



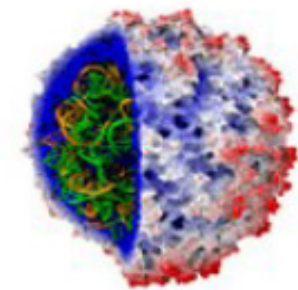
XVI Simpósio em Sistemas Computacionais de Alto Desempenho

Verificação de *Kernels* em Programas CUDA usando *Bounded Model Checking*

Phillipe Pereira, Higo Albuquerque, Hendrio Marques, Isabela Silva,
Vanessa Santos, Celso Carvalho, Ricardo Ferreira, Lucas Cordeiro

Plataforma CUDA

- Desenvolvida pela NVIDIA
- GPU como *hardware*
 - Milhões de dispositivos no mercado
 - Alto poder computacional
 - Inicialmente voltada para processamento gráfico de jogos
- Atualmente abrange áreas da:
 - Biomedicina
 - Controle de tráfego aéreo
 - Simulações meteorológicas
- Necessidade de garantir corretude



Erros em CUDA

- Usada por linguagens como C, C++, Fortran
- Erros de programação
 - Bloqueio fatal, estouro aritmético e divisão por zero
 - Causam resultados errados no processamento do programa
 - Difíceis de identificar em CUDA devido ao alto número de operações paralelas
- Erros comuns à plataforma CUDA
 - Condições de corrida, compartilhamento de memória e divergência de barreira

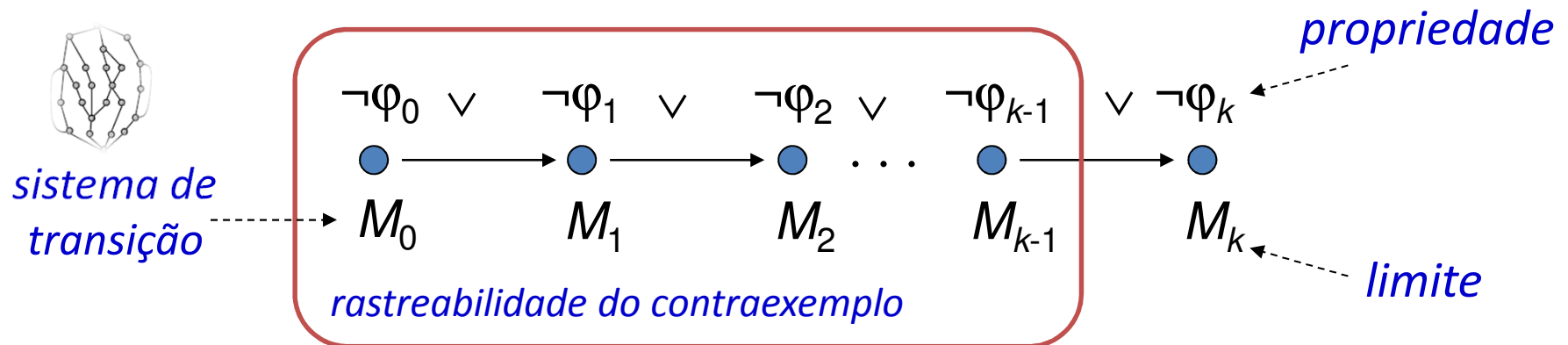
Objetivos deste Trabalho

Verificar propriedades em programas CUDA

- Aplicar a técnica de verificação de modelos limitada baseada nas teorias do módulo da satisfatibilidade
- Desenvolver um modelo operacional para plataforma CUDA (MOC)
 - Integrar o MOC ao *Efficient SMT-Based Context-Bounded Model Checker* (ESBMC)
- Comparar os resultados obtidos com ferramentas estado da arte na verificação de programas CUDA

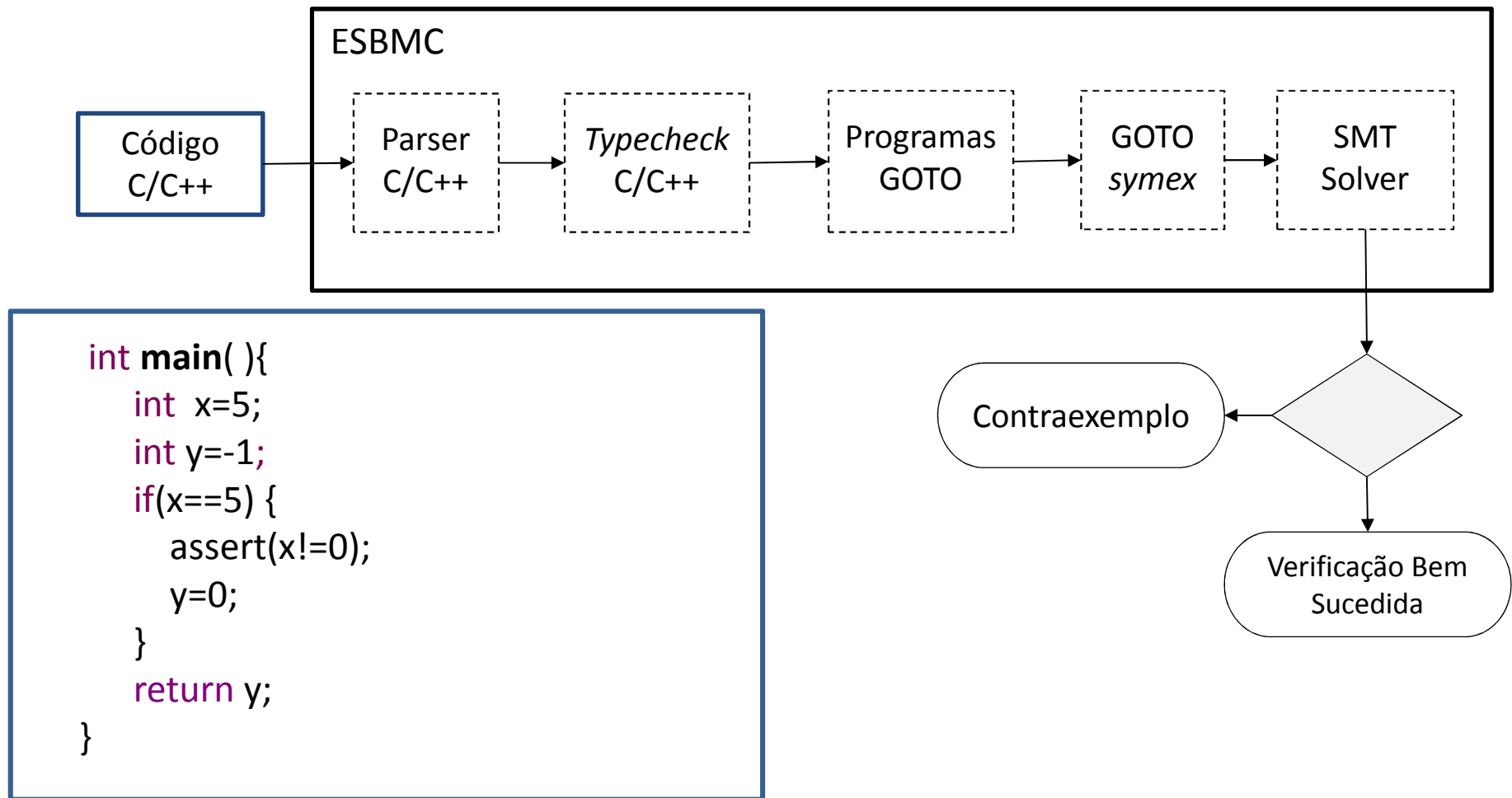
Verificação de Modelos Limitados

- Do inglês, *Bounded Model Checking* (BMC) checa a negação de uma propriedade em uma determinada profundidade

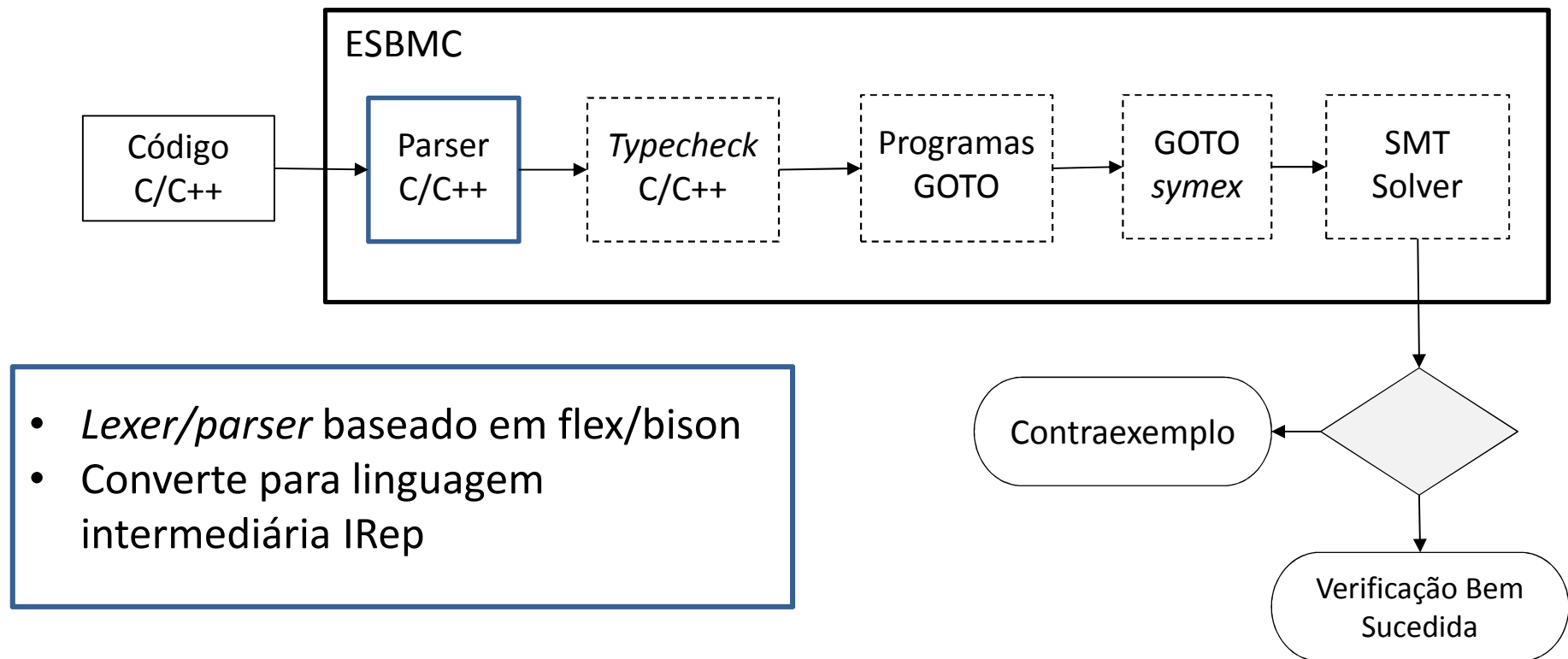


- Sistema de transição M desdobrando k vezes
 - Para programas: *loops*, vetores, ...
- Traduzido em uma condição de verificação ψ tal que ψ é satisfatível sse ϕ tem um contraexemplo de profundidade menor ou igual a k

