Guião para aula laboratorial de Verificação Formal (2018/19)

CBMC

Esta aula é dedicada à familiarização com o CBMC, uma ferramenta para a verificação formal de programas ANSI-C baseada em *Bounded Model Checking*. Esta ferramenta faz a detecção de erros de execução de programas C usando *satisfiability solvers*.

Deverá ter instalado o CBMC. O website do CBMC disponibiliza a ferramenta, documentação, assim como alguns tutoriais.

1 Introdução

O CBMC é um *Bounded Model Checker* para verificação formal de programas ANSI C. O seu objectivo é a detecção de erros de execução dos programas C usando SAT solvers.

No *Bounded Model Checking* a relação de transição de uma máquina de estados complexa e sua especificação são desenroladas em conjunto para obter uma fórmula booleana, que é então verificada quanto à satisfatibilidade usando um SAT solver. Se a fórmula for satisfatível, um contraexemplo é extraído do output do procedimento SAT. Se a fórmula não for satisfatível, o programa pode ser desenrolado mais para determinar se existe um contra-exemplo mais longo.

A verificação é realizada transformando o programa num conjunto de equações e passando a equação resultante (da sua conjunção) para um *solver*. As propriedades verificadas são essencialmente propriedade de *safety* (segurança de apontadores, limites de arrays, divisões por zero, etc), mas também asserções especificadas pelo utilizador.

Tal como um compilador, o CBMC recebe o nome dos ficheiro .c na linha de comando. Em seguida, o CBMC converte o programa e funde as definições feitas nos vários ficheiros .c, da mesma forma que um linker. No entanto, em vez de produzir um binário para execução, o CBMC faz execução simbólica do programa.

A ideia chave do BMC é codificar comportamentos limitados do sistema que aproveitam alguma propriedade dada como uma fórmula lógica cujos modelos (se houver algum) descrevem um traço de execução que conduz a uma violação da propriedade.

2 Transformando o programa C numa fórmula lógica

O CBMC reduz o problema de análise do programa à determinação da validade de uma fórmula. Esse processo de transformação do programa num conjunto de equações tem várias etapas.

Simplificação do fluxo de controlo. Assume-se que o programa ANSI-C já está pré-processado, por exemplo, todas as directivas `#define` são expandidas.

Em seguida, substituem-se expressões com efeitos laterais por atribuições equivalentes usando variáveis auxiliares, `break` e `continue` por instruções equivalentes de `goto` e loops `for` e `do-while` por loops `while` equivalentes.

Desenrolar dos ciclos. As construções de loop são desenroladas um deterninado número de vezes.. Estas construções incluem instruções while, chamadas de funções (recursivas) e instruções `goto`.

Conversão numa fórmula lógica. O programa resultante das etapas anteriores consiste apenas em instruções `if` (possivelmente aninhadas), atribuições, *asserts*, *labels* e instruções `goto` (com saltos para a frente).

```
i = a[0];
if (x > 0){
  if (x < 10)
    x = x + 1;
  else
    x = x − 1;
}
assert(y > 0 && y < 5);
a[y] = i;
```

Para converter um programa numa fórmula lógica:

1. O programa é convertido em formato *Static Single Assigment* (SSA), no qual várias versões indexadas de cada variável são usadas (uma nova versão para cada atribuição feita na variável original).

```
i1 = a0[0];
if (x0 > 0){
  if (x0 < 10)
    x1 = x0 + 1;
  else
    x2 = x0 − 1;
  x3 = x0 < 10 ? x1 : x2;
}
x4 = x0 > 0 ? x3 : x0;
assert(y0 > 0 && y0 < 5);
a1[y0] = i1;
```

2. O programa resultante é convertido à *conditional normal form*: uma sequência de instruções da forma `if b then C`, em que `C` é um comando atómico.

```
if (true) i1 = a0[0];
if (x0 > 0 && x0 < 10) x1 = x0 + 1;
if (x0 > 0 && !(x0 < 10)) x2 = x0 − 1;
if (x0 > 0 && x0 < 10) x3 = x1;
if (x0 > 0 && !(x0 < 10)) x3 = x2;
if (x0 > 0) x4 = x3;
if (!(x0 > 0)) x4 = x0;
if (true) assert(y0 > 0 && y0 < 5);
if (true) a1[y0] = i1;
```

Agora, constróiem-se dois conjuntos de fórmulas:

- $\mathcal{C}$ contendo a codificação lógica do programa
- $\mathcal{P}$ contendo as propriedades a serem verificadas

$$
\begin{array}{ll}
\mathcal{C} = \{ & i_1 = a_0[0], \\
& (x_0 > 0 \wedge x_0 < 10) \rightarrow x_1 = x_0 + 1, \\
& (x_0 > 0 \wedge \neg(x_0 < 10)) \rightarrow x_2 = x_0 - 1, \\
& (x_0 > 0 \wedge x_0 < 10) \rightarrow x_3 = x_1, \\
& (x_0 > 0 \wedge \neg(x_0 < 10)) \rightarrow x_3 = x_2, \\
& x_0 > 0 \rightarrow x_4 = x_3, \\
& \neg(x_0 > 0) \rightarrow x_4 = x_0, \\
& a_1[y_0] = i_1 \\
\} & \\
\mathcal{P} = \{ & (y_0 > 0 \wedge y_0 < 5) \ \}
\end{array}
$$

$\mathcal{C}$ e $\mathcal{P}$ são tais que:

$$\mathcal{C} \models \mathcal{P} \quad \text{sse} \quad \text{nenhum traço de execução viola nenhuma propriedade}$$

Relembre que: $\mathcal{C} \models \bigwedge \mathcal{P}$ sse $\mathcal{C} \cup \{\neg \bigwedge \mathcal{P}\} \models \bot$
sse $\bigwedge \mathcal{C} \wedge \neg \bigwedge \mathcal{P}$ é UNSAT

Os modelos de $(\bigwedge \mathcal{C} \wedge \neg \bigwedge \mathcal{P})$ (se existirem) correspondem a traços de execução do programa que conduzem à violação da propriedade.

