código de sistemas escritos em C que empregam estas características.

A ferramenta executa uma técnica chamada verificação de modelos limitada (BMC). A idéia básica no BMC é procurar por um contra-exemplo nas execuções, cujo comprimento é limitado por algum inteiro K. Por esta razão é chamado Bounded Model Checking. Se nenhum erro for encontrado então k é incrementado em um até que um erro seja encontrado, ou que o problema torne-se intratável, ou ainda se algum limite superior preestabelecido seja alcançado. O problema de BMC pode eficientemente ser reduzido a um problema de satisfatibilidade da proposição, e pode conseqüentemente ser resolvido por métodos SAT. Os resolventes SAT modernos podem segurar problemas da satisfatibilidade da proposição com centenas de milhares de

variáveis ou mais. Em BMC, as relações da transição para uma máquina de estados complexa e sua especificação são desenvolvidas conjuntamente para obter uma fórmula booleana que seja satisfatível se existir um caminho do erro. A fórmula é verificada então usando um procedimento SAT. Se a fórmula for satisfatível, um contra-exemplo é extraído da saída do procedimento SAT. Na maioria dos casos CBMC pode determinar o limite superior N. Se falhar, o usuário pode então fornecer um limite superior para que seja usado pelo CBMC. Se o usuário fornecer este limite superior, não se pode garantir que não há nenhum contra-exemplo que é mais longo do que o limite superior. Neste caso CMBC pode somente ser usado como uma ferramenta para encontrar erros e não provar a exatidão, desde que os erros podem passar despercebidos [8].

A ferramenta vem com uma interface gráfica (GUI) que esconde do usuário os detalhes da execução. Se um contra-exemplo for encontrado, a GUI permite um passo a passo do trace, como um debugger.

O domínio da aplicação desta ferramenta inclui softwares embarcados e de simulação de protótipos.

3.5 - SATABS

O SatAbs é uma ferramenta de verificação de modelos para a linguagem C. C é um das linguagens de programação mais populares, com grande uso em sistemas embarcados críticos de segurança. Assim, a ferramenta foi projetada para fazer verificação de programas em C como entrada. No SatAbs, uma ênfase especial foi feita para suportar um subconjunto grande da linguagem C [12].

Com o objetivo de enfrentar o problema da escalabilidade, o SatAbs processa automaticamente uma abstração do programa dado como entrada. A abstração é o principal método na redução do espaço de estados dos sistemas de software. Abstração de predicado é um dos métodos mais populares e extensamente aplicado. Cada predicado é representado por uma variável booleana no modelo abstrato, enquanto as variáveis originais são eliminadas. O programa abstrato é criado usando a Abstração Existencial, que é uma abstração conservadora para propriedades de atingibilidade(reachability). Se a

propriedade prender no modelo abstrato, prende também no programa original. O inconveniente da abstração conservadora é que quando a verificação de modelos do programa falha, ele pode produzir um contra-exemplo que não corresponde a um contra-exemplo concreto. Isto é chamado um contra-exemplo “spurious”. Quando um contra-exemplo “spurious” é encontrado, o refinamento é executado ajustando o conjunto dos predicados de uma maneira que elimine este contra-exemplo. Isto é automatizado pelo CEGAR.

A característica que distingue o SatAbs de outras ferramentas existentes é a integração de um resolvente SAT na abstração, na simulação, e nas etapas de refinamento do loop do refinamento da abstração. Isto permite codificações precisas da semântica da linguagem C, incluindo o limite da aritmética de ponteiro e do vetor de bits.

A fim fazer o SatAbs aplicável a uma escala larga de programas de baixo nível, SatAbs suporta a maioria de construções encontradas na linguagem C. Tem o suporte para arrays (com tamanho possivelmente ilimitado), e as uniões. SatAbs é integrado na interface gráfica do usuário(GUI) do CBMC. A interface do usuário permite ao usuário um passo a passo no trace do contra-exemplo gerado pelo SatAbs como se fosse um debugger.

O SatAbs usa a quantificação booleana baseada no SAT a fim de computar o modelo abstrato. O modelo abstrato é passado a um verificador modelo. Se o verificador de modelos retornar um contra-exemplo, tem que ser simulado no código original para verificar se é “spurious”. Dado um trace abstrato do erro, SatAbs primeiramente verifica se contém transições “spurious”. Estas transições “spurious” são causadas pelo particionamento feito durante a computação da abstração. SatAbs forma uma instância SAT para cada transição no trace do erro. Se encontrada como sendo insatisfatível, a transição é “spurious”. A ferramenta usa o núcleo insatisfatível da instância para o refinamento eficiente. A ausência de transições “spurious” não garante que o trace do erro é real. Assim, SatAbs dá forma a uma outra instância SAT. Corresponde ao BMC no programa original que segue o fluxo do controle dado pelo trace abstrato do erro. Se satisfatível, SatAbs constrói um trace do erro da atribuição satisfeita, que mostra o caminho do erro. Se insatisfatível, o modelo abstrato é refinado adicionando predicados [9].

Quando os usuários do Satabs não possuírem conhecimento sobre os algoritmos subjacentes da abstração do refinamento, a compreensão das classes das propriedades que podem ser verificadas é crucial. Algumas propriedades que o Satabs permite a verificação são [9]:

- Overflow do buffer. Para cada array, Satabs verifica se o limite superior ou inferior são violados sempre que o array é acessado (isto é, sempre que o programa lê ou escreve no buffer).

- Segurança do ponteiro. O Satabs busca por null-pointers.

- Divisão por zero. Satabs verifica se há um caminho no programa que executa uma divisão por zero.

- Afirmações especificadas pelo usuário. Esta é a classe mais genérica das propriedades suportadas. Satabs verifica violações da afirmação. O usuário pode usar a função “assert” para especificar condições arbitrárias que têm para prender em determinados pontos no programa.

3.6 – MAGIC / COMFORT

O MAGIC e o ComFoRT são o resultado da colaboração entre Carnegie Mellon’s School of Computer Science (SCS) e a Software Engineering Institute (SEI). Ambas as ferramentas beneficiaram-se desta colaboração.

- MAGIC é um software verificador de modelos para programas concorrentes escritos em C. Está publicamente disponível na versão 1.0.

- ComFoRT é uma ferramenta de raciocínio que combina o verificador de modelos COPPER (baseado no MAGIC v1.0), com uma estrutura adicional que permite seu uso efetivo em desenvolvimento de software baseado em componentes.

3.6.1 - MAGIC

O alvo do projeto MAGIC é desenvolver ferramentas e técnicas para analisar e raciocinar sobre os componentes de software escritos na linguagem de programação C. O objetivo total do MAGIC é verificar a conformidade entre as especificações dos componentes e suas implementações. As implementações podem ser concorrentes, isto é, compostas de múltiplas linhas ou de processos que se comunicam através de mensagens ou de memória compartilhada. O MAGIC segue o paradigma do refinamento guiado da abstração do contra-exemplo (CEGAR). Primeiramente, um modelo finito é extraído da implementação do componente usando várias técnicas de abstração, por exemplo, abstração de predicado. Em seguida o modelo é verificado de encontro à especificação. Se um contra-exemplo for encontrado, sua validade está verificada e o modelo é refinado sucessivamente para se livrar de contra-exemplos "spurious". A ferramenta MAGIC é composicional. Usando o MAGIC, o problema de verificar uma implementação grande pode ser decomposto na verificação de um número de fragmentos menores, mais gerenciáveis. Estes fragmentos podem ser verificados separadamente, permitindo ao MAGIC suportar programas industriais grandes. Atualmente, o foco da pesquisa está na manipulação melhorada da concorrência e da memória compartilhada [4].

As características chaves do MAGIC v1.0 foram desenvolvidas conjuntamente pelo SCS e pelo SEI com a finalidade expressa de suportar o ComFoRT.

Para ver a evolução do MAGIC para o ComFoRT e como eles estão relacionados, ver a Figura 3.7. Esta figura situa também o COPPER, o verificador de modelos que está sendo utilizado pelo SEI para o uso no ComFoRT. O COPPER é baseado na linha do MAGIC v1.0, mas está evoluindo para satisfazer às necessidades do ComFoRT.

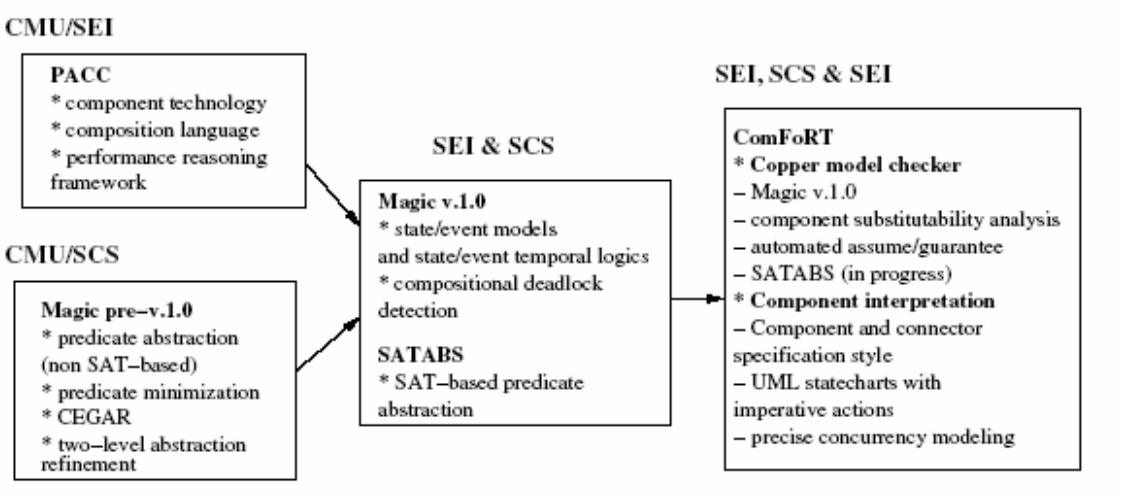

Figura 3.7 - Evolução dos projetos MAGIC e ComFoRT

3.6.2 - COMFORT (Motor Verificador de Modelos)

Um objetivo importante do ComFoRT é conseguir a verificação escalável de sistemas de software baseados em componentes. Conseqüentemente, o suporte para técnicas de provas de abstração e raciocínio composicional, fatores chaves em softwares verificadores de modelos, guiaram o desenvolvimento de um motor verificador de modelos. O MAGIC foi usado como ponto de partida por suportar estas técnicas.

O ComFoRT usa a técnica automática de abstração de predicado do MAGIC para criar modelos de estados finito do software. A validação do contra-exemplo e os procedimentos de refinamento da abstração do MAGIC são usados dentro de um loop inteiramente automatizado do CEGAR para reduzir a complexidade da verificação [13].

O verificador de modelos do ComFoRT, o COPPER, foi construído baseado na ferramenta MAGIC. O COPPER executa um número de técnicas de redução do espaço de estados, incluindo abstração automatizada do predicado, CEGAR e raciocínio composicional.