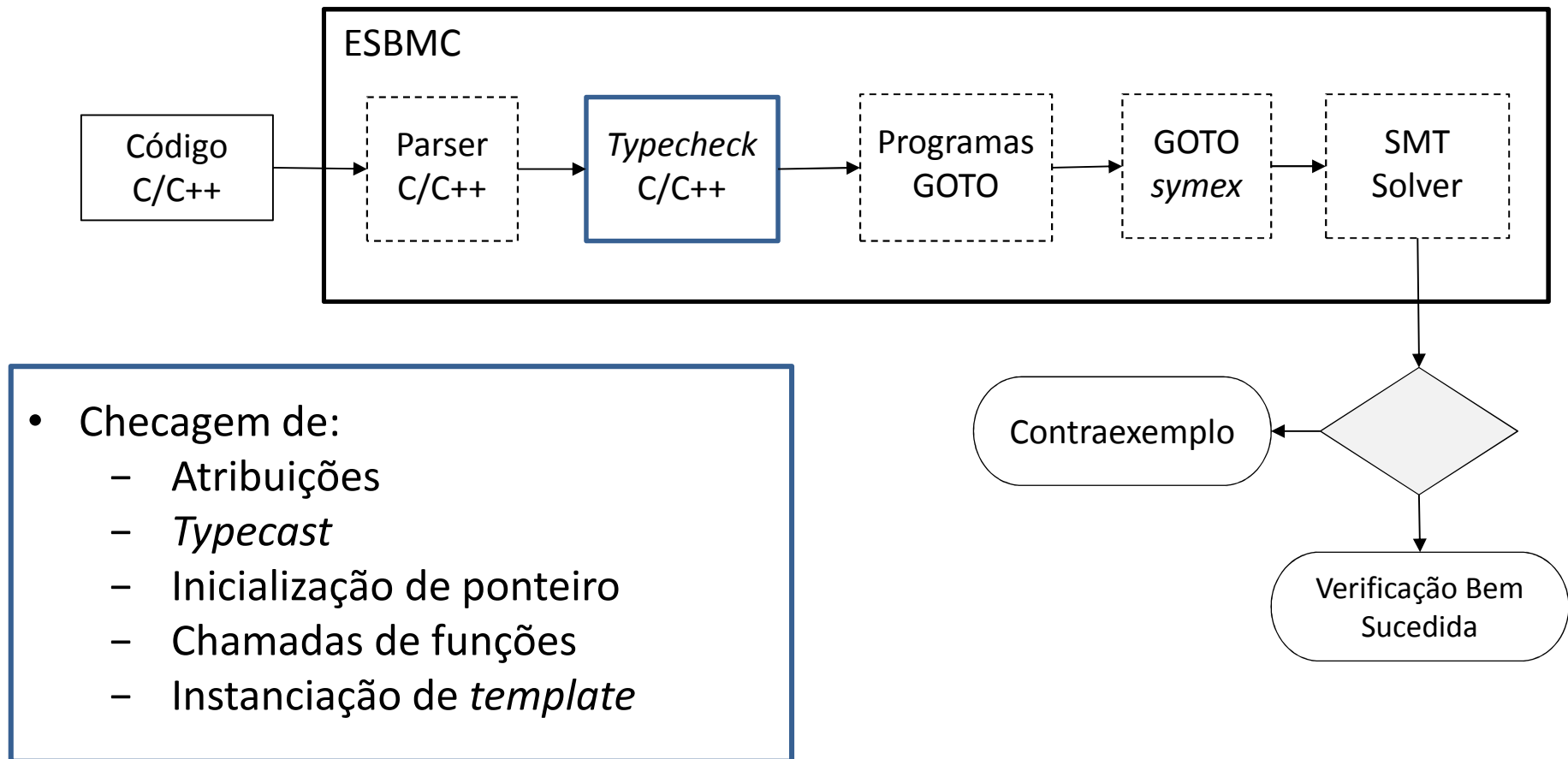
ESBMC - Arquitetura



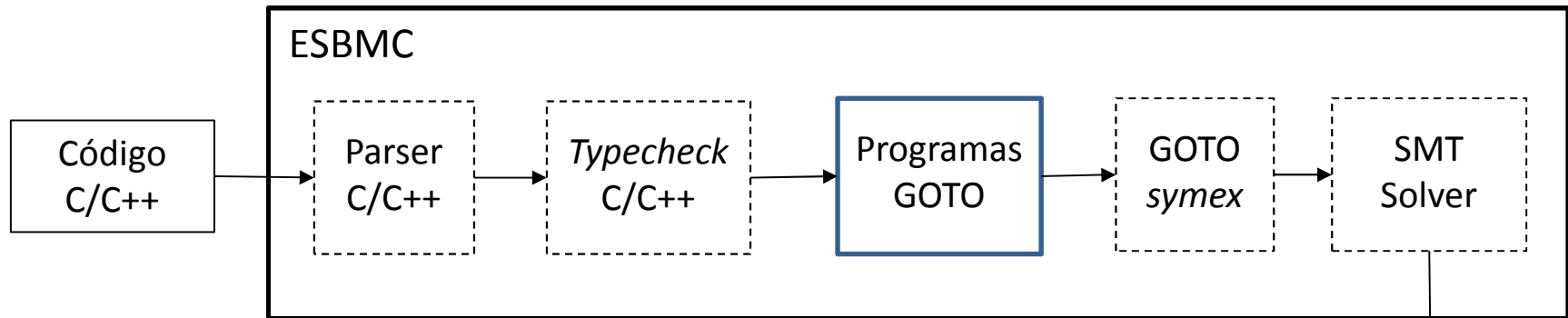
ESBMC - Arquitetura



ESBMC - Arquitetura

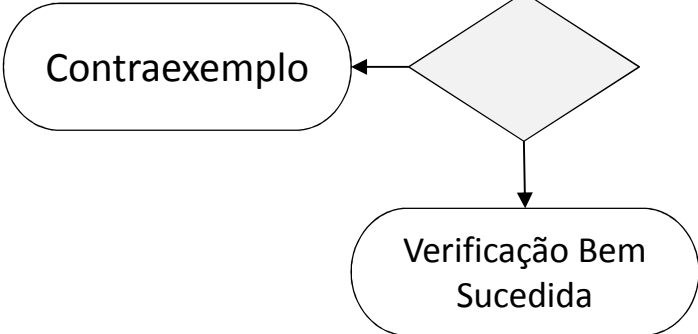


ESBMC - Arquitetura

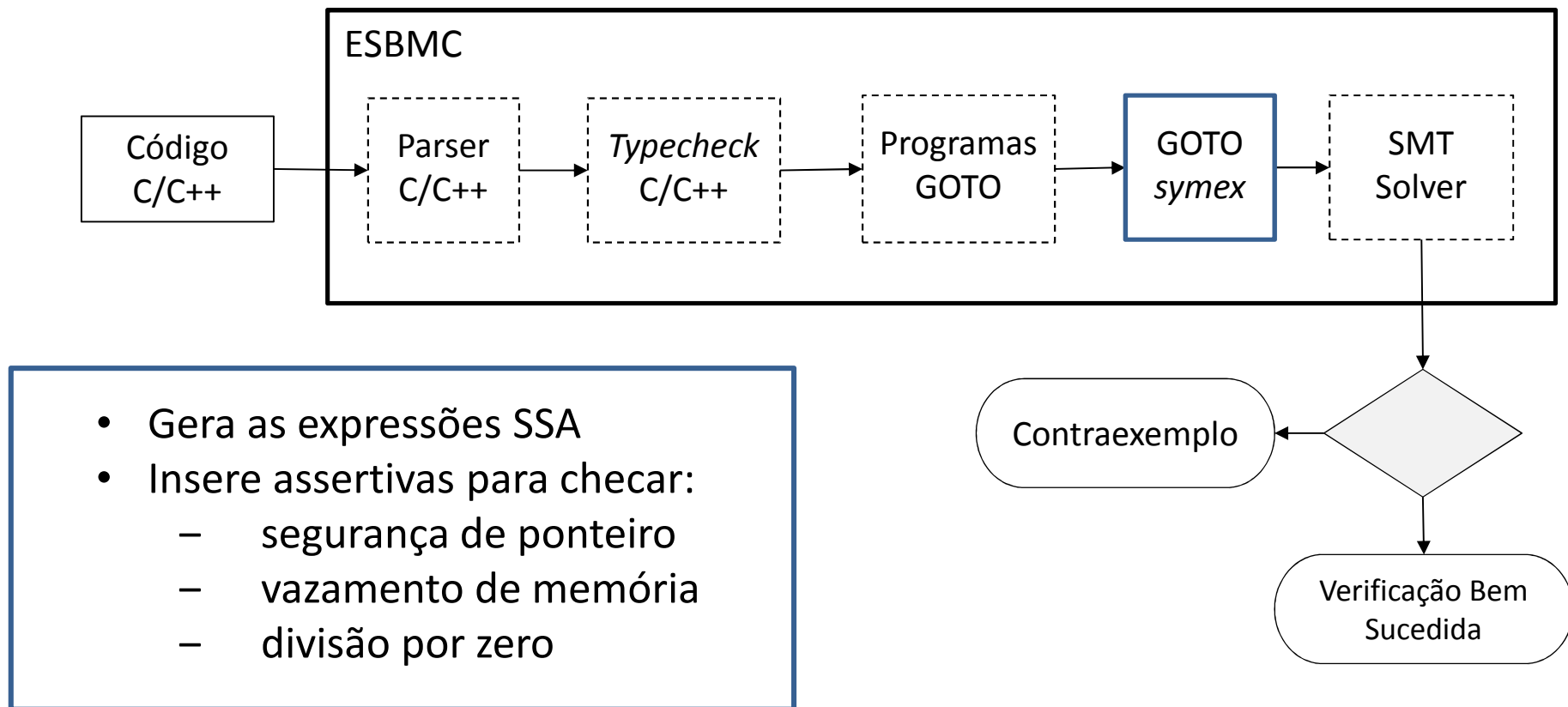


Conversão de IRep para programa GOTO

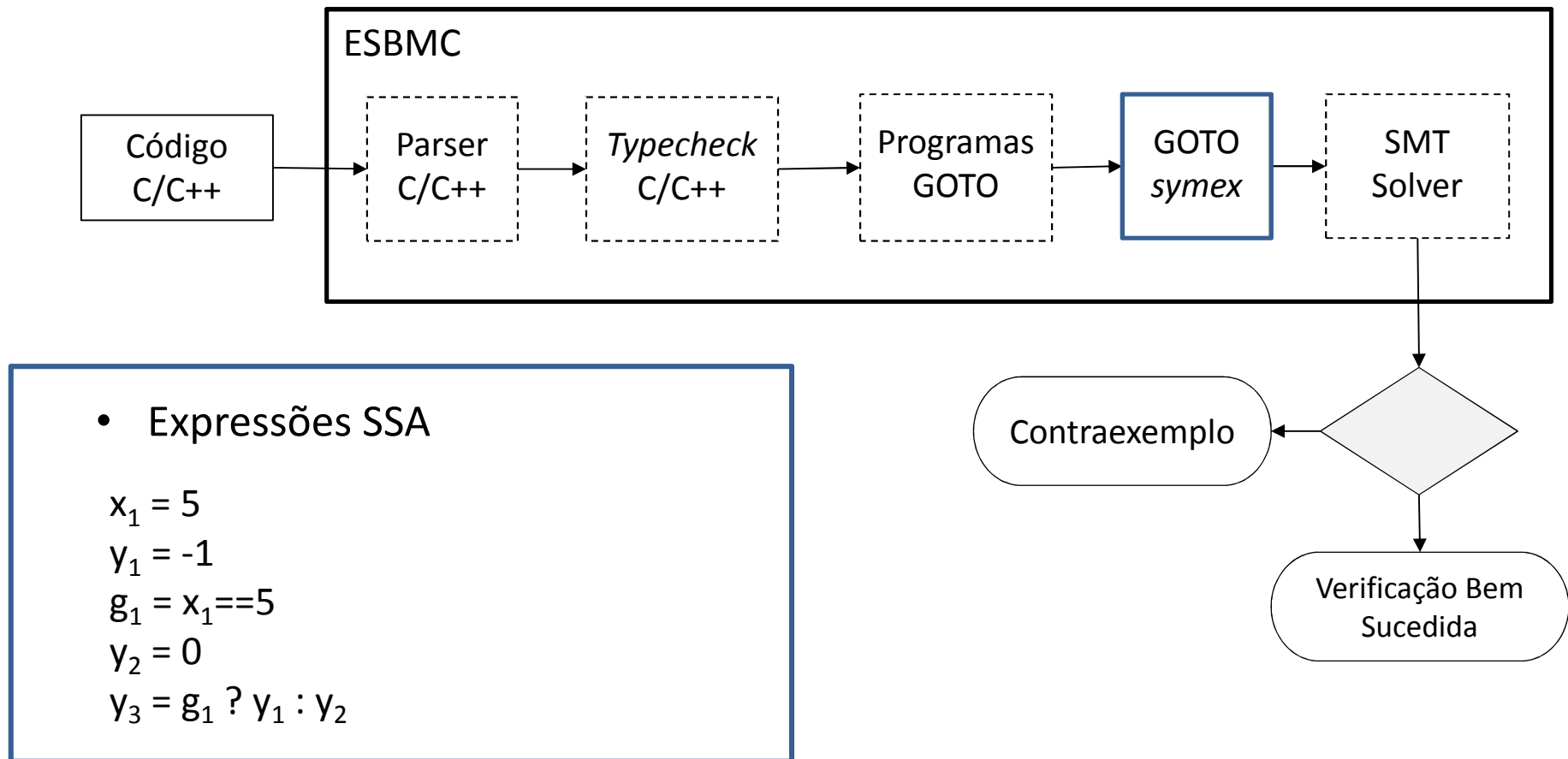
```
main( ) (c::main):  
  int x,y;  
  x=5;  
  y=-1;  
  IF !(x==5) THEN GOTO L1  
    y=0;  
  L1: return y;  
END__FUNCTION
```



