Universidade do Minho

Mestrado Integrado de Engenharia Informática

Verificação Formal

VeriFast Program Verifier

Bruno Cancelinha
Marcelo Miranda

27 de Junho de 2018

1 Introdução

VeriFast é uma ferramenta de verificação para programas C e Java que visa verificar propriedades de correção tanto em programas *single-threaded* como *multi-threaded*, recorrendo a anotações que descrevem pré-condições e pós-condições . É importante notar que a ferramenta em estudo não tem em conta propriedades de *liveness*, sendo impossível verificar a terminação de um programa. Um programa verificado pelo VeriFast é considerado *sem erros* quando

- não efetua acessos ilegais à memória, como ler ou escrever fora dos limites de um *array* ou em memória que já foi libertada;
- não possui *data races*, isto é, acessos concorrentes não sincronizados em que pelo menos um deles é uma escrita;
- e cujas funções respeitam as pré-condições e pós-condições especificadas pelo programador.

Estes erros são detetados através de uma execução simbólica do programa, no qual o corpo de cada função é executado isoladamente partindo do estado descrito na pré-condição. Para cada comando de uma função, é necessário verificar que existem permissões no estado simbólico tal que os vários acessos de memória realizados são legais. É ainda necessário atualizar o estado para refletir os efeitos do comando.

Esta ferramenta distingue-se de ferramentas como o *Frama-C* por ser baseada em lógica de separação, uma extensão à lógica de Hoare. A lógica de separação permite raciocinar sobre manipulação de apontadores, possibilitando a descrição de estruturas de dados dinâmicas, *aliasing* de apontadores e partilha de dados por varias *threads*.

Começaremos este relatório por demonstrar como verificar um pequeno programa C de forma a expor os leitores ao funcionamento da ferramenta (Secção 2). De seguida, fazemos uma apresentação, sem grande detalhe, ao uso do VeriFast para verificar programas *multi-threaded* (Secção 3). Para terminar, analisamos o modo como o VeriFast recorre à execução simbólica para realizar a verificação de programas (Secção 4), terminando com algumas observações sobre a ferramenta (Secção 5).

2 Verificação de um pequeno exemplo

Nesta secção iremos verificar um pequeno programa C de forma a ganhar alguma intuição sobre o funcionamento da ferramenta. Apenas funções com um contrato são verificadas portanto, como queremos verificar a função *main*, devemos adicionar-lhe o devido contrato.

```
struct account {
    int balance;
};

int main()
    //@ requires true;
    //@ ensures true;
{
    struct account *myAccount = malloc(sizeof(struct account));
    // if (myAccount == 0) { abort(); }
    myAccount->balance = 5;
    free(myAccount);
    return 0;
}
```

O programa em causa começa por alocar uma estrutura de dados, prosseguindo a alterar o seu conteúdo. Antes de terminar o programa, a memória relativa à estrutura é libertada. Para verificar este programa, o VeriFast realiza uma execução simbólica, analisando todas as possíveis ramificações. Como podemos ver através da mensagem de erro abaixo, o VeriFast determina corretamente que este programa não é válido visto que, no caso da alocação falhar, este tenta ainda alterar o conteúdo da estrutura, realizando um acesso à memória inválido.

```
No matching heap chunks: account_balance(myAccount)
```

Nesta mensagem, o VeriFast informa que não existe nenhum *chunk* `account_balance(myAccount)` no estado simbólico do programa. A rotina `malloc`, quando bem sucedida, produz um *chunk* `malloc_block_account(myAccount)` e um *chunk* `account_balance(myAccount)`. O primeiro indica que `myAccount` aponta para uma estrutura na *heap* e é necessário para libertar a memória dessa estrutura. Isto é importante para impedir que seja realizado um `free` numa estrutura de dados que esteja guardada na *stack* ao invés da *heap*. O segundo *chunk*, quando presente, indica que o programa tem permissões para aceder ao campo `balance` da estrutura.

Para corrigir o programa, bastaria descomentar a linha em comentário no corpo da função, abortando o programa nos casos em que a rotina `malloc` falha.