A entrada para o motor verificador de modelos do ComFoRT é um programa expressado nas combinações do código em C, das expressões de processos seqüenciais finito (FSP), e declarações auxiliares. Para simplificar,

esse conjunto é chamado como programas CFA (C, FSP e Auxiliar). A interpretação do ComFoRT gera uma descrição de sistema em que o comportamento do sistema é dividido em comunicação, em módulos concorrentes descritos no CFA. A maioria do comportamento está descrito em C, enquanto expressões FSP são usadas para descrever a maneira em que os processos se comunicam um com o outro através dos eventos [13].

Algumas características foram originalmente parte do MAGIC e outras foram introduzidas para a ferramenta ComFoRT. Ambas as características serão consideradas como parte do motor verificador de modelos resultante do ComFoRT.

Abordagem de verificação baseada em abstração

A característica do núcleo do ComFoRT que o permite de verificar software é a abordagem baseada na abstração para extração do modelo de estados finito. Adicionalmente, o ComFoRT tem diversas outras características importantes que o fazem útil para verificar software [13]:

- Extrai modelos de estados finitos de concorrência, passagem de mensagem em programas em C e refina estes modelos usando uma ferramenta CEGAR.

- Abstração de predicado, validação do contra-exemplo, e o refinamento do modelo são todos executados em composição, isto é, uma unidade concorrente de cada vez [15].

- Usa numerosos algoritmos de otimização que reduzem extremamente os tamanhos dos modelos de estados finito que produz [14].

Na extração do modelo de estados finito, o conjunto inicial dos predicados pode ser obtido de muitas maneiras. A maneira mais comum é coletar as fórmulas que aparecem em expressões condicionais assim como na reivindicação a ser verificada. O usuário pode também especificar os predicados de interesse, talvez baseados em alguma compreensão mais

profunda do sistema. Os predicados novos são gerados, se necessários, na fase de refinamento do modelo, que é descrita em seguida.

No refinamento do modelo usando CEGAR**,** o modelo construído pela abstração do predicado é garantido como sendo uma abstração conservadora do sistema original, significando que cada comportamento no sistema original é representado por algum comportamento no modelo, embora o modelo possa conter mais comportamentos. Em conseqüência, se o modelo satisfazer à reivindicação, faz assim no sistema original [16]. Entretanto, um contra-exemplo obtido verificando o modelo pode ser “spurious”. O motor verificador de modelos analisa o contra-exemplo e, se for “spurious”, usa esta informação para derivar predicados adicionais e para construir uma abstração nova, mais fina do sistema. Os predicados novos são obtidos automaticamente usando um provador de teorema. A verificação é repetida então com o modelo refinado.

Uma técnica foi desenvolvida junto com a Carnegie Mellon’s School of Computer Science para computar abstrações exatas de predicado de software e de hardware escritos em C e em SystemC enumerando atribuições a uma única instância SAT. Essa abordagem não requer um número exponencial de chamadas do provador de teorema, ao contrário da maioria das ferramentas existentes de abstração de predicado. Uma outra vantagem da abordagem baseada no SAT é que permite modelar o overflow aritmético, manipulações do vetor de bits, ponteiros e arrays.

Abordagem Composicional da verificação

Além dos procedimentos automatizados de abstração, o motor verificador de modelos aplica o raciocínio composicional dentro da ferramenta CEGAR para reduzir a complexidade da verificação.

O loop do refinamento abstrato da verificação continua até que um contra-exemplo real seja obtido, ou o sistema ser verificado como correto. Na teoria, o loop do CEGAR não é garantido para terminar ao verificar programas arbitrários do CFA. Na prática, entretanto, foi completamente eficaz [17].

Verificação baseada em Estado/Evento.

O verificador de modelos COPPER fornece modelos formais para a verificação de software que considera a distinção entre dados (estados) e estruturas de uma comunicação (eventos). A maioria dos modelos formais são baseados em estados ou baseados em eventos, mas o COPPER fornece modelos que incorporam ambos.

O algoritmo de verificação de modelos do COPPER suporta a verificação de propriedades de segurança e de liveness de sistemas estado/evento. Uma outra característica da ferramenta baseada em estado/evento é uma técnica composicional de detecção de deadlock que não somente detectam eficientemente deadlock, mas também atos como um procedimento adicional de redução do espaço [13].

Detecção de deadlock

Verificar a ausência de deadlock em um sistema composto é uma exigência comum que deve ser satisfeita antes que um sistema possa ser desenvolvido. Isto é especialmente verdadeiro para sistemas críticos de segurança, tais como sistemas embarcados, que se esperam sempre responder aos estímulos externos ou requisições de serviço dentro de um prazo fixo. Além disso, sempre que o deadlock é detectado, é altamente desejável fornecer aos desenvolvedores do sistema um feedback do que causou o deadlock. Entretanto, apesar dos esforços significativos, validar a ausência de deadlock nos sistemas é um grande desafio. O problema é especialmente complexo para processos concorrentes que se comunicam, por exemplo, através de semáforos. O obstáculo preliminar é a explosão do espaço de estados. A abstração e o raciocínio composicional, são úteis em detectar o deadlock. O ComFoRT usa uma abordagem de verificação de modelos para a detecção de deadlock [18]. Para detectar deadlock, o motor verificador de modelos do ComFoRT estende a ferramenta composicional do CEGAR com uma noção de recusa da abstração para detectar um deadlock ou para provar que nenhum deadlock existe. A abordagem resultante do CEGAR para a detecção de deadlock é completamente automatizada e fornece um contra-exemplo sempre que um deadlock é detectado [13].

Model Checker	Instituto	Modelo	Método
BLAST	UC Berkeley	ANSI C restrito	Abstração de predicado, CEGAR, provador de teorema.
CBMC	CMU	ANSI C	Bounded Model Checking (BMC)
MAGIC	CMU	ANSI C restrito	Abstração de predicado, CEGAR, provador de teorema.
MOPS	UC Davis, UC Berkeley	ANSI C restrito	Verificação de Modelos em CFG
SatAbs	CMU	ANSI C restrito	Abstração de predicado, CEGAR, resolvente SAT.
SLAM	Microsoft Research	ANSI C restrito	Abstração de predicado, CEGAR, provador de teorema.

Tabela 3.1 - Lista de Model Checkers

4 - ANÁLISE DAS FERRAMENTAS

Neste capítulo, algumas ferramentas descritas no capítulo anterior são analisadas, com relação às abordagens que usam e suas vantagens, suas características, como também suas potencialidades e limitações.

Quatro propriedades são consideradas para ser um bom verificador de modelos: corretude (soundness), integralidade (completeness), terminação (termination) e utilidade (usefulness). Entretanto, é desafiador conseguir todas as características de uma vez.

A análise é considerada “sound” se cada erro verdadeiro estiver relatado pela análise, ou seja, não apresentar “bugs”. A análise está completa se cada erro relatado for um erro verdadeiro, isto é, não há nenhum positivo falso. Um positivo falso é quando uma ferramenta de análise relata que um programa tem um erro, mas não tem. Terminação se refere à terminação da verificação, por exemplo, quando se faz uma verificação, a análise que é iniciada pelo verificador de modelos pode terminar ou não terminar, ou seja, se o problema for indecidível, é possível que o algoritmo do verificador de modelos possa não terminar.

4.1 - SLAM

4.1.1 - POTENCIALIDADES

O SLAM foi usado com sucesso para verificar propriedades dominantemente de controle, propriedades que consideram o fluxo de controle do programa mais importante do que o fluxo de dados. Ele suporta procedimentos recursivos e mutuamente recursivos. O SLAM é considerado “sound” com respeito às suposições iniciais que fez (a respeito do modelo lógico da memória, do aliasing, etc.). O SLAM está atualmente incompleto, ou seja, pode reportar erros falsos. Como o SLAM foi usado com sucesso em Device Drivers e é parte do SDV (Static Driver Verifier) [19] agora, ele é definitivamente útil.

O SLAM reivindica ser preciso no relato dos erros detectados. O SLAM dá como saída um caminho do erro que possa ser traçado ao programa original diretamente. Embora haja um caminho do erro para cada causa do erro, o SLAM pode negligenciar algumas causas do erro, ou seja, pode desconsiderar algumas possíveis causas que a ferramenta acha não ter sido crucial para o acontecimento do erro, diminuindo o número de possibilidades a serem revisadas e ainda pode relatar causas do erro “spurious”, causas que a ferramenta aponta como causadora do erro, mas na verdade não é, ou seja, causas falsas [1].

Os resultados do SLAM são muito bons. O maior Driver processado pelo SLAM tem aproximadamente 60K linhas de código. A maior abstração analisada pelo SLAM tem várias centenas de variáveis booleanas.

Assim, trabalha bem para problemas específicos do domínio como Device Drivers. Embora o toolkit do SLAM fosse testado também para bibliotecas de software multi-threaded, a análise feita pelo toolkit do SLAM foi focada no domínio da aplicação de Device Drivers. Por isso, não se sabe muito sobre suas potencialidades em outros domínios. Desde que o toolkit do SLAM não está disponível publicamente, sua avaliação e comparações com outras ferramentas têm somente sido estudada pela Microsoft Research [1].

4.1.2 - LIMITAÇÕES

Atualmente, o SLAM enfrenta alguns problemas, sendo o principal o que trata dos ponteiros. Abstrair de uma linguagem com ponteiros (C) a uma sem ponteiros (programas booleanos) é difícil. Estritamente falando, C suporta somente a chamada pelo valor, mas com ponteiros e operador de endereço, a chamada por referência pode ser simulada. Isto cria um problema, porque os programas booleanos suportam somente resultados de chamada por valor. O SLAM imita a chamada por referência com resultados da chamada por valor. Além disso, o SLAM não suporta programas muito grandes atualmente, como já dito, o maior Driver processado pelo SLAM tem aproximadamente 60K linhas de código [1].

4.2 - BLAST

4.2.1 - POTENCIALIDADES

O BLAST é usado para analisar estaticamente programas em C. Usa a abstração sob demanda para reduzir o refinamento desnecessário da abstração. Há muitas vantagens na abordagem da abstração “preguiçosa”. Primeiramente, somente a parte alcançável do espaço de estados é abstraída, que é muito menor do que o espaço de estados abstrato inteiro. A partir daí, as partes diferentes do espaço de estados têm precisões diferentes. Assim, poucos predicados têm que ser processados em cada ponto. Por último, a verificação do modelo não é repetida para aquelas partes do espaço de estados que já se sabe não ter erros (de alguma abstração mais grosseira). Isto reduz as exigências de espaço e do tempo consideravelmente. O BLAST separa as estruturas de dados internas da abstração simbólica do algoritmo de verificação de modelos. Uma das razões para isso é facilitar o reuso do código para construir os verificadores de modelos. Atualmente, o BLAST diz suportar muitas construções sintáticas de C, incluindo estruturas e procedimentos. Entretanto, a aritmética do inteiro é modelada como a aritmética da precisão infinita, e um modelo lógico da memória é suposto. As chamadas do procedimento também são suportadas usando uma pilha explícita [2].