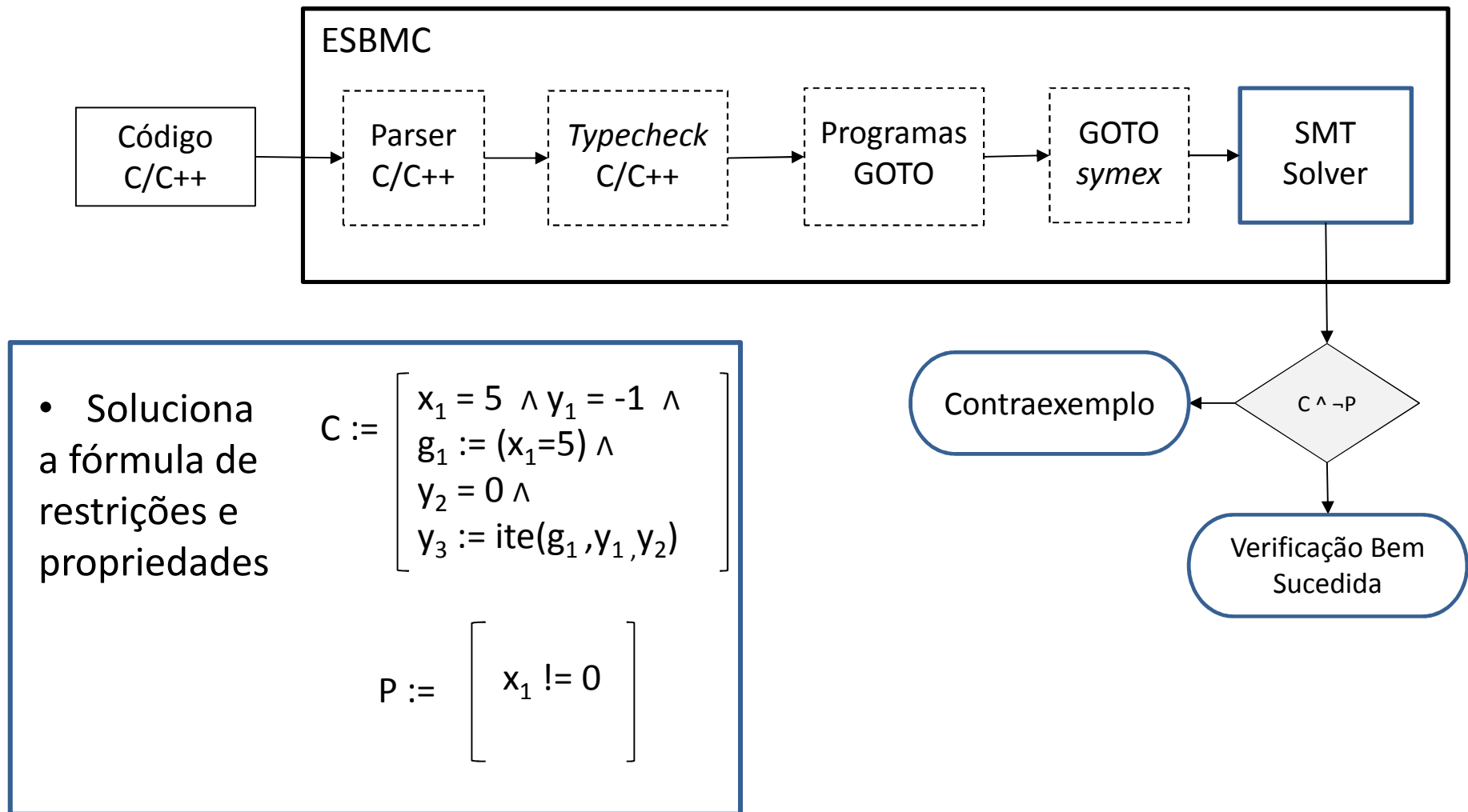
ESBMC - Arquitetura



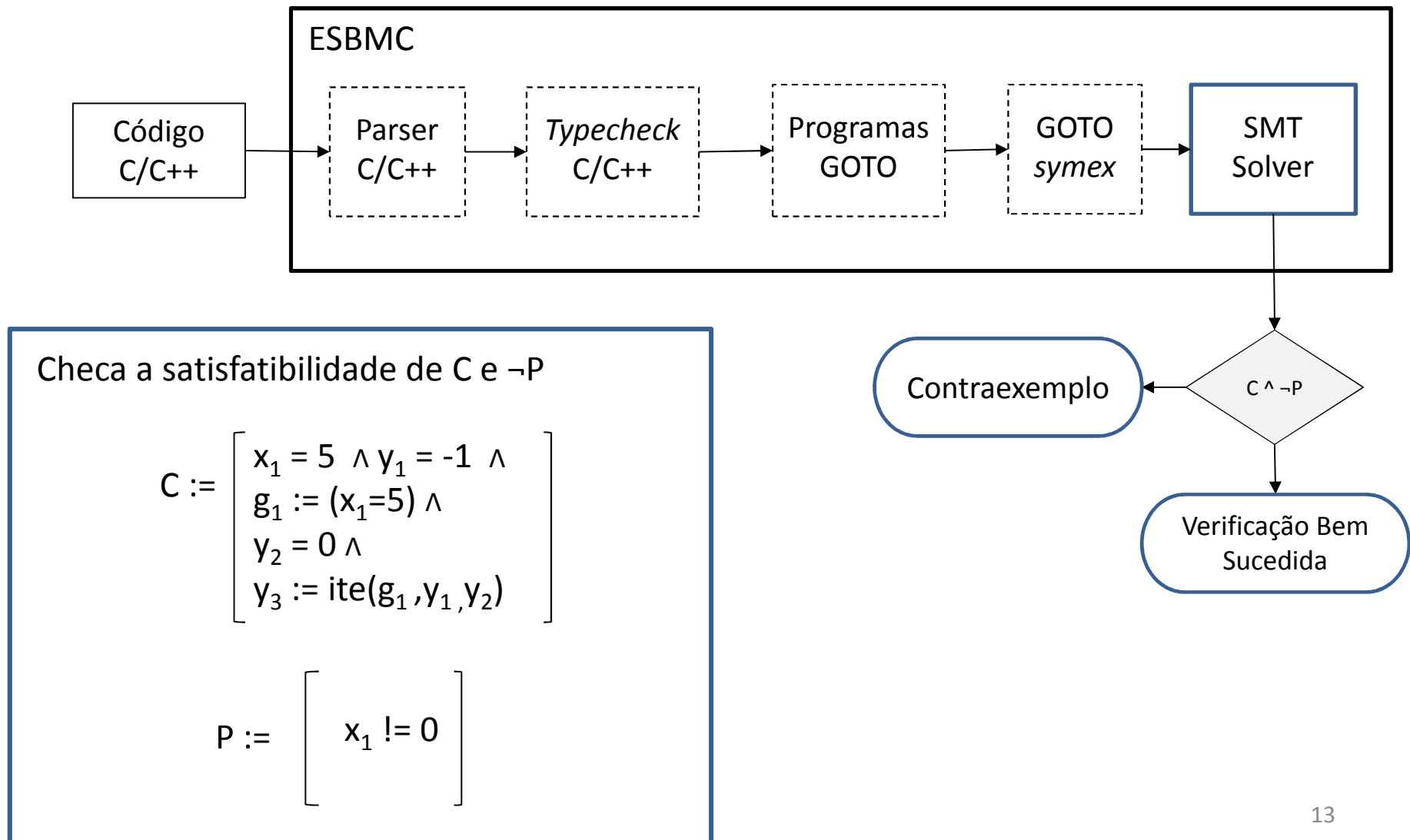
ESBMC - Arquitetura



ESBMC - Arquitetura



ESBMC - Arquitetura



Exploração Preguiçosa

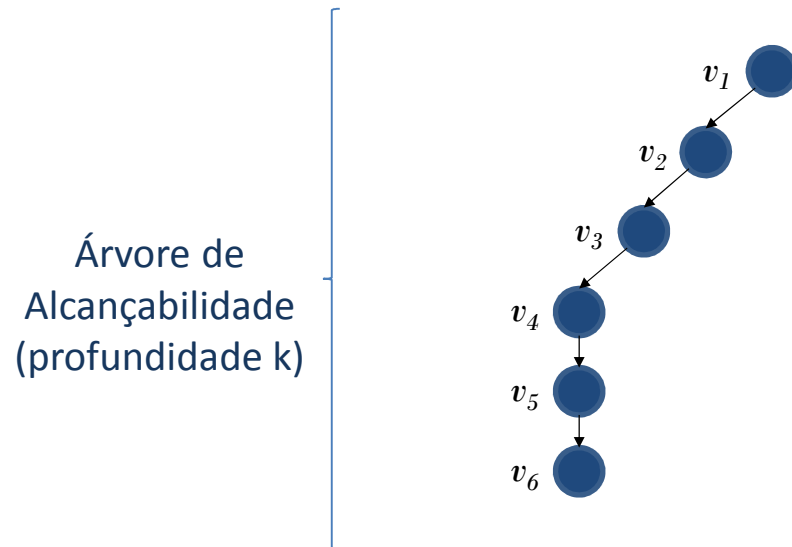
- Técnica de verificação usada pelo ESBMC

Exploração Preguiçosa

- Técnica de verificação usada pelo ESBMC
- Verifica as intercalações de forma incremental

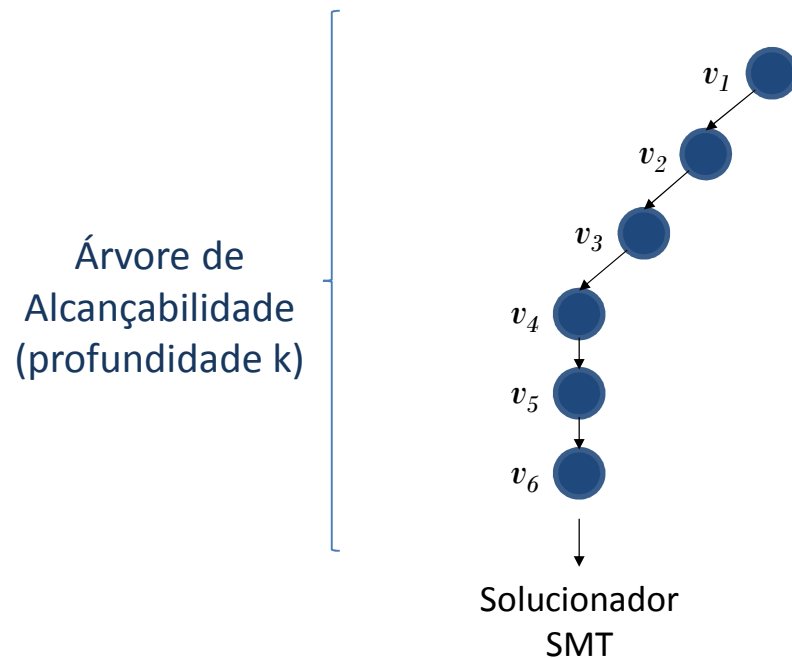
Exploração Preguiçosa

- Técnica de verificação usada pelo ESBMC
- Verifica as intercalações de forma incremental