Para terminar este exemplo, vamos mover a atribuição do campo `balance` para uma nova função `account_set_balance`. Dado que a verificação da função *main* requer que todas as rotinas que são chamadas possuam também contrato, começamos por introduzir o mesmo contrato que definimos para a função *main*.

```
void account_set_balance(struct account *myAccount, int newBalance)
    //@ requires true;
    //@ ensures true;
{
    myAccount->balance = newBalance;
    //@ leak account_balance(myAccount, _);
}

int main()
    //@ requires true;
    //@ ensures true;
{
    struct account *myAccount = malloc(sizeof(struct account));
    if (myAccount == 0) { abort(); }
    account_set_balance(myAccount, 5);
    free(myAccount);
    return 0;
}
```

Tendo em conta o que já foi visto e dado que cada função é verificada isoladamente, facilmente concluímos que a função `account_set_balance` não é verificada devido à ausência do `chunk` `account_balance` no estado simbólico.

```
void account_set_balance(struct account *myAccount, int newBalance)
    //@ requires account_balance(myAccount, _);
    //@ ensures true;
{
    myAccount->balance = newBalance;
    //@ leak account_balance(myAccount, _);
}
```

Adicionando o respetivo *chunk* à pré-condição, a atribuição é verificada com sucesso. Como no fim da verificação existe ainda o *chunk* `account_balance` no estado simbólico, o VeriFast requer que declaremos explicitamente que a rotina possui um *leak*.

No entanto, embora a função `account_set_balance` seja verifica com sucesso, a função *main* falha. Isto acontece porque o `account_set_balance` consome o *chunk* `account_balance`, o qual é necessário para realizar o `free`. Este problema pode ser corrigdo retornando o *chunk* para o invocador da função através da pós-condição.

```
void account_set_balance(struct account *myAccount, int newBalance)
    //@ requires account_balance(myAccount, _);
    //@ ensures account_balance(myAccount, newBalance);
{
    myAccount->balance = newBalance;
}
```

É importante ainda notar que o parâmetro `newBalance` se encontra na posição relativa ao valor do campo `balance`. Isto significa que, aquando o retorno da função, o campo balance vai ter o mesmo valor que o parâmetro.

Para permitir a especificação de programas mais ricos, o VeriFast permite que o programador defina tipos de dados indutivos, funções recursivas sem *side-effects* sobre estes tipos de dados, assim como predicados abstratos em lógica de separação. Quanto à verificação destas especificações, o programador pode escrever *lemma functions* – funções que provam que a pré-condição implica a pós-condição. O verificador confere que também estas funções não possuem *side-effects* e garante a sua terminação.

Estes assuntos, no entanto, não estão no âmbito do nosso relatório mas podem ser estudados detalhadamente no tutorial da ferramenta [3].

3 Verificação de programas *multi-threaded*

O VeriFast permite verificar programas *multi-threaded* que partilham a *heap*, *mutexes* e outros recursos pelas várias *threads*. Para este efeito, o Verifast suporta permissões fracionárias e, através de uma biblioteca, suporta também *counting permissions*.

Nesta secção iremos debruçar-nos sobre o uso de permissões fracionárias, enunciando algumas das funcionalidades suportadas para facilitar o seu uso, assim como as suas limitações.

3.1 Permissões fracionárias

As permissões fracionárias são usadas para suportar acesso de leitura a memória partilhada. Cada *chunk* da memória possui um coeficiente, um número real que varia entre zero (exclusive) e um (inclusive). O coeficiente *default* é 1 e é tipicamente omitido. Um *chunk* com coeficiente 1 representa uma permissão completa, ou seja, permissão para realizar acessos tanto de escrita como leitura. Um *chunk* com coeficiente menor que 1 representa uma permissão fracionária e permite apenas acessos de leitura.

```
int read_cell(int *cell)
    //@ requires [?f]integer(cell, ?v);
    //@ ensures [f]integer(cell, v) &*& result == v;
{
    return *cell;
}
```

Como exemplo, vejamos a função acima que requer uma fração arbitrária `f` do *chunk* `integer` de forma a permitir a leitura do valor contido em `cell`, retornando a mesma fração para o invocador.

Imaginemos agora um cenário em que temos duas *threads* que vão derivar valores (somatório, média, produtório, etc) a partir da mesma estrutura de dados. Para isto, as função que derivam os valores devem especificar uma fração até 1/2 na pré-condição, permitindo assim que ambas as

threads consigam realizar leituras sobre a estrutura de dados partilhada.

Para facilitar a aplicação de permissões fracionárias a predicados definidos pelo programador, o VeriFast suporta *precise predicates*, assim como *dummy fractions* para facilitar a partilha ilimitada de um recurso, útil em casos nos quais reagrupar as frações não é necessário.

Permissões fracionárias são suficientes em muitos dos cenários de partilha de recursos. No entanto, um cenário em que não são aplicáveis é quando o programa a verificar utiliza *reference counting* para sincronizar os acessos a um recurso. Neste cenário, é tipicamente usado outro método de gestão de permissões, conhecido por *counting permissions* [1].