O BLAST é “sound” com respeito às suposições feitas como também não é completo, pode relatar erros falsos. Com relação à propriedade de terminação, o BLAST diz ter terminado a maioria das vezes, às vezes após a adição de predicados pelo usuário, ou seja, após a adição de predicados, um determinado problema pode passar a ser decidível, fazendo com que o algoritmo termine. Entretanto, o problema de encontrar muitos predicados permanece. Pode haver exemplos quando o BLAST não encontra muitos predicados para provar a propriedade. Isto acontece porque o motor da descoberta de predicado do BLAST pode não ser esperto o bastante para descobrir que muitos predicados provam a propriedade e por isso pode falhar. Entretanto, em programas grandes, os predicados podem não ser intuitivos para o usuário. Pode haver alguma classe dos programas onde o motor da

descoberta do predicado não descobre muitos predicados e relata um erro (positivos falsos). Similarmente, os programas que manipulam suas variáveis com o aliasing dos ponteiros puderam satisfazer à propriedade dada, mas desde que o BLAST ignora aquelas declarações, uma aceitação falsa (negativo falso) pode ser produzida. Aqui também, os predicados novos podem ser adicionados para alvejar especificamente o aliasing. Em ambos os casos, adicionar predicados novos pode ajudar. Entretanto, ao adicionar predicados manualmente, o usuário deve ter algum conhecimento do funcionamento interno da ferramenta e de uma compreensão completa do programa. Há um incentivo para programadores usarem o BLAST para verificação, um plugin do eclipse foi desenvolvido para o BLAST. Assim a verificação de software e o desenvolvimento de software podem ser feitos lado a lado. Isto impulsiona a popularidade do BLAST entre programadores.

O BLAST foi desenvolvido para verificar propriedades de segurança em programas em C. Foi usado também com sucesso no domínio de Device Drivers para verificar propriedades temporais de segurança. Assim como o SLAM, o BLAST verificou com sucesso violações de propriedades de segurança encontradas em programas de Device Drivers com até 60K linhas de código. O BLAST encontrou erros em diversos Drivers. E diz também ter provado que outros Drivers executam corretamente a especificação [2].

4.2.2 - LIMITAÇÕES

O BLAST é relativamente independente da máquina e do compilador. Entretanto, o BLAST foi testado somente em Intel x86 usando o compilador Ocaml (versão 3.04) no Linux e o cygwin no Windows.

A versão atual não suporta ponteiros para função. A exatidão da análise é a suposição que as chamadas do ponteiro da função são irrelevantes à propriedade que está sendo verificada [2].

Atualmente funções recursivas não são suportadas. Esta é uma limitação grande nos arquivos que podem ser analisados.

4.3 - SLAM x BLAST

O SLAM e o BLAST são baseados em conceitos similares e têm muitas características em comum. Como já dito, ambas as ferramentas executam a análise estática e o refinamento guiado da abstração do contra-exemplo (CEGAR) para extrair um modelo de estados finito de um programa em C. Assim, ambas as ferramentas enfrentam o problema enfrentado pela análise estática e a propriedade que verifica, isto é a possibilidade de falsos alarmes e de não terminação. Ambas as ferramentas verificam propriedades de segurança nos programas em C seqüenciais que são especificados pelo usuário. Ambas foram testadas em Device Drivers e o SLAM foi integrado no produto SDV (Static Driver Verifier) do Windows [19]. O BLAST é comparável ao SLAM em escalabilidade e em precisão.

O SLAM e o BLAST suportam várias construções da linguagem C (como ponteiros, estruturas, e procedimentos) e supõem um modelo lógico da memória. Ambas as ferramentas têm opções para incluir escolhas não determinísticas para chamadas às funções de biblioteca. Embora ambas as ferramentas sejam projetadas inicialmente para programas em C seqüenciais, o BLAST está sendo estendido para programas multi-threaded. Há algumas diferenças entre o SLAM e o BLAST. Uma diferença chave é o uso da abstração preguiçosa no BLAST. A abstração preguiçosa permite que a descoberta do predicado seja feita localmente e sob-demanda e economiza assim muito espaço e tempo. Com respeito à linguagem de especificação, o BLAST tem uma vantagem sobre o SLAM. O SLIC do SLAM não suporta propriedades do tipo-estado. Monitora somente chamada de funções e o retorno e assim, é limitado para especificação de interfaces. BLAST, entretanto, considera as propriedades do tipo-estado (CIL). Além disso, a linguagem de especificação do BLAST diz ter a sintaxe na linguagem C, o que é mais fácil de aprender para programadores do que aprender uma nova linguagem de especificação. De fato, ambas as linguagens de especificação têm um olhar e uma sensação de linguagens orientadas a aspecto. A abstração usada no SLAM é um programa booleano que é construído antes da execução, visto que, o BLAST constrói a árvore abstrata de atingibilidade

(reachability) sob-demanda durante a execução do CFA. Entretanto, internamente o SLAM e o BLAST usam diagramas binários de decisão (BDDs) para sua análise. Além disso, o SLAM diz que suporta funções recursivas e mutuamente recursivas, visto que o BLAST não suporta funções recursivas enquanto mantém uma pilha para chamadas dos procedimentos.

4.4 - MOPS

4.4.1 - POTENCIALIDADES

O MOPS tem o objetivo de ser soundness, ter precisão e ter escalabilidade. Desde que o MOPS é projetado para ser uma ferramenta prática para verificar propriedades de segurança em programas grandes, tenta ter um equilíbrio entre soundness, precisão, e escalabilidade. Para ser sound, o MOPS segue cada caminho no programa. Para assegurar escalabilidade, o MOPS usa abordagem que: sacrificar a precisão de sua análise do fluxo de dados é melhor que sacrificar a escalabilidade. Desde que a análise do fluxo de dados apresenta muitas dificuldades para a escalabilidade, MOPS escolhe desconsiderar o fluxo de dados. Ou seja, o MOPS ignora a maioria de valores dos dados no programa e supõe que cada variável pode assumir qualquer valor. Como está, o MOPS é mais recomendado para verificar as propriedades de fluxo de controle.

Em outras palavras, os dois principais objetivos do MOPS são soundness e escalabilidade. O Soundness permite ao MOPS de ser usado não somente como uma ferramenta que encontra erros, mas também como uma ferramenta de verificação de propriedades. Para avaliar o soundness do MOPS, se consideram dois estágios do MOPS: programa de transformação do código em C em um PDA e a verificação de modelos no PDA. O último estágio é sempre “sound”. O estágio anterior é sound contanto que cada caminho da execução no programa seja capturado no PDA. Isto requer que o programa seja um programa em C portátil e single-threaded, por exemplo, não tenha nenhuma geração de código em tempo de execução. Escalabilidade permite ao MOPS de trabalhar em uma escala larga de programas, especialmente os mais complexos. O MOPS conseguiu uma alta escalabilidade negligenciando a

maioria do fluxo de dados e comprimindo os CFGs de forma muito eficiente. Esta vantagem, entretanto, acarreta em uma precisão mais baixa: O MOPS pode, equivocadamente, considerar praticáveis os caminhos que são inalcançáveis no programa, e fornecer alertas estranhos. Embora haja sempre um problema em equilibrar a escalabilidade e a precisão, está se estudando como melhorar a precisão do MOPS sem sacrificar a escalabilidade. O MOPS suporta bem os programas grandes no tempo e no espaço, superando o problema da escalabilidade que outros verificadores de modelos possuem.

Desde que toda a propriedade não trivial sobre uma linguagem reconhecida por uma máquina de Turing é indecidível, nenhuma ferramenta que verifica uma propriedade não trivial pode ser “sound” e completa. O MOPS se esforça para ser “sound”, então está inevitavelmente incompleto, pois o MOPS pode gerar “traces” positivos falsos, isto é, “traces” do programa que são inalcançáveis ou aqueles que não violam a propriedade. Infelizmente, um grande número de “traces” positivos falsos traz muitas dificuldades para o usuário, conseqüentemente, é essencial evitar a geração de muitos positivos falsos [10].

A usabilidade do MOPS é considerada boa, pois uma vez o usuário formalizando uma propriedade de segurança na central do fluxo de controle da propriedade temporal de segurança, é razoavelmente fácil de descrevê-la usando um FSA. Embora seja necessário se familiarizar com a sintaxe de ASTs antes, para que se possa escrever novos FSAs. O MOPS relata todos os erros no programa listando todos os “traces” do erro que violou a propriedade.

As ferramentas de análise estática como MOPS são valiosas em encontrar vulnerabilidades nos programas que funcionam hoje, mas são ainda mais valiosas em impedir vulnerabilidades de serem introduzidas em programas no futuro.

A ferramenta SLAM é muito precisa, entretanto, não suporta ainda os programas muito grandes. Comparado ao SLAM e ao BLAST, o MOPS oferece precisão para a escalabilidade e eficiência pela consideração somente do fluxo de controle e por ignorar a maioria de fluxo de dados, ou seja, o MOPS é muito mais escalável que o SLAM e o BLAST, mas perde precisão para escalabilidade, por isso, é menos precisa que o SLAM e o BLAST. Também pelo MOPS não ser um processo iterativo, não sofre da possível não

terminação como o SLAM e o BLAST. Uma contribuição chave à usabilidade do MOPS é sua habilidade em relatar somente um “trace” de erro para cada causa do erro no programa, que reduz significativamente o número de erros que o programador tem que rever. Entre as ferramentas discutidas, somente o SLAM documentou uma similar habilidade. Comparada a abordagem do MOPS, a abordagem do SLAM é menos precisa porque pode negligenciar causas do erro.

4.4.2 - LIMITAÇÕES

O MOPS é sound sob as seguintes condições:

- O programa é single-threaded. Ou seja, o MOPS é inapropriado para verificar programas concorrentes.
- O programa é seguro de memória, por exemplo, nenhum over-runs de buffer.
- O programa é portátil.

Por priorizar a escalabilidade e ignorar o fluxo de dados, a precisão do MOPS é afetada, sendo mais baixa: O MOPS pode, equivocadamente, considerar praticáveis os caminhos que são inalcançáveis no programa, e fornecer alertas estranhos. Quando o MOPS encontra potenciais violações de propriedades de segurança em um programa, relata os traces do erro, que são úteis aos programadores para identificar erros no programa. Desde que o MOPS é conservador, pode relatar traces positivos falsos, isto é, traces que de fato não violam as propriedades de segurança, mas que são consideradas como violação devida à análise imprecisa do MOPS.

Além disso, a versão atual do MOPS não considera os fluxos do controle que não estão no CFG, tal como as chamadas indiretas através do ponteiro da função, jumps não locais e bibliotecas carregadas em tempo de execução. Embora esta abordagem introduza o unsoundness, não é uma limitação da abordagem, mas sim uma limitação da implementação atual do MOPS.

4.5 - CBMC

4.5.1 - POTENCIALIDADES

O CBMC usa a técnica BMC, técnica apresentada na seção 2.4.1. A motivação original de BMC era o sucesso dos resolventes SAT em resolver fórmulas booleanas para verificação de modelos. Durante os últimos anos houve um aumento tremendo no poder de raciocínio dos resolventes SAT. Agora podem lidar com exemplos com centenas de milhares de variáveis e de milhões de cláusulas. A principal potencialidade do CBMC é o suporte a maioria das estruturas de C, ou seja, a escalabilidade é um grande diferencial do CBMC, suportando, por exemplo, a aritmética de ponteiro, operadores de inteiro, casts de tipo, chamadas de funções, não-determinismo, estruturas, uniões nomeadas, memória dinâmica, etc.