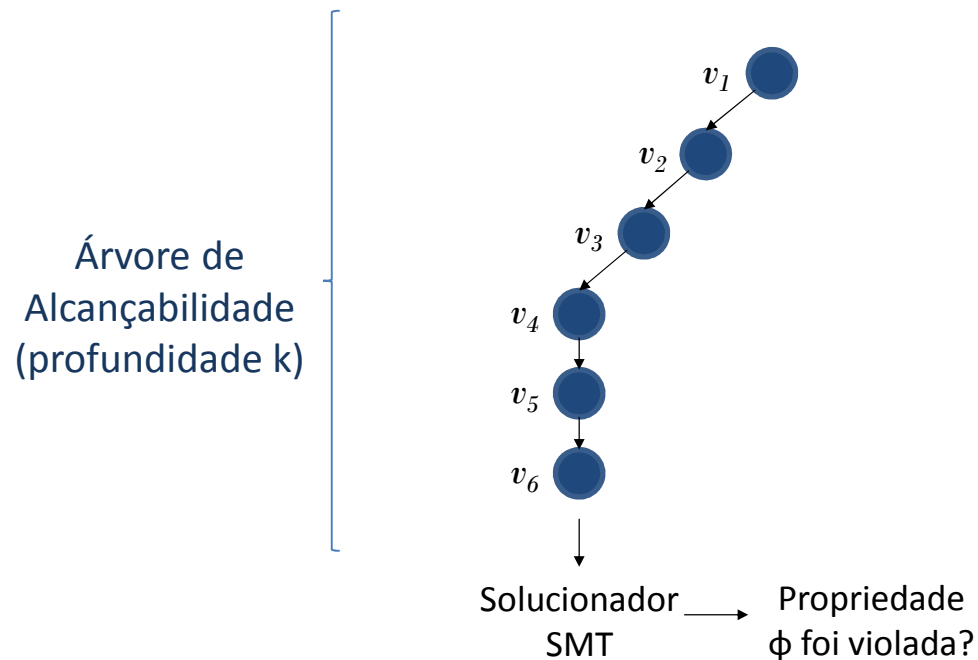
Exploração Preguiçosa

- Técnica de verificação usada pelo ESBMC
- Verifica as intercalações de forma incremental



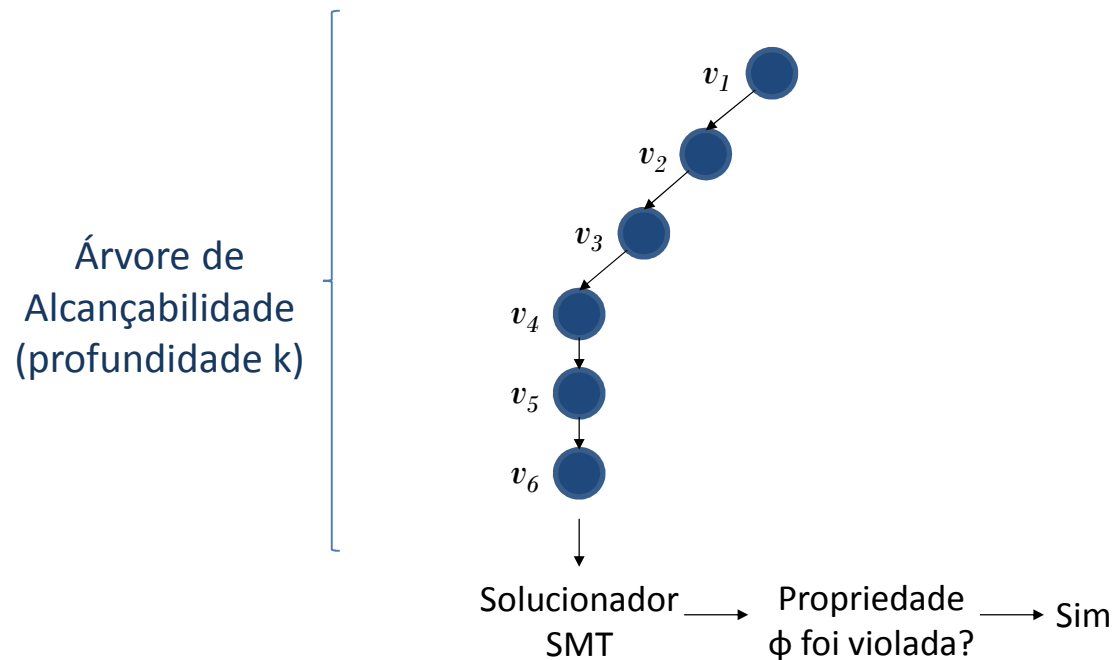
Exploração Preguiçosa

- Técnica de verificação usada pelo ESBMC
- Verifica as intercalações de forma incremental



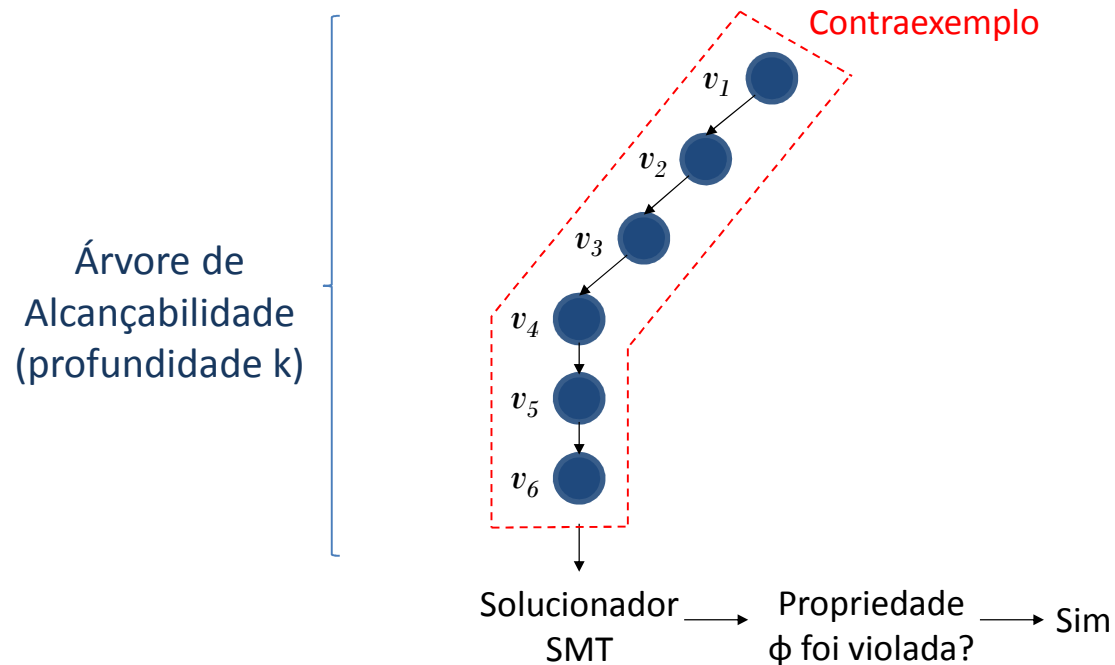
Exploração Preguiçosa

- Técnica de verificação usada pelo ESBMC
- Verifica as intercalações de forma incremental



Exploração Preguiçosa

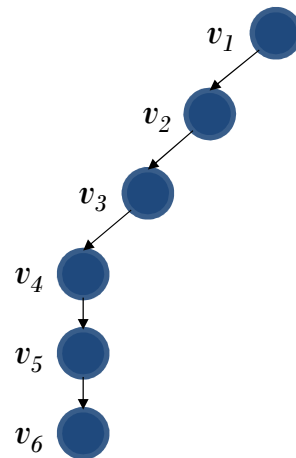
- Técnica de verificação usada pelo ESBMC
- Verifica as intercalações de forma incremental



Exploração Preguiçosa

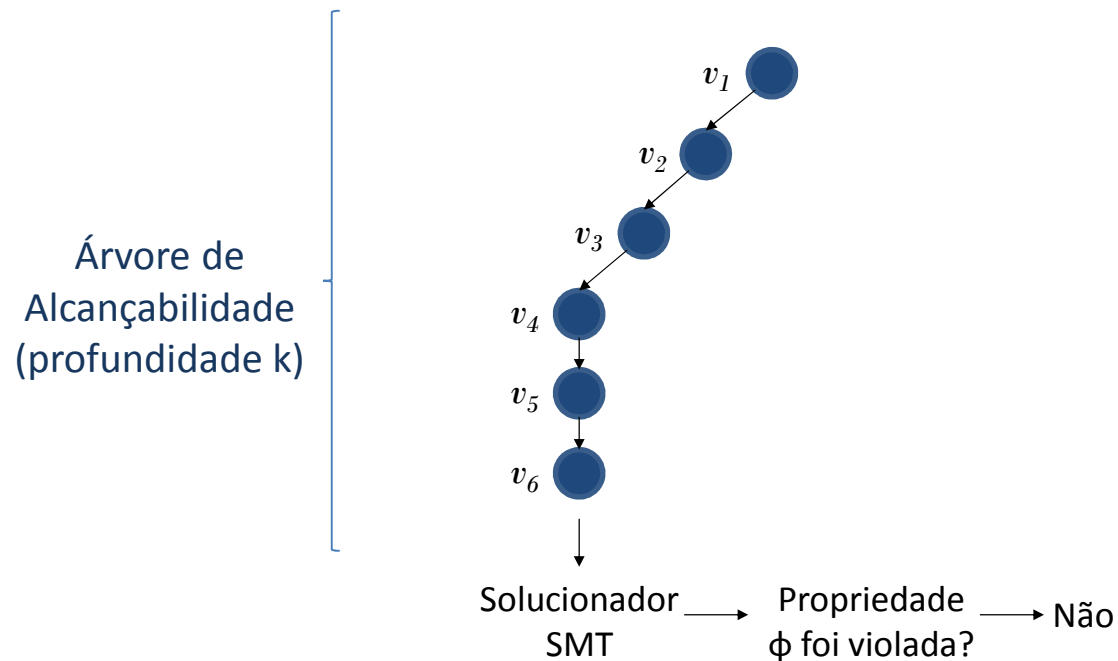
- Técnica de verificação usada pelo ESBMC
- Verifica as intercalações de forma incremental

Árvore de
Alcançabilidade
(profundidade k)



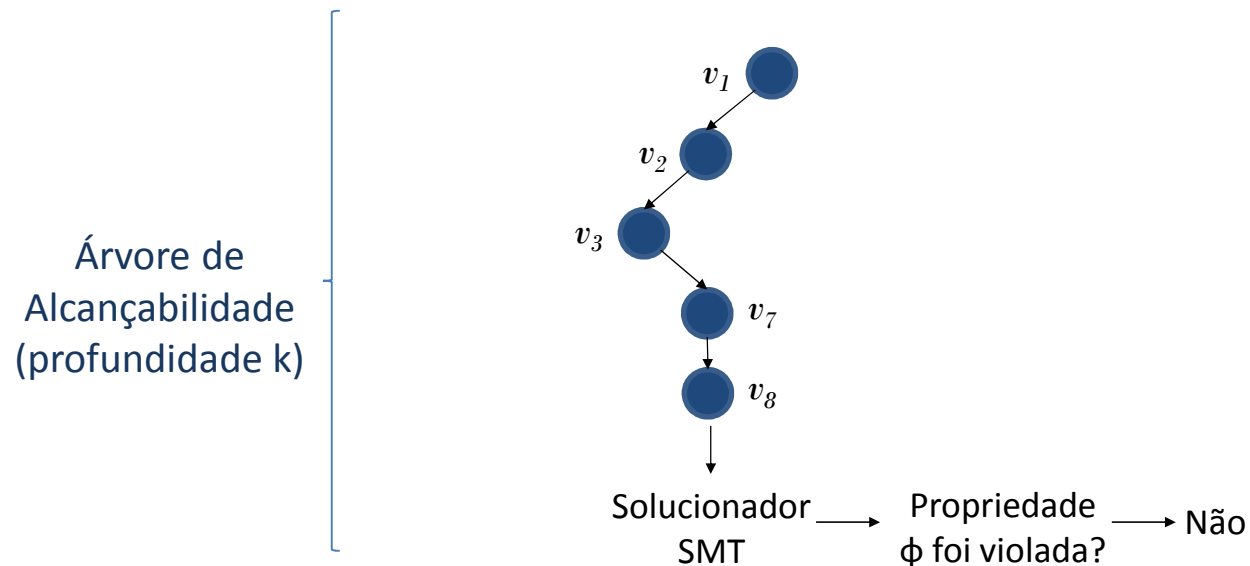
Exploração Preguiçosa

- Técnica de verificação usada pelo ESBMC
- Verifica as intercalações de forma incremental