4 Execução simbólica

A verificação é baseada na execução simbólica de cada função, determinando quais os inputs que causam os vários comportamento do programa. Um interpretador segue o programa, assumindo valores simbólicos para os inputs ao invés de obter valores reais como aconteceria numa execução normal do programa. À medida que a função é verificada, são derivadas restrições e expressões para descrever o conjunto de valores que estes valores simbólicos podem assumir. Com estas expressões e restrições é possível, por exemplo, identificar quais os resultados possíveis em cada *branch* condicional.

4.1 Estados simbólicos

O VeriFast verifica modularmente os programas, executando simbolicamente cada rotina e recorrendo aos contratos das restantes rotinas para verificar as respetivas invocações. A execução simbólica é em muito semelhante à execução concreta exceto no facto de, ao invés de usar valores concretos, são usados termos de um *SMT solver* que contêm símbolos lógicos. No inicio da execução simbólica de uma rotina é usado um novo símbolo lógico para representar cada um dos parâmetros da rotina.

Um estado simbólico $\sigma = (\Sigma, \mathtt{h}, \mathtt{s})$ consiste em *path conditions* Σ, uma *symbolic heap* `h`, e uma *symbolic store* `s`. As *path conditions* são o conjunto de fórmulas em lógica de primeira ordem usadas para descrever os valores dos símbolos lógicos que aparecem na *symbolic store* e na *symbolic heap*. A *heap* simbólica contém *heap chunks*. Um *heap chunk* pode dizer respeito a um campo de uma estrutura ou a um predicado definido pelo utilizador. A *symbolic store* mapeia os nomes das variáveis locais para termos que representam os seus valores.

4.2 Execução de uma rotina

A execução simbólica de uma rotina começa por produzir a pré-condição, verificando depois o corpo da rotina, e finalmente consumir a pós-condição. Produzir uma asserção significa adicionar os *chunks* e assunções descritas por essa asserção ao estado simbólico. Inversamente, consumir uma asserção significa remover os *chunks* referidos da *symbolic heap* e verificar as assunções descritas na asserção contra as *path conditions*.

Verificar uma chamada a uma rotina significa consumir a pré-condição dessa rotina, escolher uma variável livre para representar o valor de retorno e produzir a pós-condição. A chamada termina fazendo *binding* do valor de retorno na *symbolic store* da função que chamou a rotina.

Verificar uma rotina significa fazer *binding* dos parâmetros para variáveis livres, produzir a pré-condição, guardar a *symbolic store* resultante s', verificar o corpo da rotina sobre a *symbolic store* original, restaurar a *symbolic store* s' e fazer *binding* do valor de retorno. Por fim, consome-se a pós-condição. A rotina é válida se existir pelo menos um conjunto de transições tal que o estado inicial não leva a `abort`, que significa que um erro foi encontrado.

5 Observações finais

Gostaríamos de começar por realçar que o VeriFast possui um excelente tutorial, no qual as várias funcionalidades são descritas de forma detalhada e clara, tornando o processo de aprendizagem gratificante. Foi devido a isto que decidimos não explorar alguns dos tópicos abordados nas aulas como invariantes, predicados, funções lema e tipos indutivos. Preferimos por isso abordar brevemente alguns tópicos que não foram estudados e julgamos serem importantes como a verificação de programas *multi-threaded* e o modo como a execução simbólica é realizada.

Da experiência que obtivemos com ferramenta, a maioria da qual obtida a resolver os exercícios do tutorial, concluímos que a ferramenta é prática e intuitiva e que o IDE que a acompanha é extremamente útil no processo de verificação, permitindo realizar a execução simbólica passo a passo, oferecendo uma melhor compreensão dos erros que surgem.

A parte da ferramenta que achamos menos atrativa foi a escrita de invariantes – um assunto não abordado neste relatório – dado que exige uma panóplia de predicados para percorrer as estruturas indutivas e assim verificar que possuímos acesso à localização da memória pretendida.

Referências

[1] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frederic Vogels, Willem Penninckx, and Frank Piessens. *VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java.* `https://people.cs.kuleuven.be/%7Ebart.jacobs/nfm2011.pdf`

[2] Bart Jacobs, and Frank Piessens. *The VeriFast Program Verifier.* `https://people.cs.kuleuven.be/%7Ebart.jacobs/verifast/verifast.pdf`

[3] Bart Jacobs, Jan Smans, and Frank Piessens. *The VeriFast Program Verifier: A Tutorial.* `https://zenodo.org/record/1068185/preview/tutorial.pdf`