Abaixo, uma tabela [8] com as estruturas de C suportadas pelo CBMC:

Supported Language Features		Properties checked
Basic Data Types	All scalar data types `float` and `double` using fixed-point arithmetic. The bit-width can be adjusted using a command line option.	
Integer Operators	All integer operators, including division and bit-wise operators Only the basic floating-point operators	Division by zero Overflow for signed data types
Type casts	All type casts, including conversion between integer and floating-point types	Overflow for signed data types
Side effects	`CBMC` allows all compound operators	Side effects are checked not to affect variables that are evaluated elsewhere, and thus, that the ordering of evaluation does not affect the result.
Function calls	Supported by inlining. The locality of parameters and non-static local variables is preserved.	1. Unwinding bound for recursive functions 2. Functions with a non-void return type must return a value by means of the return statement.
Control flow statements	`goto`, `return`, `break`, `continue`, `switch` ("fall-through" is not supported)	

Non-Determinism	User-input is modeled by means of non-deterministic choice functions	
Assumptions and Assertions	Only standard ANSI-C expressions are allowed as assertions.	Assertions are verified to be true for all possible non-deterministic choices given that any assumption executed prior to the assertion is true.
Arrays	Multi-dimensional arrays and dynamically-sized arrays are supported	Lower and upper bound of arrays, even for arrays with dynamic size
Structures	Arbitrary, nested structure types; may be recursive by means of pointers; incomplete arrays as last element of structure are allowed	
Unions	Support for named unions, anonymous union members are currently not supported	`CBMC` checks that unions are not used for type conversion, i.e., that the member used for reading is the same as used for writing last time.
Pointers	Dereferencing	When a pointer is dereferenced, `CBMC` checks that the object pointed to is still alive and of matching type. If the object is an array, the array bounds are checked.
	Pointer arithmetic	
	Relational operators on pointers	`CBMC` checks that the two operands point to the same object.
	Pointer Type Casts	Upon dereferencing, the type of the object and the expression are checked to match
	Pointers to Functions	The offset within the object is checked to be zero
Dynamic Memory	`malloc` and `free` are supported. The argument of malloc may be a nondeterministically chosen, arbitrarily large value.	Upon dereferencing, the object pointed to must still be alive. The pointer passed to `free` is checked to point to an object that is still alive. `CBMC` can check that all dynamically allocated memory is deallocated before exiting the program ("memory leaks").

Tabela 4.1 - Estruturas de C suportadas pelo CBMC

A combinação de uma análise automática, de um grande número de características de C suportadas e uma interface amigável para o programador faz esta ferramenta muito útil.

4.5.2 - LIMITAÇÕES

O BMC ainda não resolve o problema da complexidade da verificação de modelos, pois ainda confia em um procedimento exponencial e por isso tem capacidade limitada. BMC tem também a desvantagem de não poder provar a ausência de erros, na maioria dos casos reais. Conseqüentemente CBMC se junta ao arsenal de ferramentas automáticas de verificação, mas não substitui algumas delas. Por priorizar a escalabilidade, a precisão é comprometida, ou seja, a ferramenta pode apresentar positivos falsos, erros falsos podem ser relatados.

Ferramenta	Linguagem	Fundamentação	Interação	Free
CBMC	C/C++	Bounded Model Checking (BMC)	Mínima	SIM
BLAST	C	Abstração de Predicado	Moderada	SIM
SLAM	C	Abstração de Predicado	Moderada	NÃO
MOPS	C	Verificação de Modelos em CFG	Moderada	SIM

Tabela 4.2 - Ferramentas Model Checkers

	Soundness	Completeness	Termination	Propriedades Alvos
MOPS	SIM	NÃO	SIM	Fluxo de Controle
SLAM	SIM	NÃO	NÃO	Segurança
BLAST	SIM	NÃO	NÃO	Segurança

Tabela 4.3 - Características de algumas ferramentas

Apesar das ferramentas parecerem equivalentes ao olhar a tabela 4.3, elas diferem em alguma coisa, seja na abordagem utilizada ou na forma em que tratam a abordagem, existem características peculiares para cada

ferramenta, tornando difícil apontar uma ferramenta que seja a melhor, de fato, o que existe é que para cada situação uma ferramenta pode se encaixar melhor, pois dependendo da situação uma ferramenta pode trazer melhores resultados, tudo isso dependendo das características do código C do programa que se quer analisar e do que se espera da análise, ou seja, as ferramentas não substituem umas as outras, elas podem ser utilizadas em conjunto, extraindo-se o que se tem de melhor em cada uma.

Porém, analisando da forma mais geral possível e levando em consideração alguns outros fatores como disponibilidade da ferramenta e aplicação com sucesso em algum domínio de aplicação, além da abordagem utilizada, pode-se destacar a ferramenta BLAST. Com relação à abordagem, o BMC, descrito na seção 2.4.1 ainda possui limitações consideráveis e a utilização da abstração usando um resolvente SAT, descrita na seção 2.4.2 ainda está em fase de consolidação, as ferramentas que a utilizam ainda estão em fase de evolução, e apesar de suportar a maioria das construções de C, as ferramentas em si ainda apresentam várias restrições. Já o MOPS tem uma abordagem baseada em verificação de CFG (Gráfico do Fluxo de Controle), que desconsidera o fluxo de dados, sendo mais recomendado para verificar as propriedades de fluxo de controle de um sistema. Com isso, restam o SLAM e o BLAST, que utilizam a abordagem de abstração de predicado usando um provador de teorema. Com relação ao sucesso em algum domínio, o SLAM e o BLAST se destacam por já terem sido utilizados com sucesso no domínio de Device Drivers, e apesar de serem ferramentas baseadas em conceitos similares e possuírem muitas características em comum, algumas peculiaridades existem e foram relatadas na seção 4.3. Uma delas se refere à forma que a abstração dos predicados é realizada, tendo o BLAST uma abordagem mais eficiente, pois realiza a abstração sob demanda, ou seja, quando necessário, ao contrário do SLAM que faz a abstração de predicados do programa inteiro. Outro fator importante que o BLAST leva vantagem é com relação à disponibilidade, o BLAST está disponível publicamente, já o SLAM se tornou parte de um produto da Microsoft, o SDV (Static Driver Verifier) [19], conseqüentemente, não está disponível publicamente.

5 - ESTUDO DE CASO

A intenção deste capítulo é aplicar um Model Checker a um componente implementado na linguagem C. Este componente é um controlador de portas de um metrô. Abaixo uma descrição do componente:

5.1 - SOBRE O SISTEMA: CONTROLADORA GERAL DE PORTAS

O sistema foi implementado usando a linguagem C e faz parte do equipamento de Controle Geral de Portas AeS-0617, armazenado e processado por um microcontrolador da linha PIC da família 18F da Microchip. A intenção é que o sistema entre em operação no projeto Santiago Linha 2, Chile.

O software, que é um componente da CGP (Controladora Geral de Portas), define a partir das entradas existentes a possibilidade de abertura das portas (momento e lado de abertura) e comando de periféricos do trem, além de intertravamentos de segurança com outros equipamentos. Além do trabalho mecânico de abrir / fechar portas, o sistema também será responsável por emitir sinalizações (sonoras e luminosas) para seus operadores.

O sistema se encarrega de, automaticamente, executar os intertravamentos necessários com as condições seguras de abertura e fechamento de portas, considerando inclusive intertravamentos entre o sistema de portas e de tração.

Esse software possui interface com o operador do trem pela cabine de comando, com os funcionários de manutenção através de conectores nas CGPs que podem ser conectados à laptops providos do software de manutenção fornecido pela AeS, interfaces com o sistema TIMS (Train Information Monitoring System, sistema de monitoramento de informações para o condutor por tela presente nas cabines de condução) e com as os demais equipamentos do SCP (Sistema de Controle de Portas).

5.1.1 - CASOS DE USO ESCOLHIDOS

As descrições de casos de uso abaixo foram extraídas do documento de requisitos do sistema SRS-0617-1 Controle Geral de Portas [20].

Função abre portas (CML)

Caso o trem estiver em CML (Comando Manual), com seleção de cabine líder, chave seletora de lado (ST) habilitada e chave de abertura "KLOAN" (chave responsável por iniciar o processo de abertura de portas) selecionada em modo de preparação será habilitado um flag de temporização de 10s. Caso a velocidade estiver abaixo de 6Km/h ou diminua de 6Km/h dentro desse intervalo e todas as condições anteriores permaneçam inalteradas, será executada abertura de portas no respectivo lado selecionado e o flag de temporização zerado. Caso o operador selecione a chave "KLOAN" em modo de despreparação dentro do período de 10s e antes do trem atingir velocidade inferior à 6Km/h o flag também será zerado. Caso o operador mude a posição da chave de seleção de lado de abertura após a abertura de portas, estas se fecharão sem a necessidade de comando.

Função fecha portas (CML – prioritária em relação à abertura)

Caso o trem estiver em CML (Comando Manual), com seleção de cabine líder, chave seletora de lado (ST) habilitada e o operador pressionar o botão de fechamento de portas correspondente ao lado da operação e mantê-lo pressionado até o fim da operação, as portas se fecharão. Caso o botão for solto antes do fim da operação, o fechamento será interrompido. Caso o botão volte a ser pressionado em 7s, as portas se fecham sem sinalização de início de fechamento. Caso os 7s tenham sido ultrapassados, todo o ciclo de fechamento deverá ser reiniciado. Caso o operador mude a posição da chave de seleção de lado de abertura após a abertura de portas, estas se fecharão sem a necessidade de comando.

5.2 – USO DE UM MODEL CHECKER NO COMPONENTE

O primeiro passo foi analisar a estrutura do componente do metrô escrito em C, onde a partir das estruturas de C usadas no componente se pode escolher uma ferramenta que seja adequada, que suporte bem as estruturas utilizadas e que apresente bons resultados. O código do componente pode ser encontrado no apêndice A.

Inicialmente, a ferramenta utilizada seria o BLAST, por motivos já relatados no capitulo 4, já que o SLAM não está disponível publicamente e no CBMC, a maioria dos recursos não se aplica às estruturas de C existentes no componente, como arrays, ponteiros, retornos de chamadas de funções, alocação dinâmica de memória e etc. Mas devido a problemas encontrados para execução do BLAST e do seu plugin do eclipse, a ferramenta utilizada para análise do componente foi o CBMC.

Como já relatado, o CBMC oferece muitos recursos para diversas estruturas de C, porém, devido às características da linguagem C utilizadas no componente em estudo (não possui arrays, alocação dinâmica de memória, etc), a única propriedade viável para ser executada no CBMC foi a que o usuário formula através da função assert, verificando, por exemplo, o estado que se encontra o programa e suas variáveis de controle depois da execução de uma função da aplicação. Apesar de ser limitado, ele pode apontar problemas como um comportamento indesejado, verificando os valores das variáveis, sendo possível também, determinar a alcançabilidade dos estados implementados no componente.

Um plugin do eclipse para a ferramenta foi instalado e o código precisou sofrer algumas adaptações devido ao mesmo ser apenas um componente de um sistema de metrô, variáveis do sistema precisaram ser declaradas no componente para simular situações que podem acontecer dependendo dos valores das variáveis. Os nomes das variáveis foram trocados para deixar mais claro a que se referem. Uma função main também foi declarada para poder executar a função responsável pelo componente do metrô. Outro detalhe é que no documento de requisitos a velocidade máxima para abertura de portas é 6 Km/h, mas no código a verificação é se a velocidade é menor que 3 Km/h para abertura de portas, sendo mantida este limite que se encontra no código.