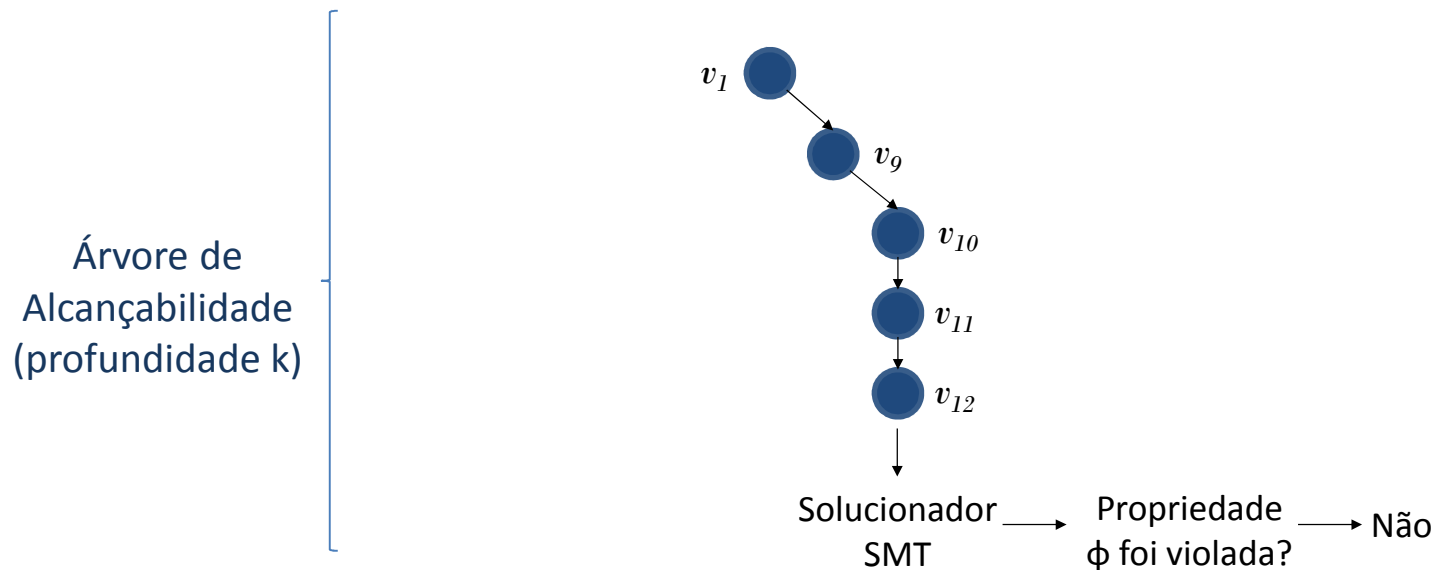
Exploração Preguiçosa

- Técnica de verificação usada pelo ESBMC
- Verifica as intercalações de forma incremental



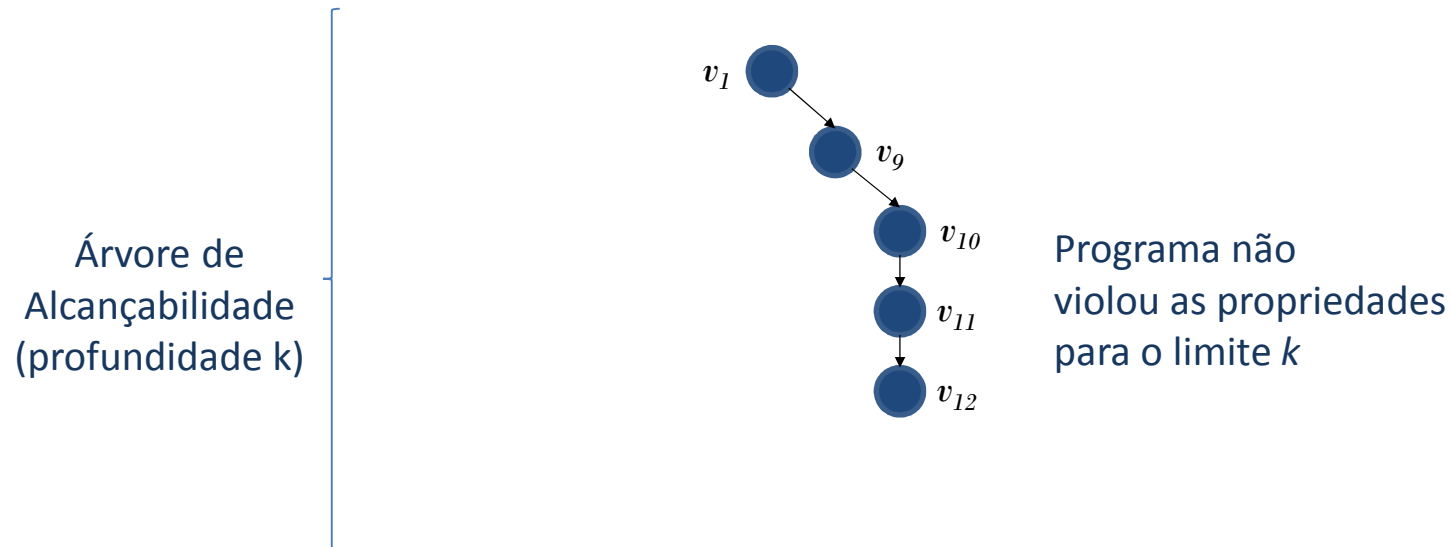
Exploração Preguiçosa

- Técnica de verificação usada pelo ESBMC
- Verifica as intercalações de forma incremental



Exploração Preguiçosa

- Técnica de verificação usada pelo ESBMC
- Verifica as intercalações de forma incremental



Redução de Ordem Parcial

- Algoritmo que classifica transições de *threads* em um programa multitarefa
 - Podem ser dependentes ou independentes
 - Identifica pares de intercalações que resultam em um mesmo estado
- Primeira aplicação da técnica para verificar programas CUDA
 - Reduzir o tempo e a complexidade da verificação
 - Eliminar intercalações de *threads* que acessam posições diferentes de um vetor

Redução de Ordem Parcial

```
kernel1(int *a){  
    a[threadIdx.x] = threadIdx.x ;  
}
```


$v_0: t_0, 0, a[0] = 0, a[1] = 0$

Redução de Ordem Parcial

```
kernel1(int *a){  
    a[threadIdx.x] = threadIdx.x;  
}
```

$v_0: t_0, 0, a[0] = 0, a[1] = 0$

$v_1: t_1, 1, a[0] = 0, a[1] = 0$



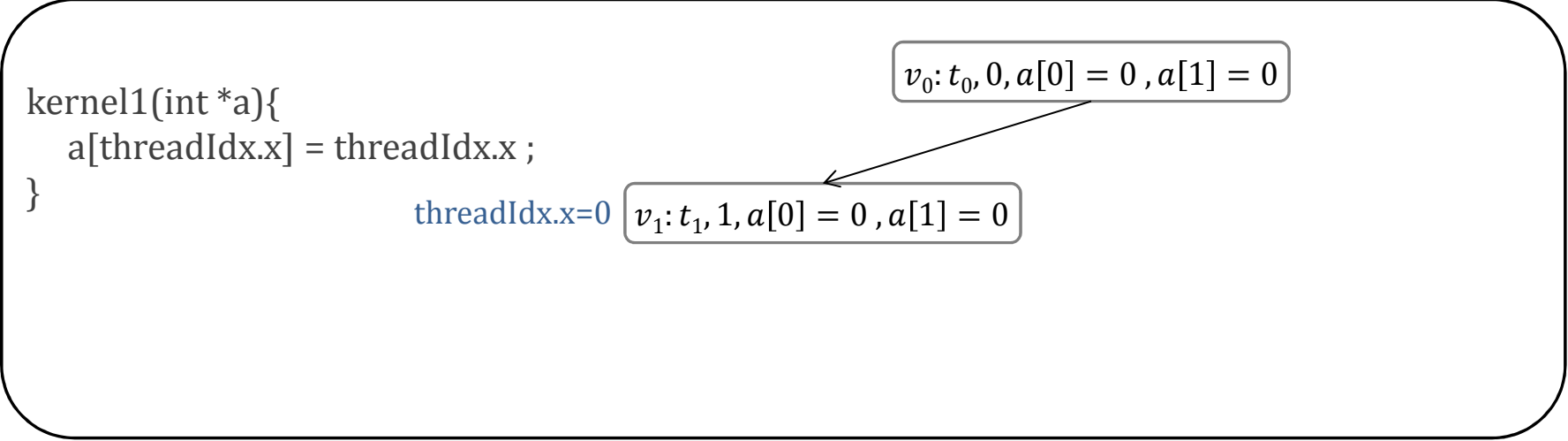
Redução de Ordem Parcial

```
kernel1(int *a){  
  a[threadIdx.x] = threadIdx.x;  
}
```

`threadIdx.x=0`

$v_1: t_1, 1, a[0] = 0, a[1] = 0$

$v_0: t_0, 0, a[0] = 0, a[1] = 0$



Redução de Ordem Parcial

```
kernel1(int *a){  
  a[threadIdx.x] = threadIdx.x;  
}
```

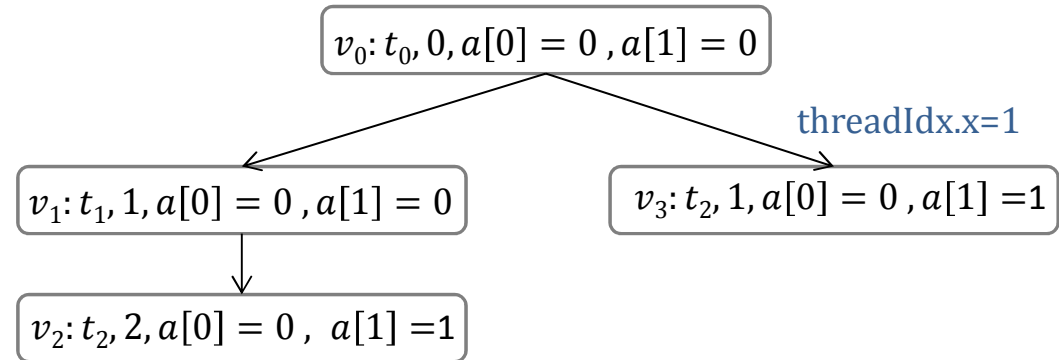
$v_0: t_0, 0, a[0] = 0, a[1] = 0$

$v_1: t_1, 1, a[0] = 0, a[1] = 0$

$\text{threadIdx.x}=1$ $v_2: t_2, 2, a[0] = 0, a[1] = 1$

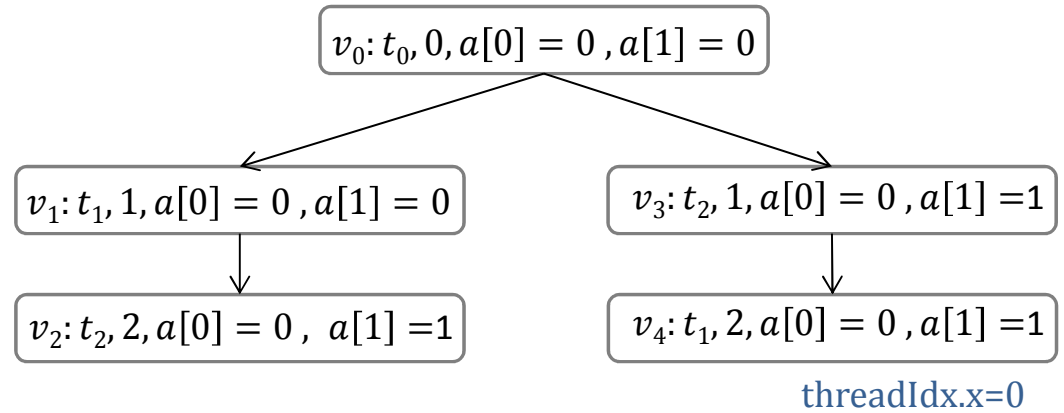
Redução de Ordem Parcial

```
kernel1(int *a){  
  a[threadIdx.x] = threadIdx.x ;  
}
```