Envio para um SAT solver. A fórmula $(\bigwedge \mathcal{C} \wedge \neg \bigwedge \mathcal{P})$ é então enviada para um *satisfiability solver*, que poderá ser um

- SMT solver, com as teorias lógicas apropriadas. Neste caso, a fórmula é traduzida para a linguagem do solver ou para SMT-LIB 2.
- SAT solver, que é o método de eleição do CBMC. Neste caso, a fórmula sofre *bit-blasting*, um processo de transformação numa fórmula proprosicional que reproduz o processo que se passa ao nível do hardware.

Conversão de um modelo num contra-exemplo. Se $\bigwedge \mathcal{C} \wedge \neg \bigwedge \mathcal{P}$ é SAT um contra-exemplo é mostrado e o traço correspondente é construído e retornado ao utilizador para inspecção.

3 Desenrolando os ciclos

A ideia básica do CBMC é modelar a execução de um programa até um número limitado de passos. Tecnicamente, isto é conseguido por um processo que consiste essencialmente em desenrolar ciclos.

Os ciclos `while` são desenrolados duplicando o corpo do ciclo n vezes. Cada cópia é protegida usando uma instrução `if` que usa a mesma condição da guarda do ciclo. A instrução `if` é adicionada para o caso em que o ciclo requer menos de n iterações.

Após a última cópia, *pode* ser adicionada uma asserção que assegura que o programa nunca requer mais iterações. Essa assersão usa a condição de ciclo negada e chama-se *unwinding*

assertion.

As *unwinding assertions* são cruciais: elas garantem que o limite de desenrolamento é realmente grande o suficiente. Se a *unwinding assertion* de um ciclo falhar, para uma qualquer execução, deverá aumentar-se (se possível) o limite n para esse ciclo, até que o limite seja grande o suficiente.

Em muitos casos, o CBMC é capaz de determinar automaticamente um limite superior no número de iterações do ciclo. No entanto, essa detecção automática do limite pode falhar se o número de iterações de ciclo for altamente dependente dos dados. Além disso, o número de iterações que são executadas por qualquer ciclo pode ser muito grande ou pode simplesmente ser ilimitado.

Para este caso, o CBMC oferece a opção de linha de comando `--unwind B`, onde `B` é o número máximo de desenrolamentos. Note que o número de desenrolamentos é medido contando o número de *backjumps*. Ou seja, o número de vezes que a guarda do ciclo é avaliada antes do fim do ciclo.

A configuração fornecida com –unwind é usada globalmente para todos os ciclo no programa. É possível definir limites individuais para os vários ciclos do programa (ver manual).

E se o número de desenrolamentos especificado for muito pequeno? Nesse caso, os erros que exigem traços mais longos para ser detectados serão perdidos. Para resolver este problema, o CBMC pode opcionalmente inserir verificações de que o limite dado para o desenrolamento do ciclo é realmente suficientemente grande. Essas verificações são chamadas de *unwinding assertions* e são activadas com a opção `--unwinding-assertions`.

4 Propriedades

O CBMC usa *asserts* para especificar as propriedades do programa. Asserções são propriedades do estado num determinado local do programa. As asserções são frequentemente escritas pelo programador usando a macro `assert`.

Além das asserções escritas pelo programador, asserções para propriedades específicas também podem ser geradas automaticamente pelo CBMC, aliviando o trabalho do programador.

O CBMC vem com um gerador de asserções, que executa uma análise estática conservadora para determinar os locais do programa que potencialmente contêm um *bug*. Este gerador pode gerar asserções para a verificação das seguintes propriedades:

- *Buffer overflows.* Para cada acesso a arrays, verifiqua se os limites superior e inferior são violados. Opção: `--bounds-check`
- *Pointer safety.* Procura acessos à memória através de apontadores inválidos.
  Opção: `--pointer-check`
- *Memory leaks.* Verifica se o programa constrói estruturas de dados alocadas dinamicamente que são subsequentemente inacessíveis.

- *Division by zero.* Verifica se existe uma divisão por zero no programa.
  Opção: `--div-by-zero-check`
- *Not-a-Number.* Verifique se o cálculo de vírgula flutuante pode resultar em NaNs.
  Opção: `--nan-check`
- *Arithmetic overflow.* Verifica se ocorre um overflow durante uma operação aritmética.
  Opções: `--signed-overflow-check` e `--unsigned-overflow-check`
- *Undefined shifts.* Verifica se há shifts com distância excessiva.
  Opção: `--undefined-shift-check`

5 Exemplos para explorar

O slides *"CBMC by example"*, apresentados na aula, contém uma serie de exemplos e vários desafios para analisar e resolver. Esses exemplos são acompanhados de ficheiros C disponíveis no wiki da disciplina.

Siga as instruções indicadas nos slides, analise estes ficheiros com o CBMC, e acrescente/altere as anotações conforma achar adequado.