5.3 – RESULTADOS OBTIDOS

Foram verificados diversos fluxos do sistema, sendo que dois deles foram selecionados e relatados:

- Um fluxo de uma operação de abrir e fechar portas com velocidade menor que 3Km/h.
- A verificação da alcançabilidade dos estados após a execução da função n vezes através do comando “while” (n limitado pela ferramenta).

Esses fluxos são determinados pelos valores de algumas variáveis do sistema, que em alguns trechos, foram colocadas para assumirem valores randômicos, utilizando um recurso oferecido pela ferramenta de possibilitar que a variável possa assumir valores aleatórios. A idéia é verificar o estado do programa e os valores das variáveis do sistema após a execução da função do componente sob certas condições, simulando valores de algumas variáveis utilizadas pela função e verificando após a execução da função se o estado e os valores de outras variáveis correspondem aos valores e estado esperados, ou seja, se a função está tendo o comportamento que se espera dela.

As condições do primeiro fluxo são: velocidade menor que 3, a chave seletora acionada para o lado direito, obrigatoriamente, a do lado esquerdo não, o sistema no estado inicial, chave KLOAN selecionada e os botões de fechamento lateral e do console do lado direito não acionados. Após a execução da função, os valores esperados são: os estados iniciais de portas fechadas e CML ativadas, comando de abrir, não executar comando de Gongo e ir para o próximo estado, no caso 1. Tudo isto foi declarado no “main” do componente, junto com o que vem em seguida relacionado a este primeiro fluxo, pois a idéia é mostrar etapa por etapa da verificação e colocando os resultados de cada trecho.

```
//vel <3
MEMORIA_informaVelocidadeMenorQueTres_CGP = true;
// dir
MEMORIA_informaChaveSeletoraLadoDireito_CGP = true;
MEMORIA_informaChaveSeletoraLadoEsquerdo_CGP = false;
//estado inicial
CGP_informarProximoEstadoDireita_MEMORIA = 0;
//chave Kloan Selecionada
MEMORIA_informaChaveKLOANSelecionada_CGP = true;
// botao dir console e lateral nao acionados
MEMORIA_informaBotaoFechamentoDireitoConsole_CGP = false;
MEMORIA_informaBotaoFechamentoDireitoLateral_CGP = false;
CGP_CML();

// verificação dos valores dadas as condições acima

assert(CGP_estadoInicialPortasFechadasCMLDireita_MEMORIA==TRUE);
assert(CGP_estadoInicialPortasFechadas_MEMORIA == TRUE);
assert(comandoDireita == ABRE);
assert(CGP_executarComandoGongo_MEMORIA == FALSE);
assert(CGP_informarProximoEstadoDireita_MEMORIA == 1);
```

O resultado da verificação das condições acima:

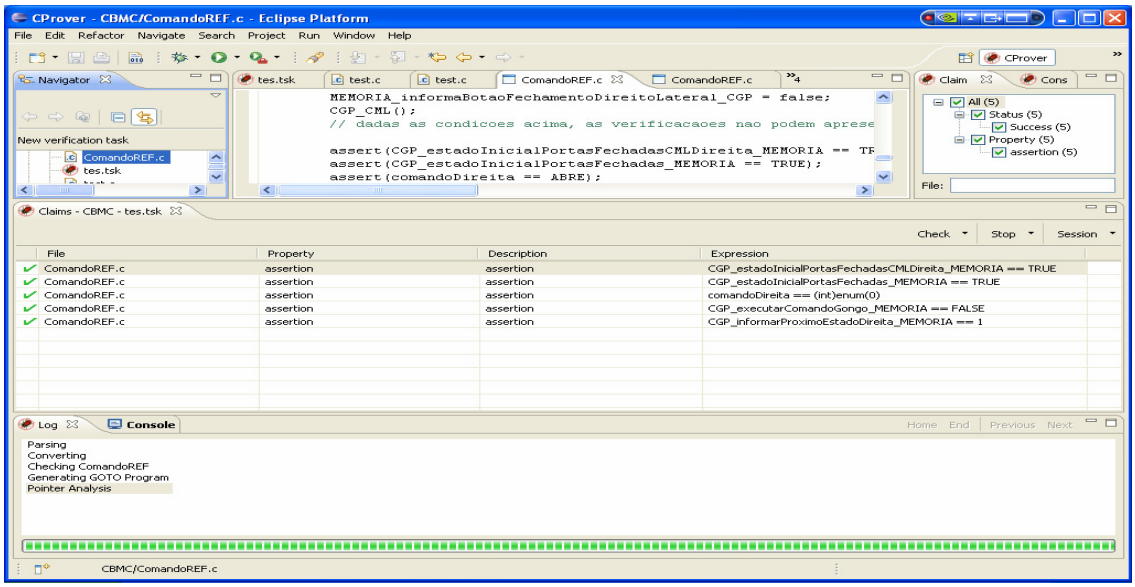

Figura 5.1 - Resultado da verificação do trecho 1 no CBMC

Como continuação do fluxo, permanecendo as condições já colocadas e com o sucesso das verificações anteriores, mostrados na Figura 5.1, desativando a seleção da chave KLOAN e chamando a função novamente, analisamos se vai para o estado 2, com esse trecho abaixo do mesmo main da verificação anterior.

```
MEMORIA_informaChaveKLOANSelecionada_CGP = FALSE;
CGP_CML();
//continuação do fluxo, retirando os outros asserts que já foram
verificados, o próximo assert é específico para a nova condição
da chave KLOAN.
//verifica se estado é igual a 2
assert(CGP_informarProximoEstadoDireita_MEMORIA == 2);
```

O resultado desse trecho:

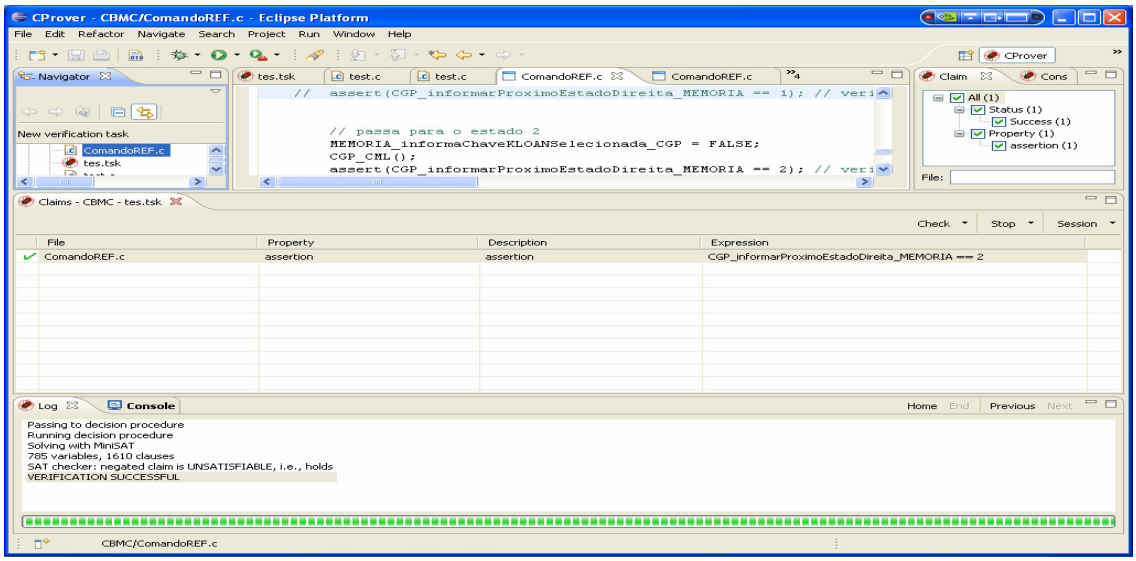

Figura 5.2 - Resultado da verificação do trecho 2 no CBMC

Continuando com as condições já estabelecidas acima, com o estado atual sendo 2, acionando os botões de fechamento do console e lateral do lado direito e chamando a função novamente. Os valores esperados estão dentro dos comandos assert estabelecidos para este trecho do main.

```
MEMORIA_informaBotaoFechamentoDireitoLateral_CGP = TRUE;
MEMORIA_informaBotaoFechamentoDireitoConsole_CGP = TRUE;
CGP_CML();

// verifica atribuições no estado 2
assert(MEMORIA_informaChaveKLOANSelecionada_CGP == FALSE);
assert(CGP_informaEstadoInicialRepouso_MEMORIA == 0);

// verifica valores atribuídos no estado 2
assert(CGP_iniciarContadorTempoGongo_MEMORIA == CNT_COUNT1);
assert(CGP_iniciarContadorTempoTotal_MEMORIA == CNT_COUNT3);
assert(CGP_zerarContadorGongo_MEMORIA == FALSE);
assert(CGP_executarComandoGongo_MEMORIA == TRUE);
assert(CGP_habilitarSinalizacaoInicioFechamento_MEMORIA==TRUE);
//verifica se estado é igual a 3
assert(CGP_informarProximoEstadoDireita_MEMORIA == 3);
```

O resultado deste trecho:

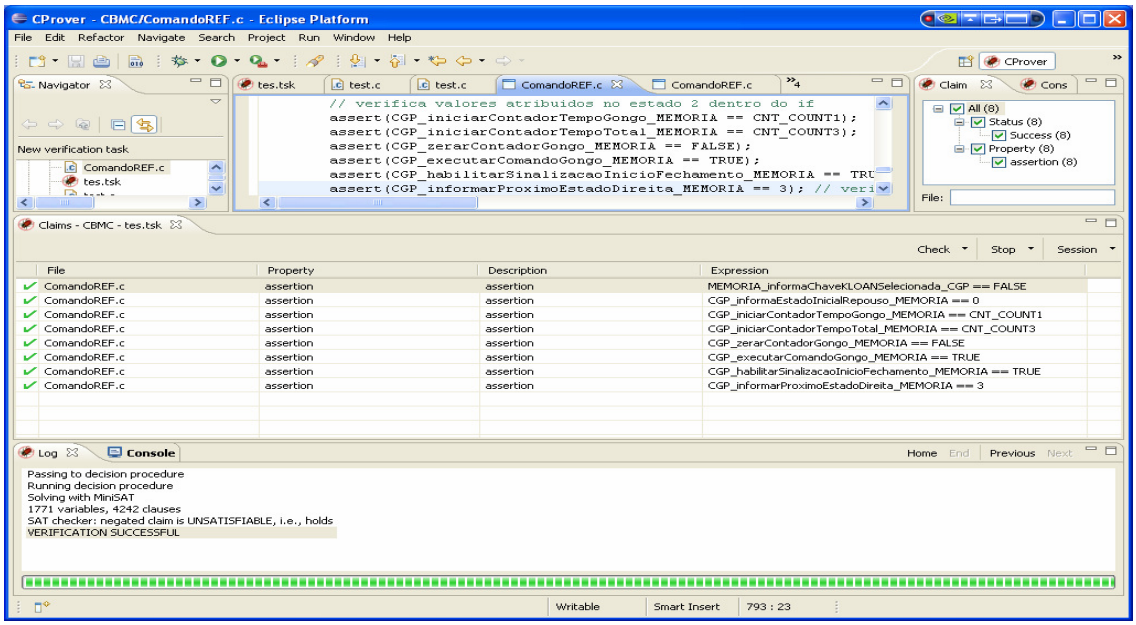

Figura 5.3 - Resultado da verificação do trecho 3 no CBMC

Agora no estado 3, ativando o zerar contador do Gongo e colocando os assert para este trecho.

```
CGP_zerarContadorGongo_MEMORIA = true;
CGP_CML();
// verifica se o comando esta sinaliza
assert(comandoDireita == SINALIZA);
// verifica se estado = 4
assert(CGP_informarProximoEstadoDireita_MEMORIA == 4);
```