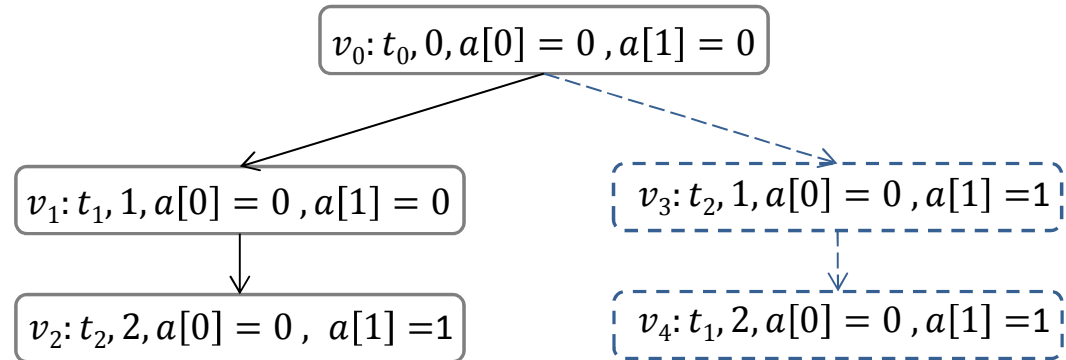
Redução de Ordem Parcial

```
kernel1(int *a){  
  a[threadIdx.x] = threadIdx.x ;  
}
```



Redução de Ordem Parcial

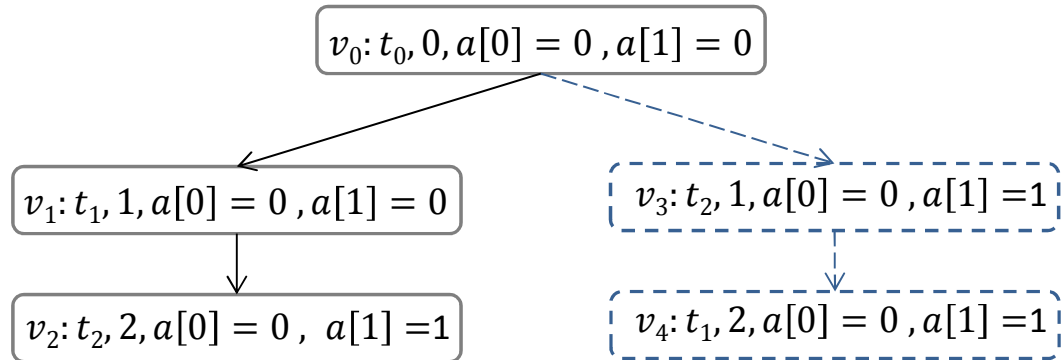
```
kernel1(int *a){  
  a[threadIdx.x] = threadIdx.x ;  
}
```



Transições independentes

Redução de Ordem Parcial

```
kernel1(int *a){  
  a[threadIdx.x] = threadIdx.x ;  
}
```



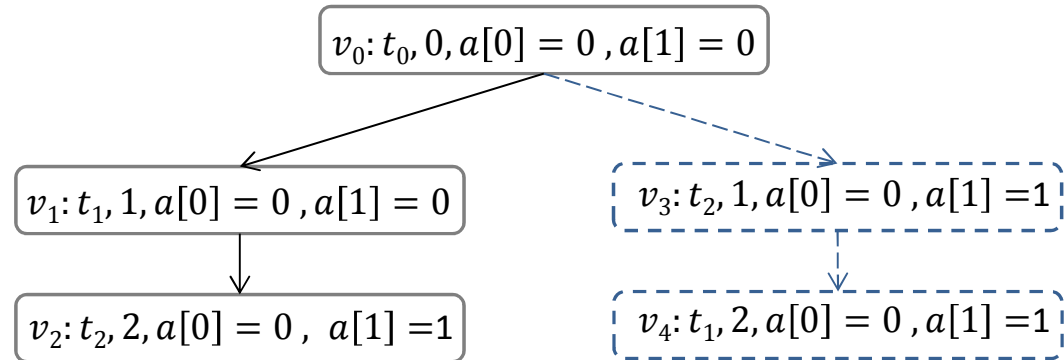
Transições independentes

```
kernel2(int *a){  
  if(a[1]==1)  
    a[threadIdx.x+2] = threadIdx.x ;  
  else  
    a[threadIdx.x] = threadIdx.x;  
}
```

$v_0: t_0, 0, a[0] = 0, a[1] = 0$

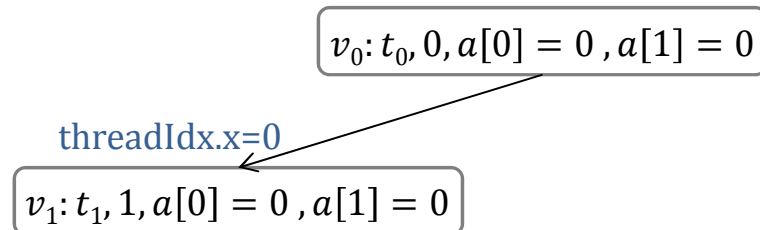
Redução de Ordem Parcial

```
kernel1(int *a){  
  a[threadIdx.x] = threadIdx.x;  
}
```



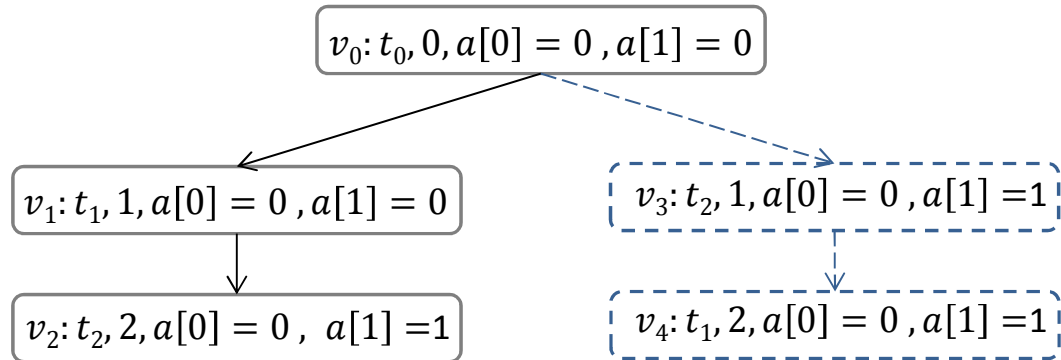
Transições independentes

```
kernel2(int *a){  
  if(a[1]==1)  
    a[threadIdx.x+2] = threadIdx.x;  
  else  
    a[threadIdx.x] = threadIdx.x;  
}
```



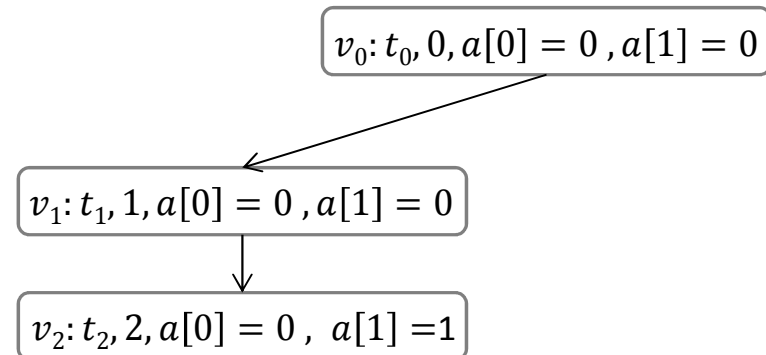
Redução de Ordem Parcial

```
kernel1(int *a){  
  a[threadIdx.x] = threadIdx.x ;  
}
```



Transições independentes

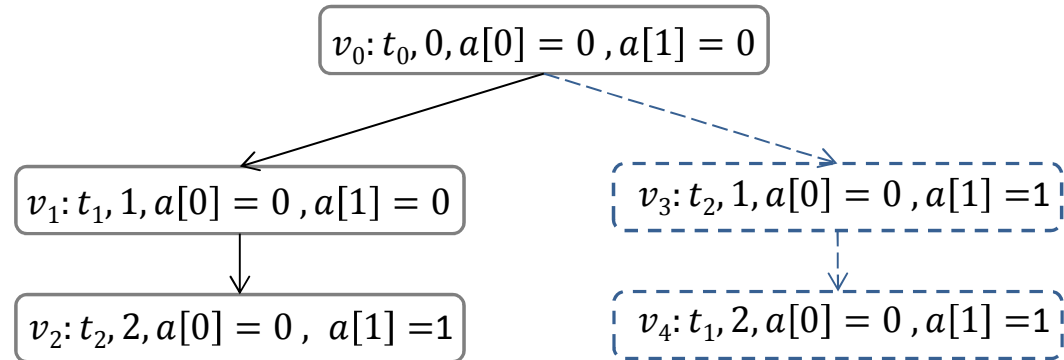
```
kernel2(int *a){  
  if(a[1]==1)  
    a[threadIdx.x+2] = threadIdx.x ;  
  else  
    a[threadIdx.x] = threadIdx.x;  
}
```



threadIdx.x=1

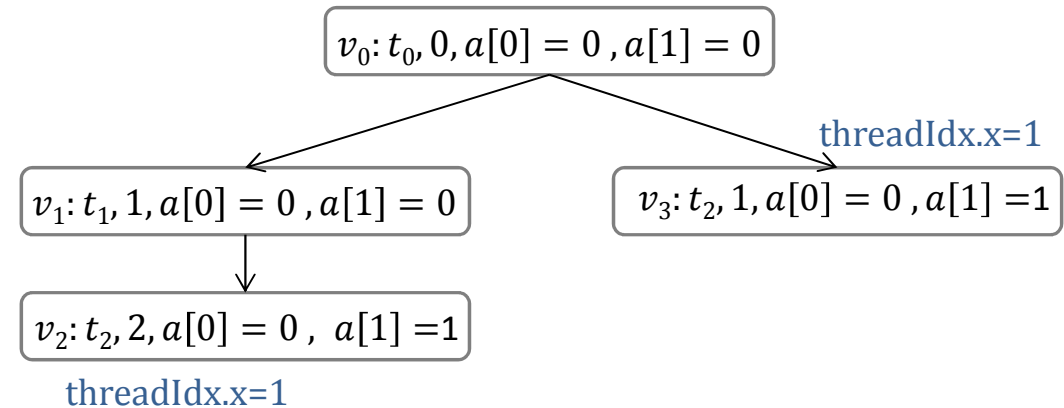
Redução de Ordem Parcial

```
kernel1(int *a){
  a[threadIdx.x] = threadIdx.x;
}
```