O resultado do trecho acima foi:

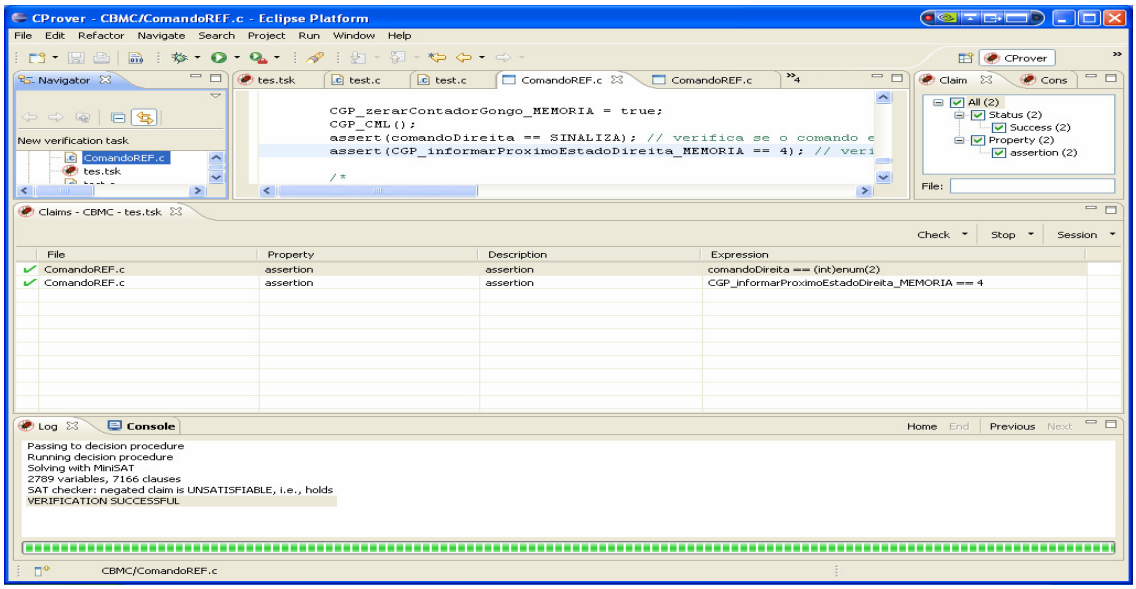

Figura 5.4 - Resultado da verificação do trecho 4 no CBMC

Já no estado 4 e ativando o estouro do tempo total e chamando a função novamente. É esperado que o comando seja de fechar e o estado seja 0, ou seja, o estado inicial.

```
MEMORIA_informarEstouroTempoTotal_CGP = true;
CGP_CML();
assert(comandoDireita == FECHA);
// volta pro estado 0 (inicial)
assert(CGP_informarProximoEstadoDireita_MEMORIA == 0);
```

O resultado deste novo trecho:

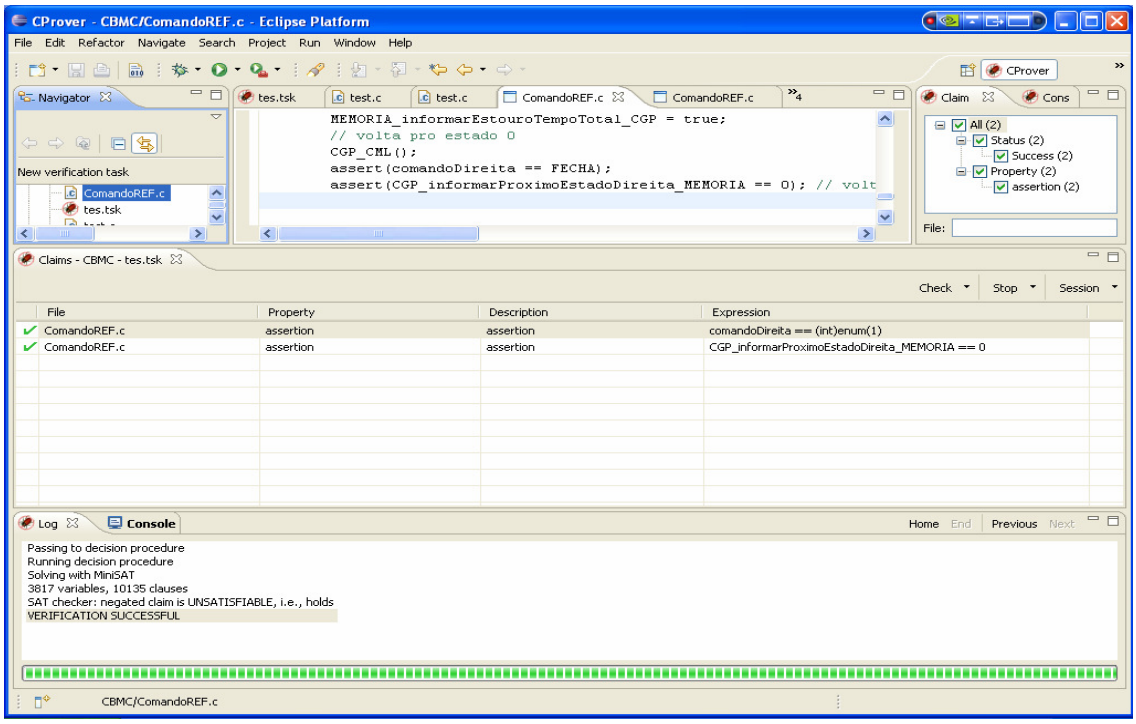

Figura 5.5 - Resultado da verificação do trecho 5 no CBMC

Ao final do fluxo, pode-se concluir através da análise realizada, que o comportamento da função corresponde ao que se espera dela dadas às condições que foram estabelecidas.

Mas com relação a um outro fluxo selecionado, um potencial problema foi encontrado, potencial porque a ferramenta é um verificador de modelos limitado, ou seja, ela busca situações até certo ponto, por isso é limitada. Foi escolhido um valor limite, valor esse para as iterações do loop ao qual a função foi submetida que levasse em conta um grande número de análises e não levasse muito tempo para fazer as verificações. Então, após a definição do parâmetro e a execução da verificação, a ferramenta apresentou um problema com relação a atingibilidade de um estado, no caso o estado 6. Porém, pode

haver a possibilidade da variável ser alterada em outro componente do sistema, pois a variável de estado é global, fazendo com que o estado não seja inatingível. Entretanto, como a análise está sendo feita somente em relação ao componente, considerando apenas as atribuições feitas pela função do componente, o estado 6 é inalcançável, ou seja, determinadas variáveis do sistema não são alteradas como pretende a implementação da função, caso o estado 6 fosse alcançado em algum momento, e com isso, podendo ocasionar algum problema para o sistema como um todo. Todo o relato foi feito considerando as funções com a chave seletora para o lado direito, mas tem validade para o lado esquerdo também, pois o fluxo é o mesmo após a seleção da chave, há uma replicação de código, só mudando algumas variáveis que especificam o lado da operação. A estratégia adotada foi a colocação do **assert(CGP_informarProximoEstadoDireita_MEMORIA<=5)**. Mostrando que este assert é sempre verdade, mostro que a variável de estado nunca assume valores maiores que 5, ou seja, não assume o valor 6, e conseqüentemente, o estado 6 não foi alcançado.

O main do componente foi declarado da seguinte forma:

```
int main(){

    // vel <3
    MEMORIA_informaVelocidadeMenorQueTres_CGP = true;
    // lado direito
    MEMORIA_informaChaveSeletoraLadoDireito_CGP = true;
    MEMORIA_informaChaveSeletoraLadoEsquerdo_CGP = true;
    //estado inicial
    CGP_informarProximoEstadoDireita_MEMORIA = 0;
    //chave Kloan Selecionada
    MEMORIA_informaChaveKLOANSelecionada_CGP = true;
    // botao dir console e lateral nao acionados
    MEMORIA_informaBotaoFechamentoDireitoConsole_CGP = false;
    MEMORIA_informaBotaoFechamentoDireitoLateral_CGP = false;
    CGP_CML();

    MEMORIA_informaChaveKLOANSelecionada_CGP = FALSE;
    CGP_CML();

    MEMORIA_informaBotaoFechamentoDireitoLateral_CGP = TRUE;
    MEMORIA_informaBotaoFechamentoDireitoConsole_CGP = TRUE;
    CGP_CML();

    CGP_zerarContadorGongo_MEMORIA = true;
    CGP_CML();

    MEMORIA_informarEstouroTempoTotal_CGP = true;
    CGP_CML();
```

```
 while(true){
    //vel <3
    MEMORIA_informaVelocidadeMenorQueTres_CGP = true;
    // dir
    MEMORIA_informaChaveSeletoraLadoDireito_CGP = true;
    MEMORIA_informaChaveSeletoraLadoEsquerdo_CGP = false;
    //estado inicial
    CGP_informarProximoEstadoDireita_MEMORIA = 0;
    //chave Kloan Selecionada
    MEMORIA_informaChaveKLOANSelecionada_CGP = true;
    // botao dir console e lateral nao acionados
    MEMORIA_informaBotaoFechamentoDireitoConsole_CGP = false;
    MEMORIA_informaBotaoFechamentoDireitoLateral_CGP = false;
    CGP_CML();

    MEMORIA_informaChaveKLOANSelecionada_CGP = FALSE;
    CGP_CML();

    MEMORIA_informaBotaoFechamentoDireitoLateral_CGP = TRUE;
    MEMORIA_informaBotaoFechamentoDireitoConsole_CGP = TRUE;
    CGP_CML();

    CGP_zerarContadorGongo_MEMORIA = true;
    CGP_CML();

    MEMORIA_informarEstouroTempoTotal_CGP = true;
    MEMORIA_informaChaveKLOANSelecionada_CGP = true;
    // botao dir console e lateral variando
    MEMORIA_informaBotaoFechamentoDireitoConsole_CGP = false;
    MEMORIA_informaBotaoFechamentoDireitoLateral_CGP = false;
          KISOL = nondet_bool();
    CGP_CML();

  while(t<30){
    MEMORIA_informaChaveKLOANSelecionada_CGP = nondet_bool();
    // botao dir console e lateral variando
    MEMORIA_informaBotaoFechamentoDireitoConsole_CGP=nondet_bool();
    MEMORIA_informaBotaoFechamentoDireitoLateral_CGP=nondet_bool();
    KISOL = nondet_bool();
    CGP_CML();

    assert(CGP_informarProximoEstadoDireita_MEMORIA <=5);
    assert(CGP_informarProximoEstadoDireita_MEMORIA <=4);

     t++;
    }
  }
    return 0;
}
```