Transições independentes

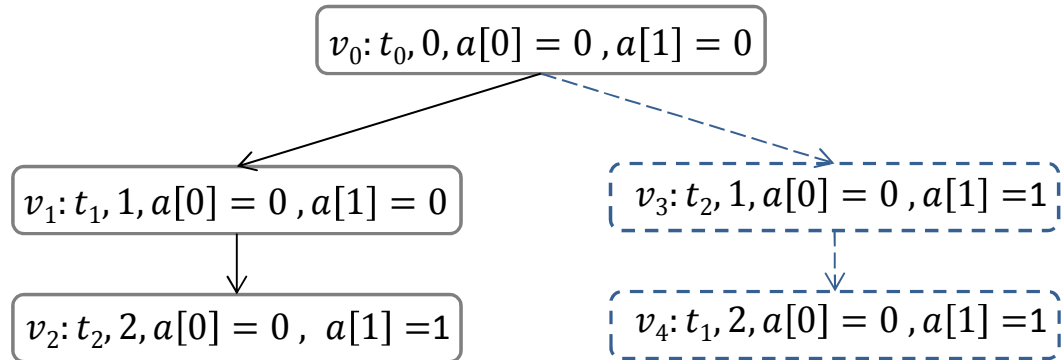
```
kernel2(int *a){
  if(a[1]==1)
    a[threadIdx.x+2] = threadIdx.x;
  else
    a[threadIdx.x] = threadIdx.x;
}
```



threadIdx.x=1

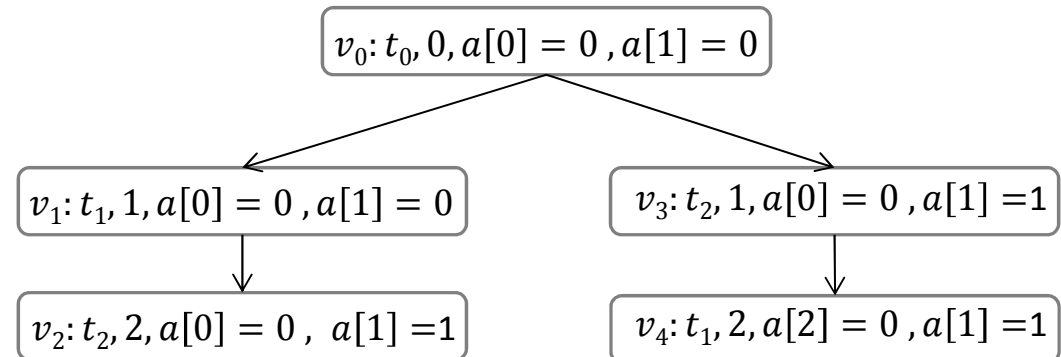
Redução de Ordem Parcial

```
kernel1(int *a){  
  a[threadIdx.x] = threadIdx.x;  
}
```



Transições independentes

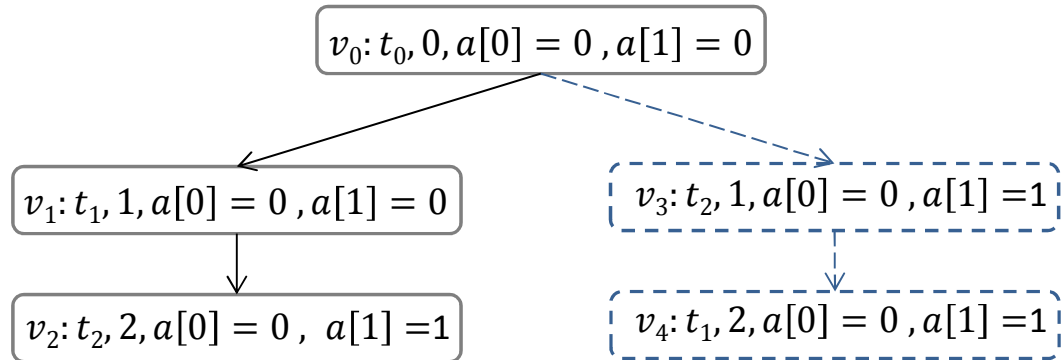
```
kernel2(int *a){  
  if(a[1]==1)  
    a[threadIdx.x+2] = threadIdx.x;  
  else  
    a[threadIdx.x] = threadIdx.x;  
}
```



$threadIdx.x=0$

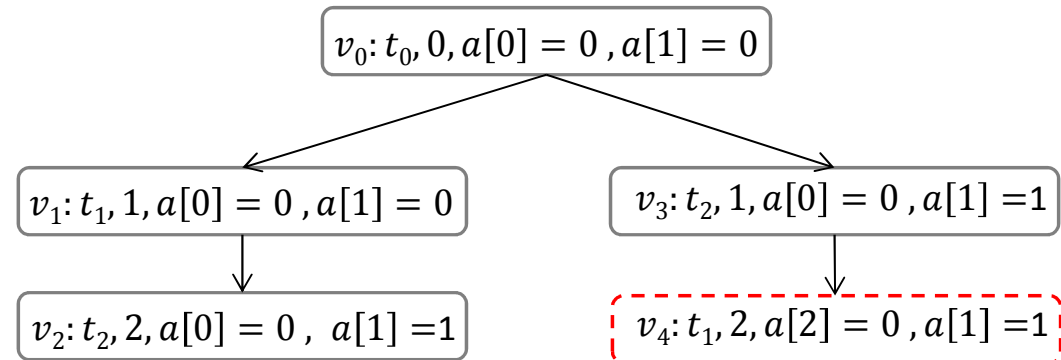
Redução de Ordem Parcial

```
kernel1(int *a){  
  a[threadIdx.x] = threadIdx.x;  
}
```



Transições independentes

```
kernel2(int *a){  
  if(a[1]==1)  
    a[threadIdx.x+2] = threadIdx.x;  
  else  
    a[threadIdx.x] = threadIdx.x;  
}
```

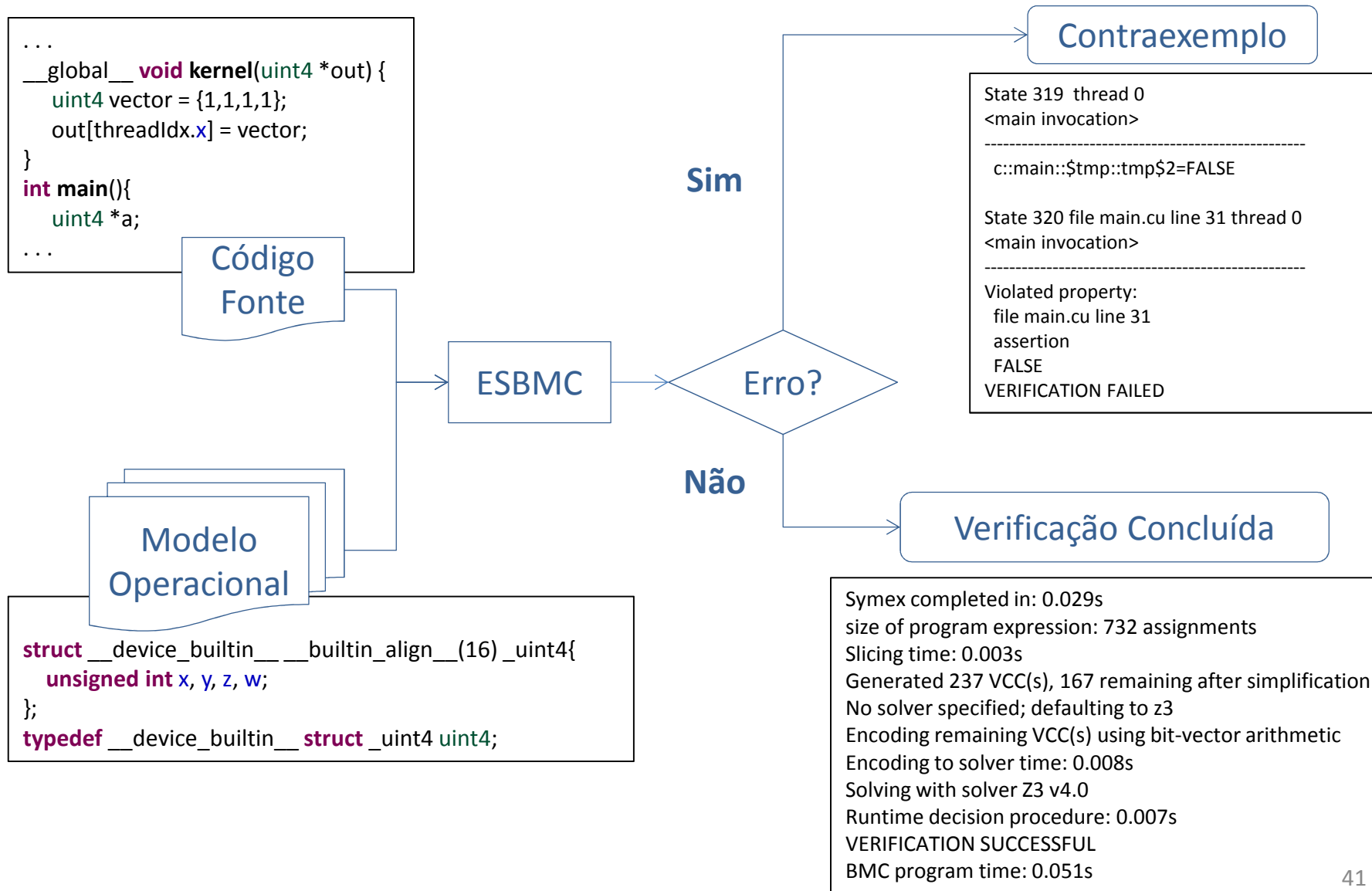


Transições dependentes

Modelo Operacional CUDA (MOC)

- O que são Modelos Operacionais?
 - Representação das estruturas da linguagem
 - Permite a checagem de pré e pós-condições
- Redução do custo computacional
 - Somente códigos relevantes
- Permite checagem de propriedades
- ESBMC opera com Modelos Operacionais
 - ESBMC++, QTOM, Verifying C++ with STL Containers
- CUDA, plataforma fechada
 - Guia de programação CUDA e a IDE Nsight

Modelo Operacional CUDA (MOC)



Implementação do MOC

```
#include <cuda.h>
#include <stdio.h>
#define N 2
__global__ void definitions(int* A){
    atomicAdd(A,10);
}
int main (){
    int a = 5;
    int *dev_a;
    cudaMalloc ((void** ) &dev_a, sizeof(int));
    cudaMemcpy(dev_a, &a,
sizeof(int),cudaMemcpyHostToDevice);
    ESBMC_verify_kernel(definitions,1,N,dev_a);
    cudaMemcpy(&a,dev_a,sizeof(int),cudaMemcpyDeviceToHost);
    assert(a==25);
    cudaFree(dev_a);
    return 0;
}
```

Implementação do MOC

```
#include <cuda.h>
#include <stdio.h>
#define N 2
__global__ void definitions(int* A){
    atomicAdd(A,10);
}
int main (){
    int a = 5;
    int *dev_a;
    cudaMalloc ((void** ) &dev_a, sizeof(int));
    cudaMemcpy(dev_a, &a,
sizeof(int),cudaMemcpyHostToDevice);
    ESBMC_verify_kernel(definitions,1,N,dev_a);
    cudaMemcpy(&a,dev_a,sizeof(int),cudaMemcpyDeviceToHost);
    assert(a==25);
    cudaFree(dev_a);
    return 0;
}
```

Implementação do MOC

```
# cudaMalloc
```

```
cudaError_t cudaMalloc(void ** devPtr, size_t size) {  
    cudaError_t tmp;  
    __ESBMC_assert(size > 0, "Size to be allocated may not be less than zero");  
    *devPtr = malloc(size);  
    if (*devPtr == NULL) {  
        tmp = CUDA_ERROR_OUT_OF_MEMORY;  
        exit(1);  
    } else {  
        tmp = CUDA_SUCCESS;  
    }  
    __ESBMC_assert(tmp == CUDA_SUCCESS, "Memory was not allocated");  
    lastError = tmp;  
    return lastError;  
}
```

Implementação do MOC

```
# cudaMalloc
cudaError_t cudaMalloc(void ** devPtr, size_t size) {
    cudaError_t tmp;
    ESBMC_assert(size > 0, "Size to be allocated may not be less than zero");
    *devPtr = malloc(size);
    if (*devPtr == NULL) {
        tmp = CUDA_ERROR_OUT_OF_MEMORY;
        exit(1);
    } else {
        tmp = CUDA_SUCCESS;
    }
    __ESBMC_assert(tmp == CUDA_SUCCESS, "Memory was not allocated");
    lastError = tmp;
    return lastError;
}
```

Implementação do MOC

```
# cudaMalloc
cudaError_t cudaMalloc(void ** devPtr, size_t size) {
    cudaError_t tmp;
    __ESBMC_assert(size > 0, "Size to be allocated may not be less than zero");
    *devPtr = malloc(size);
    if (*devPtr == NULL) {
        tmp = CUDA_ERROR_OUT_OF_MEMORY;
        exit(1);
    } else {
        tmp = CUDA_SUCCESS;
    }
    __ESBMC_assert(tmp == CUDA_SUCCESS, "Memory was not allocated");
    lastError = tmp;
    return lastError;
}
```

Implementação do MOC

```
#include <cuda.h>
#include <stdio.h>
#define N 2
__global__ void definitions(int* A){
    atomicAdd(A,10);
}
int main (){
    int a = 5;
    int *dev_a;
    cudaMalloc ((void**) &dev_a, sizeof(int));
    cudaMemcpy(dev_a, &a,
sizeof(int),cudaMemcpyHostToDevice);
    ESBMC_verify_kernel(definitions,1,N,dev_a);
    cudaMemcpy(&a,dev_a,sizeof(int),cudaMemcpyD
eviceToHost);
    assert(a==25);
    cudaFree(dev_a);
    return 0;
}
```

Implementação do MOC

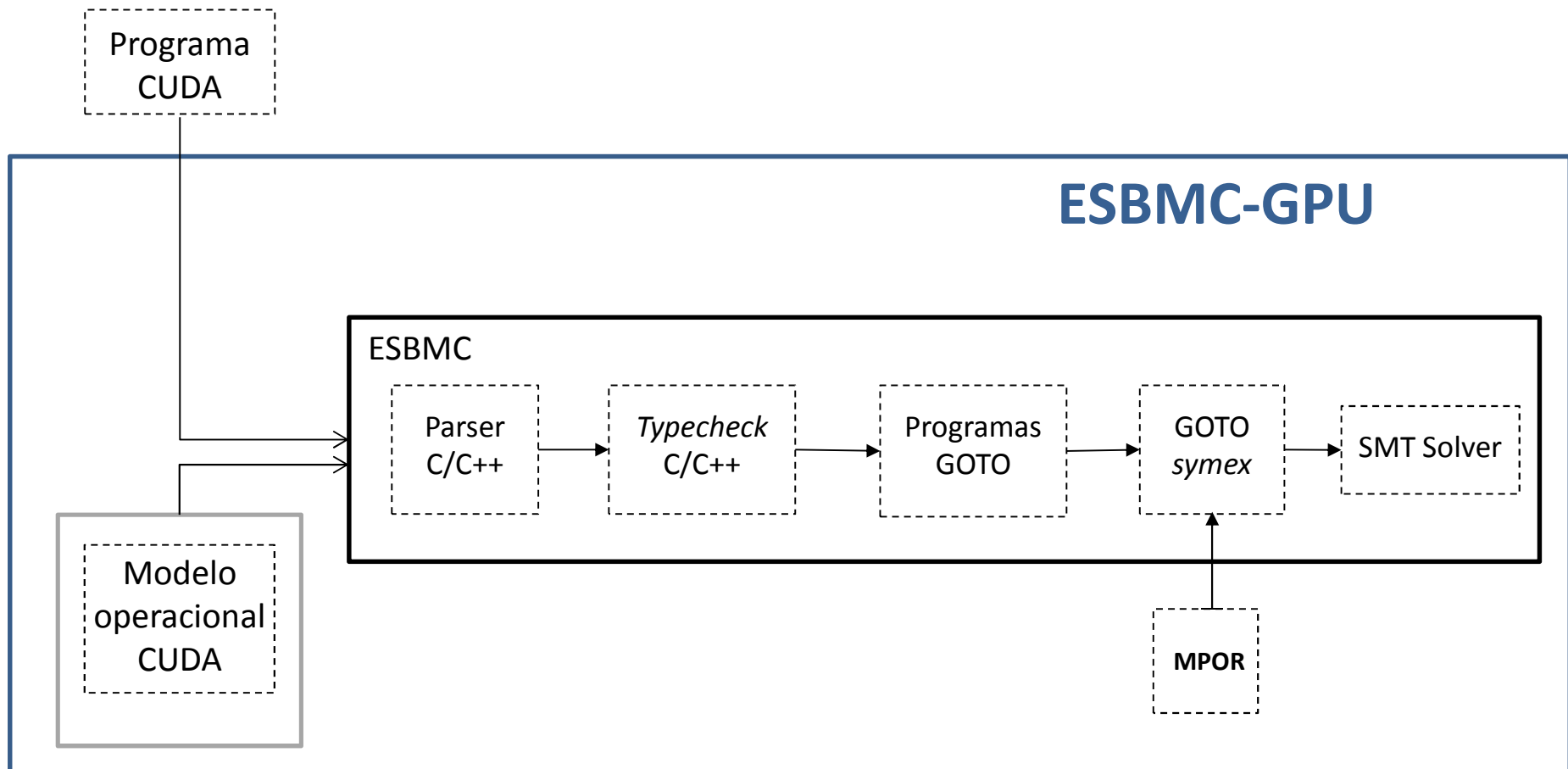
```
# atomicAdd
__device__ int atomicAdd(int *address, int val){

    __ESBMC_atomic_begin();
    int old_value, new_value;

    old_value = *address;
    new_value = old_value + val;
    *address = new_value;

    return old_value;
    __ESBMC_atomic_end();
}
```


Arquitetura do ESBMC-GPU



Avaliação Experimental

- Objetivo: assegurar a consistência dos modelos operacionais implementados
 - Os resultados da verificação são coerentes com as especificações de CUDA?
 - A ferramenta é capaz de detectar erros em programas reais?
- Questões de pesquisa
 - **Q1**: qual o resultado obtido pelo ESBMC-GPU sobre os *benchmarks*?
 - **Q2**: qual o resultado do ESBMC-GPU quando comparado com as ferramentas GPUVerify e PUG?

Descrição dos *Benchmarks*

- Composta por 160 *benchmarks* provenientes da literatura
 - Operações aritméticas
 - Chamada de funções *device*
 - Funções específicas de:
 - C/C++ (*e.g.*, *memset*, *assert*)
 - CUDA (*e.g.*, *atomicAdd*, *cudaMemcpy*, *cudaMalloc*, *cudaFree*, *syncthreads*)
 - Bibliotecas de CUDA (*e.g.*, *curand.h*)
 - Variáveis *int*, *float*, *char* e seus modificadores (*long* e *unsigned*)
 - Ponteiros para variáveis e funções
 - *Typedefs*
 - Variáveis intrínsecas de CUDA (*e.g.*, *uint4*)

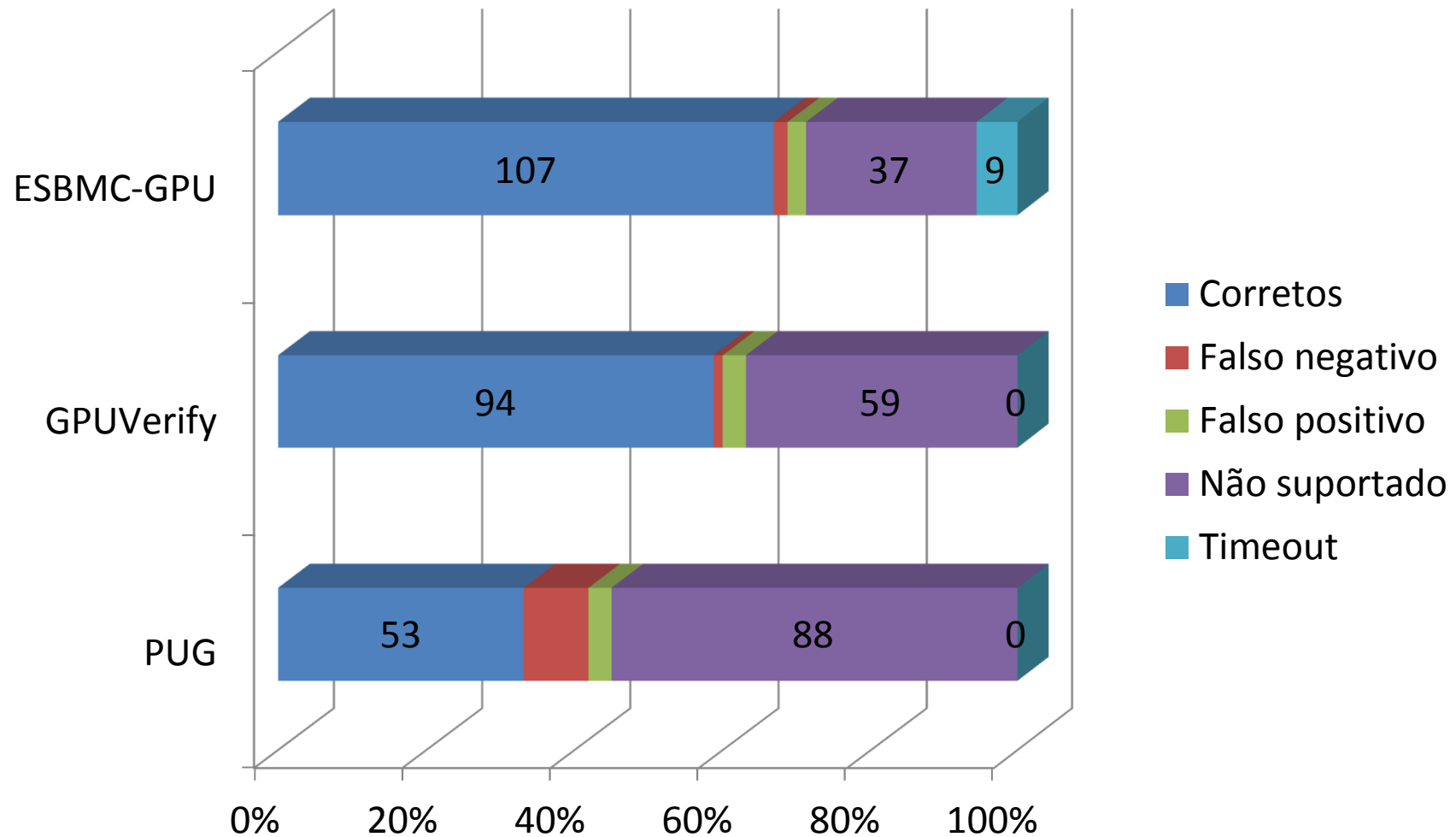
Ferramentas Relacionadas

- GPUVerify usa uma semântica para verificação de corrida de dados e divergência de barreira
 - Sem suporte à função *main*
 - Não considera a execução do programa
- PUG analisa *kernels* usando solucionadores SMT
 - Detecta corrida de dados, sincronização de barreira e conflitos de banco em memória compartilhada

Resultados Experimentais

Resultado \ Ferramenta	ESBMC-GPU	GPUVERIFY	PUG
Correto	107	94	53
Falsos negativos	3	2	14
Falsos positivos	4	5	5
Não suportado	37	59	88
<i>Timeout</i>	9	0	0
Tempo(s)	15.912	160	10

Comparação com outras ferramentas



Análise dos Resultados – ESBMC-GPU

- Falsos negativos
 - Assertivas e variável *float* na função *cudaMalloc*
- Falsos positivos
 - Assertiva, condição de corrida em variável *shared* e ponteiro nulo
- Não suportados
 - Funções (*e.g.*, *mul24*) e bibliotecas (*curand*) específicas de CUDA
 - Ponteiros para função
 - Tipo de dado *char* e *struct* como argumento em *kernel*
- *Timeouts*
 - Execução concreta das intercalações do programa
 - Considera as possíveis trocas de contexto entre *threads*

Considerações Finais

- Desenvolvimento de uma ferramenta capaz de verificar programas CUDA
 - Primeira ferramenta que usa:
 - BMC e SMT para verificar programas CUDA
 - MPOR, responsável por reduzir em 80% o tempo de verificação dos *benchmarks*
- 66,8% de resultados corretos em comparação aos 58,7% da GPUVerify e 33,1% da PUG
 - Ferramentas consideradas estado da arte
- Trabalhos futuros:
 - Detecção de divergência de barreira
 - Suporte a novos tipos de memória (*e.g.*, pinada e unificada)
 - Técnicas para redução de intercalações e do tempo de verificação