Após a execução da verificação na ferramenta, o resultado obtido foi:

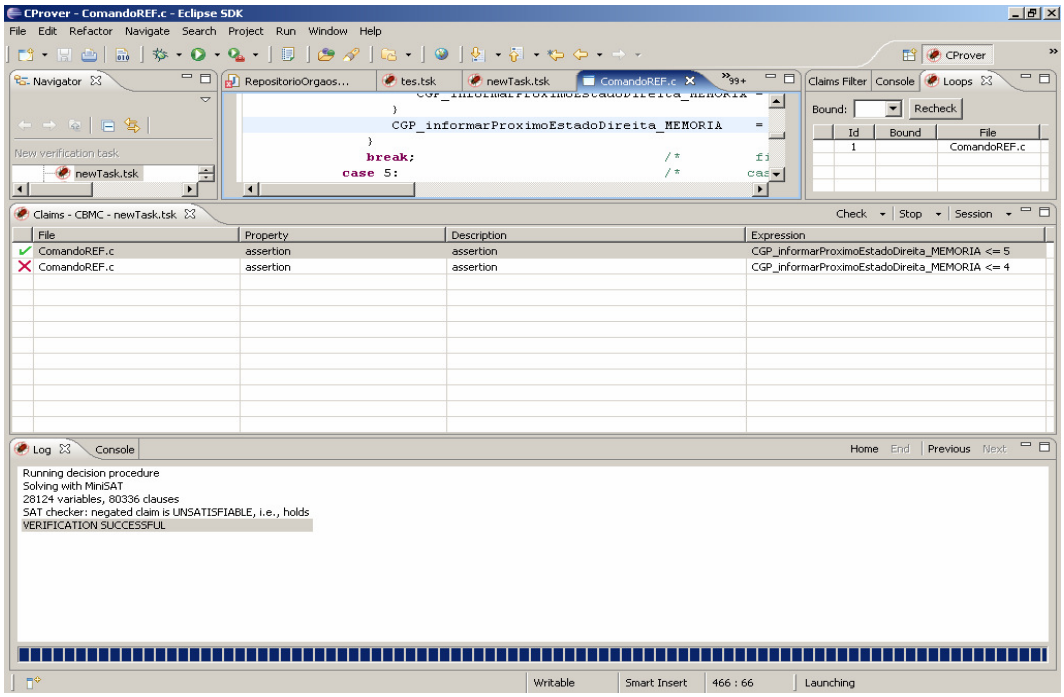

Figura 5.6 - Resultado da verificação de alcançabilidade no CBMC

A partir do resultado da verificação mostrado na Figura 5.6, onde o **assert(CGP_informarProximoEstadoDireita_MEMORIA<=5)** passou sem problemas, sendo marcado com um V, é mostrado que a variável de estado nunca assume o valor 6, ou seja, nunca alcança o estado 6, pois se tivesse alcançado, a verificação seria falsa e o assert seria marcado com um X, ao contrário do que aconteceu, por exemplo, com a verificação do **assert(CGP_informarProximoEstadoDireita_MEMORIA<=4)**, colocado para mostrar que a estratégia adotada está correta, onde a ferramenta relatou um problema, marcando o **assert** com um X, demonstrando que o estado 5 foi alcançado, por isso aconteceu o relato de um problema, pois o assert verificou que a variável pode assumir o valor 5, resultado mostrado com destaque no "trace" da Figura 5.7, que demonstra o que determinou o problema, mostrando que a variável assumiu o valor 5 em algum momento.

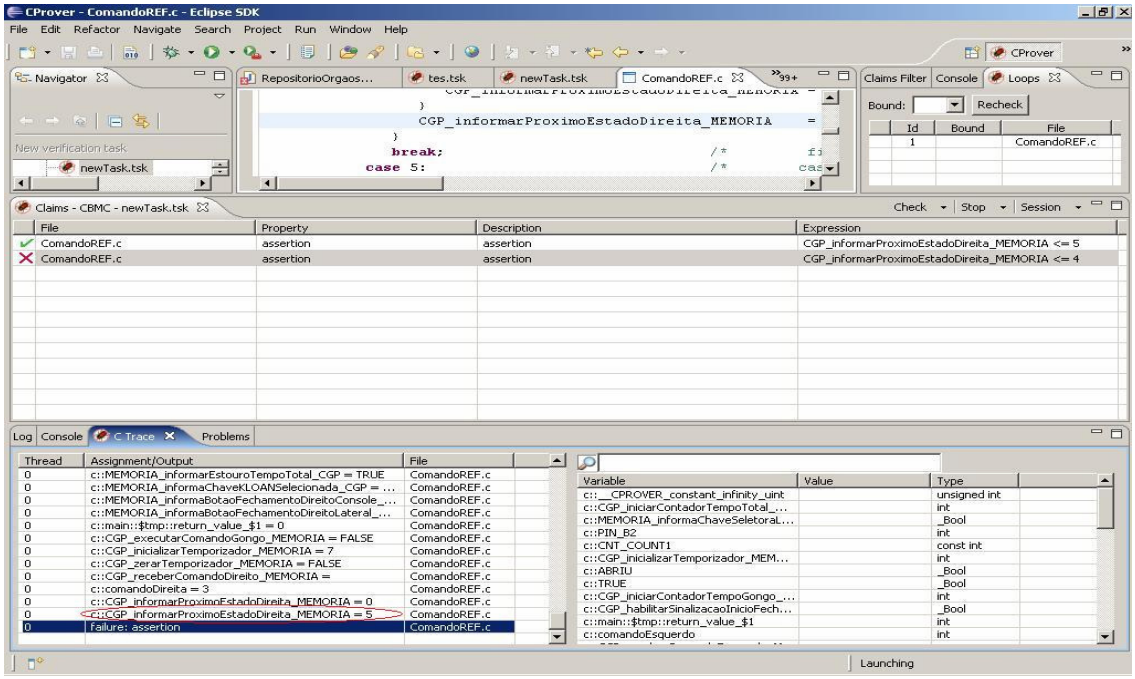

Figura 5.7 - Erro de uma verificação de alcançabilidade no CBMC

O estudo de caso em questão utilizou uma propriedade (propriedade de alcançabilidade) típica de “Static Checkers”, que são verificadores estáticos, ou seja, que fazem uma análise estática do código, reportando erros como casts de tipos incompatíveis, divisão por zero, acesso a array fora do seu limite, etc. Este tipo de propriedade também é possível de ser verificada em Model Checkers, o que aconteceu neste estudo de caso.

Enfim, este estudo de caso mostrou a importância da verificação de modelos em sistemas críticos, como o componente do metrô em estudo, problemas podem ser identificados e corrigidos, garantindo o funcionamento perfeito e o aumento da confiabilidade do sistema, evitando danos consideráveis que uma falha possa vir a proporcionar.

6 - CONCLUSÃO

A técnica de verificação de modelos vem obtendo bastante sucesso nos últimos anos, as indústrias têm, cada vez mais, reconhecido os verificadores de modelos como uma ferramenta promissora para o desenvolvimento de sistemas. Sistemas críticos necessitam da garantia de funcionamento perfeito, uma vez que falhas podem levar a perdas financeiras ou de vidas humanas. Para sistemas concorrentes, a análise ainda é mais complexa que para sistemas seqüenciais. Isto gera a necessidade de um suporte ferramental que de alguma forma, automatize o processo de verificação deste tipo de programa. Muitos estudos ainda estão sendo realizados, pois as ferramentas ainda apresentam muitas limitações. As abordagens utilizadas nas ferramentas analisadas também possuem limitações.

No BMC existe a limitação que os erros podem ser ignorados se limite n não for escolhido corretamente. De qualquer modo, mesmo se n for escolhido pequeno, este método pode ser usado com a finalidade de eliminar erros.

Na abordagem de verificação de modelos com abstração de predicados que usa um provador de teorema, algumas limitações são conhecidas: número exponencial de chamadas a um provador de teoremas, número limitado das construções de C suportadas, suporte limitado para a aritmética do ponteiro e desconsiderar o fluxo de dados. Nem todos os operadores de C podem ser suportados por estes provadores de teorema. O último problema deriva-se do uso da abstração de predicado. Esta técnica depende da abstração das variáveis dos dados e não pode observar o fluxo de dados. Em programas em C para sistemas embarcados, o fluxo de dados é mais importante do que em Drivers e protocolos.

Uma outra abordagem de abstração de predicado é utilizando um resolvente SAT. Este método substitui o uso de provadores de teorema com o uso de um resolvente SAT. A abstração de predicado baseada no SAT tem melhor performance que as abordagens que usam provadores de teorema, desde que uma chamada única a um resolvente SAT pode substituir um número exponencial de chamadas do provador de teorema. Uma outra vantagem da técnica de abstração baseada no SAT é que a maioria das

construções de C podem ser suportadas durante a abstração do programa. A habilidade de suportar construções de programação de baixo nível de C difere das abordagens de outros verificadores de modelos que operam somente sobre um subconjunto pequeno da linguagem C. A única limitação é que a recursão e a alocação dinâmica de memória não estão permitidas. Esta limitação não pode ser evitada, pois programa abstrato booleano requer ser finito, já que o programa em C é transformado em uma fórmula booleana e esta fórmula booleana tem que ser finita. Mas a recursão e a alocação dinâmica de memória são duas construções que são infinitas. Esta abordagem pode ser usada para verificação da linguagem C para sistemas embarcados, se estas duas construções não estiverem presentes no código em C.

O SLAM é uma ferramenta que utiliza a técnica de abstração de predicado usando um provador de teorema, por isso, ainda possui algumas limitações, principalmente com referência a escalabilidade, ou seja, algumas características de C não são suportadas pela ferramenta, isso devido à limitação que um provador de teorema impõe. Programas muito grandes também não são suportados**.** O SLAM diz ser “sound”, ou seja, todo erro verdadeiro é relatado pela análise, mas não é completo, pode relatar erros que não são verdadeiros. Por outro lado, é uma ferramenta que não está disponível publicamente, pois é parte de um produto comercial da Microsoft, o SDV (Static Driver Verifier) do Windows [19].

O BLAST é uma ferramenta semelhante ao SLAM, que verifica propriedades de segurança de programas em C usando a abstração de predicado com um provador de teorema, e com isso, possuindo também os mesmos problemas do SLAM com relação a escalabilidade. Uma diferença chave é o uso da abstração sob-demanda no BLAST. A abstração preguiçosa permite que a descoberta do predicado seja feita localmente e sob-demanda, diferentemente do SLAM que gera a abstração do programa inteiro, ou seja, o BLAST faz somente quando necessário e economiza assim muito espaço e tempo. O SLAM diz suportar funções recursivas, já o BLAST não suporta. O BLAST diz ser “sound”, relata todos os erros verdadeiros, mas também não é completo, pode relatar erros falsos.

O MOPS é uma ferramenta de análise estática verificadora de modelos para software com aplicações críticas de segurança. O MOPS utiliza uma

verificação de modelos baseada em CFG (gráfico do fluxo de controle). O foco principal dos MOPS é o fluxo do controle do programa. Os dois principais objetivos do MOPS são soundness e escalabilidade. O MOPS é mais recomendado para verificar as propriedades de fluxo de controle. O MOPS suporta bem programas grandes, superando o problema da escalabilidade que outros verificadores de modelos possuem. Desde que toda a propriedade não trivial sobre uma linguagem reconhecida por uma máquina de Turing é indecidível, nenhuma ferramenta que verifica uma propriedade não trivial pode ser “sound” e completa [10]. Como o MOPS teve como objetivo ser “sound”, então está inevitavelmente incompleto, pois o MOPS pode gerar falsos erros. Para o MOPS ser “sound”, o programa tem que ser single-threaded, ou seja, não pode ser concorrente, tem que ser seguro de memória e portátil. Considerando estes fatos não é possível usá-lo para verificar modelos em código C para sistemas embarcados.

CBMC é um Bounded Model Checker (verificador de modelos limitado) usado para a análise de programas em C. Utiliza a abordagem BMC, descrita na seção 2.4.1 e usa um resolvente SAT. A principal potencialidade do CBMC é o suporte a maioria das estruturas de C, ou seja, a escalabilidade é um grande diferencial do CBMC. A combinação de uma análise automática, de um grande número de características de C suportadas e uma interface amigável para o programador faz esta ferramenta muito útil. Por priorizar a escalabilidade, a precisão é comprometida, ou seja, a ferramenta pode relatar erros falsos. O domínio da aplicação desta ferramenta inclui softwares embarcados e de simulação de protótipos.

A técnica de abstração baseada no SAT é implementada na ferramenta SatAbs que pode ser invocada dentro do ComFoRT. O verificador de modelos SatAbs possui pequenas limitações nas construções permitidas de C. O SatAbs usa a abstração de predicados como o BLAST, o MAGIC e o SLAM. Mas em vez de usar um provador de teorema, usa um resolvente SAT. O ComFoRT é uma ferramenta de raciocínio que combina o verificador de modelos COPPER, com uma estrutura adicional que permite seu uso efetivo em desenvolvimento de software baseado em componentes. O motor verificador de modelos do ComFoRT, o COPPER, foi construído baseado na ferramenta MAGIC. A característica que distingue o ComFoRT com

abordagem baseada no SAT das ferramentas existentes e as abordagens é a integração de um resolvente SAT na abstração. A abordagem da abstração baseada no SAT traz grandes vantagens já citadas. O ComFoRT ainda está em fase de desenvolvimento, mas promete ser uma ferramenta muito útil em breve, pois características importantes para um verificador de modelos estão sendo agrupadas como a utilização de um resolvente SAT, que não estava incluído na ferramenta MAGIC. Oferece suporte para sistemas concorrentes. Hoje está disponível publicamente uma versão experimental, onde nem todas as características foram testadas adequadamente.

Enfim, como já relatado, é difícil apontar uma ferramenta que seja a melhor, de fato, o que existe, é que para cada situação uma ferramenta pode se encaixar melhor, pois dependendo da situação uma ferramenta pode trazer melhores resultados, tudo isso dependendo das características do código C do programa que se quer analisar e do que se espera da análise, ou seja, as ferramentas não substituem umas as outras, elas podem ser utilizadas em conjunto, extraindo-se o que se tem de melhor em cada uma. Algumas estruturas de C ainda não são suportadas, e quanto mais estruturas uma ferramenta suportar, mais ela estará sujeita a relatar falsos erros, devido à complexidade de suportar algumas características do código em C. Esta limitação com referência a suporte de estruturas de C limita o domínio, como conseqüência, muitas aplicações não podem ser analisadas eficientemente. Por isso, o domínio testado com sucesso ainda é relativamente pequeno, pois algumas delas foram usadas com sucesso apenas para verificação de propriedades de segurança em aplicações de Device Drivers e de protocolos. Ferramentas para código C pertencem a uma área da verificação de modelos que ainda tem muito a evoluir, as ferramentas disponíveis atualmente ainda necessitam de melhorias, ainda possuem vários aspectos a serem melhorados. Várias pesquisas estão sendo feitas, realizadas pelos próprios desenvolvedores ou por pesquisadores da área, no sentido de aprimorar a abordagem atual utilizada pela ferramenta ou substituí-la por uma abordagem melhor, como também em melhorar a própria implementação atual da ferramenta e oferecer cada vez mais recursos.

Este trabalho apresentou a técnica de verificação de modelos, falando sobre os principais conceitos envolvidos, as abordagens utilizadas pelas

ferramentas e as principais ferramentas Model Checkers para a linguagem C, fazendo uma análise de algumas delas, destacando potencialidades e limitações. De forma resumida, as principais contribuições deste trabalho foram:

- Procura de ferramentas Model Checkers para a linguagem C.
- Descrição e análise das abordagens utilizadas pelas ferramentas para a linguagem C.
- Descrição do funcionamento das principais ferramentas.
- Análise das principais características das ferramentas mais utilizadas.
- Aplicação prática de uma ferramenta analisada num estudo de caso referente a um componente de um sistema de metrô [20], relatando os resultados obtidos, as verificações realizadas, assim como os problemas encontrados.

6.1 - TRABALHOS RELACIONADOS

A seguir são mostrados alguns trabalhos relacionados com este trabalho, abaixo está uma breve descrição de cada um deles:

- Analisando uma abordagem para extração de uma modelagem em CSP a partir de especificações em linguagem natural

Este trabalho [21] também foi voltado para o sistema CGP do metrô [20], nele foi definida uma gramática de onde se extraiu todas as entidades bem como seus comportamentos.

- Uma abordagem para extração de especificação CSP a partir de uma implementação em linguagem C

Este trabalho [22] também foi voltado para o sistema CGP do metrô [20], nele foi definida uma abordagem para modelar uma especificação em CSP a partir de uma implementação na linguagem de programação C.

6.2 - TRABALHOS FUTUROS

Em continuidade ao trabalho desta pesquisa, recomenda-se como trabalho futuro um estudo de caso mais abrangente, a aplicação das diversas ferramentas a um sistema real e completo e não a apenas um componente, aplicando os conceitos obtidos neste trabalho, utilizando o que cada uma oferece de melhor e relatar os resultados obtidos. Poderia ser feito ainda um estudo mais aprofundado da possibilidade de se utilizar verificações através de lógica temporal nas ferramentas, e conseqüentemente, um estudo para formulação de propriedades importantes na lógica temporal que um determinado sistema a ser analisado precisa validar, submeter estas propriedades às diversas ferramentas e relatar os resultados. Outra proposta seria trabalhar na melhoria de uma determinada ferramenta que esteja disponível publicamente, estudando o que pode ser melhorado em relação às abordagens utilizadas, como também na própria implementação da ferramenta.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] The *SLAM Project*. Disponível em http://research.microsoft.com/slam/. Acessado em 16.05.2007.

[2] MTC (Models and Theory of Computation): BLAST Project. Disponível em http://mtc.epfl.ch/software-tools/blast/. Acessado em 15.05.2007.

[3] Andrews, G.R. Multithreaded, Parallel, and Distributed Programming. Addison-Wesley, 2000

[4] *MAGIC*. Disponível em http://www.cs.cmu.edu/~chaki/magic/. Acessado em 16.05.2007.

[5] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav, "Predicate abstraction of ANSI–C programs using SAT." Formal Methods in System Design (FMSD), volume 25, September–November 2004, pp. 105–127.

[6] Model Checking. Disponível em http://en.wikipedia.org/wiki/Model_checking. Acessado em 16.05.2007.

[7] MOPS (MOdelchecking Programs for Security properties). Disponível em http://www.cs.berkeley.edu/~daw/mops/. Acessado em 25.07.2007.

[8] CBMC Homepage. Disponível em http://www.cs.cmu.edu/~modelcheck/cbmc/. Acessado em 15.07.2007.

[9] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav, "Satabs: Sat-based predicate abstraction for ANSI-C." Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005), volume 3440 of Lecture Notes in Computer Science, 2005, pp. 570–574.

[10] Chen and D. Wagner, "MOPS: An Infrastructure for Examining Security Properties of Software." ACM CCS 2002, 2002, pp. 235–244.

[11] Ferreira, Nelson França Guimarães, "Verificação Formal de Sistemas Modelados em Estados Finitos" 2005. Disponível em http://www.teses.usp.br/teses/disponiveis/3/3141/tde-19092006-134100/.

[12] SATABS Homepage. Disponível em http://www.verify.ethz.ch/satabs/documentation.html. Acessado em 02.08.2007.

[13] J. Ivers and N. Sharygina. Overview of ComFoRT: A Model Checking Reasoning Framework. Technical Report CMU/SEI-2004-TN-018, SEI, CMU, 2004.

[14] Chaki, S.; Clarke, E.; Groce, A.; & Strichman, O. "Predicate Abstraction with Minimum Predicates," 19-34. 2003.

[15] Chaki, S.; Ouaknine, J.; Yorav, K.; & Clarke, E. “Automated Compositional Abstraction Refinement for Concurrent C Programs: A Two-Level Approach.” SoftMC 2003: Workshop on Software Model Checking (in Electronic Notes in Theoretical Computer Science [ENTCS], volume 89). Boulder, Colorado, July, 2003. New York, NY: Elsevier Science, 2003.

[16] Clarke, E. M.; Grumberg, O.; & Long, D. E. “Model Checking and Abstraction.” ACM Transactions on Programming Languages and Systems 16, 5 (September 1994): 1512-1542.

[17] Clarke, E.; Grumberg, O.; & Peled, D. Model Checking. Cambridge, MA: MIT Press, 1999.

[18] Chaki, S.; Clarke, E.; Ouaknine, J.; & Sharygina, N. “Automated, Compositional and Iterative Deadlock Detection,” 201-210. Proceedings of Formal Methods and Models for Codesign (MEMOCODE ’04). San Diego, CA, June 23-25, 2004. Madison, WI: Omnipress, 2004.

[19] Static Driver Verifier. Disponível em http://www.microsoft.com/whdc/devtools/tools/sdv.mspx. Acessado em 21.07.2007.

[20] Especificação Técnica dos Requisitos de Software para a Controladora Geral de Portas. Projeto Santiago Linha 2, Chile. 2005.

[21] GOMES, A.; “Analisando uma abordagem para extração de modelagem em CSP a partir de especificação de requisitos”. Trabalho de Graduação. Universidade Federal de Pernambuco. Agosto 2007

[22] MILLANO, F.; “Uma abordagem para extração de especificação CSP a partir de uma implementação em linguagem C”. Trabalho de Graduação. Universidade Federal de Pernambuco. Agosto 2007

[23] Montenegro, P. Artefatos do Projeto CGP- Metrô. Disponível em: http://www.cin.ufpe.br/~pmr/tg.

ASSINATURAS

Este Trabalho de Graduação é resultado dos esforços do aluno Pedro Montenegro Rodrigues, sob a orientação do professor Alexandre Cabral Mota, sob o título: "*Model Checkers: Uma análise de ferramentas para a linguagem de programação C*". Todos abaixo estão de acordo com o conteúdo deste documento e os resultados deste Trabalho de Graduação.

_____

Pedro Montenegro Rodrigues

_____

Alexandre Cabral Mota