Model Checking

Aula para disciplina de Métodos Formais

Gabriela Moreira

Departamento de Ciência da Computação - DCC
Universidade do Estado de Santa Catarina - UDESC

30 de junho de 2025

Conteúdo

Outline

TLC (LAMPORT, 2002)

Explicit-State model checker

TLC (LAMPORT, 2002)

Explicit-State model checker

- Enumera todos os estados

TLC (LAMPORT, 2002)

Explicit-State model checker

- Enumera todos os estados
- Usa um grafo direcionado com estados e transições (*reachability graph*) e uma fila de estados para checar

TLC (LAMPORT, 2002)

Explicit-State model checker

- Enumera todos os estados
- Usa um grafo direcionado com estados e transições (*reachability graph*) e uma fila de estados para checar
- Provavelmente, o tipo de *model checker* que um de nós escreveria se fôssemos tentar (sem pesquisar um monte antes)

Apalache (KONNOV; KUKOVEC; TRAN, 2019)

Model checker simbólico

Apalache (KONNOV; KUKOVEC; TRAN, 2019)

Model checker simbólico

- Traduz a especificação para um conjunto de restrições (sem quantificação)

Apalache (KONNOV; KUKOVEC; TRAN, 2019)

Model checker simbólico

- Traduz a especificação para um conjunto de restrições (sem quantificação)
- Usa um SMT *solver* para verificar a satisfabilidade das restrições

Apalache (KONNOV; KUKOVEC; TRAN, 2019)

Model checker simbólico

- Traduz a especificação para um conjunto de restrições (sem quantificação)
- Usa um SMT *solver* para verificar a satisfabilidade das restrições

Bounded model checking

Apalache (KONNOV; KUKOVEC; TRAN, 2019)

Model checker simbólico

- Traduz a especificação para um conjunto de restrições (sem quantificação)
- Usa um SMT *solver* para verificar a satisfabilidade das restrições

Bounded model checking

- Necessário definir um número máximo de passos (*bound*) para o qual gerar restrições. A verificação se dá dentro desse limite, podendo haver estados atingíveis fora desse limite que não satisfazem a propriedade.

SMT

- Generalização de problemas SAT

SMT

- Generalização de problemas SAT
  - SAT: Problema da satisfabilidade Booleana

SMT

- Generalização de problemas SAT
  - SAT: Problema da satisfabilidade Booleana
  - SMT: Satisfabilidade "*modulo*" teorias

SMT

- Generalização de problemas SAT
  - SAT: Problema da satisfabilidade Booleana
  - SMT: Satisfabilidade "*modulo*" teorias
    - "*modulo*" no sentido de "dentro de"

SMT

- Generalização de problemas SAT
  - SAT: Problema da satisfabilidade Booleana
  - SMT: Satisfabilidade "*modulo*" teorias
    - "*modulo*" no sentido de "dentro de"
    - Permite mais elementos: inteiros, reais, arrays, etc

SMT

- Generalização de problemas SAT
  - SAT: Problema da satisfabilidade Booleana
  - SMT: Satisfabilidade "*modulo*" teorias
    - "*modulo*" no sentido de "dentro de"
    - Permite mais elementos: inteiros, reais, arrays, etc

SMT

- Generalização de problemas SAT
  - SAT: Problema da satisfabilidade Booleana
  - SMT: Satisfabilidade "*modulo*" teorias
    - "*modulo*" no sentido de "dentro de"
    - Permite mais elementos: inteiros, reais, arrays, etc

Z3 (DE MOURA; BJØRNER, 2008):

- Leonardo de Moura e Nikolaj Bjørner
  - Podcast: Type Theory Forall #33 Z3 and Lean, the Spiritual Journey

SMT

- Generalização de problemas SAT
  - SAT: Problema da satisfabilidade Booleana
  - SMT: Satisfabilidade "*modulo*" teorias
    - "*modulo*" no sentido de "dentro de"
    - Permite mais elementos: inteiros, reais, arrays, etc

Z3 (DE MOURA; BJØRNER, 2008):

- Leonardo de Moura e Nikolaj Bjørner
  - Podcast: Type Theory Forall #33 Z3 and Lean, the Spiritual Journey
- Versão em WebAssembly
  - `https://people.csail.mit.edu/cpitcla/z3.wasm/z3.html`

Resolvendo SMT com Z3

```
(declare-fun p () Bool)
(declare-fun q () Bool)
(assert (implies p q))
(assert p)
(assert (not q))
(check-sat)
```

Resultado: unsat

Resolvendo SMT com Z3

```
(declare-fun p () Bool)
(declare-fun q () Bool)
(assert (implies p q))
(assert p)
(assert (not q))
(check-sat)
```

Resultado: unsat

```
(declare-fun p () Bool)
(declare-fun q () Bool)
(assert (implies p q))
(assert p)
(assert q)
(check-sat)
```

Resultado: sat

Resolvendo SMT com inteiros

```
1 (declare-const a Int)
2 (declare-const b Int)
3 (assert (= (+ a b) 20))
4 (assert (= (+ a (* 2 b)) 10))
5 (check-sat)
6 (get-model)
```

Resultado: `sat`

Simulação simbólica

Podemos classificar não-determinismo em dois tipos:

Simulação simbólica

Podemos classificar não-determinismo em dois tipos:

- Não-determinismo de dados (*data non-determinism*)
  - `1.to(10).oneOf()`

Simulação simbólica

Podemos classificar não-determinismo em dois tipos:

- Não-determinismo de dados (*data non-determinism*)
  - `1.to(10).oneOf()`
- Não-determinismo de controle (*control-flow non-determinism*)
  - `any { A, B }`

Simulação simbólica

Podemos classificar não-determinismo em dois tipos:

- Não-determinismo de dados (*data non-determinism*)
  - `1.to(10).oneOf()`
- Não-determinismo de controle (*control-flow non-determinism*)
  - `any { A, B }`

Simulação simbólica:

Simulação simbólica

Podemos classificar não-determinismo em dois tipos:

- Não-determinismo de dados (*data non-determinism*)
  - `1.to(10).oneOf()`
- Não-determinismo de controle (*control-flow non-determinism*)
  - `any { A, B }`

Simulação simbólica:

- *Model Checkers*: exaustivamente checam todos os dados e fluxos possíveis

Simulação simbólica

Podemos classificar não-determinismo em dois tipos:

- Não-determinismo de dados (*data non-determinism*)
  - `1.to(10).oneOf()`
- Não-determinismo de controle (*control-flow non-determinism*)
  - `any { A, B }`

Simulação simbólica:

- *Model Checkers*: exaustivamente checam todos os dados e fluxos possíveis
- Simuladores aleatórios: aleatoriamente checam alguns dados e fluxos

Simulação simbólica

Podemos classificar não-determinismo em dois tipos:

- Não-determinismo de dados (*data non-determinism*)
  - `1.to(10).oneOf()`
- Não-determinismo de controle (*control-flow non-determinism*)
  - `any { A, B }`

Simulação simbólica:

- *Model Checkers*: exaustivamente checam todos os dados e fluxos possíveis
- Simuladores aleatórios: aleatoriamente checam alguns dados e fluxos
- Simulador simbólico: aleatóriamente checa alguns fluxos, mas considerando todos os dados possíveis

Invariância indutivas

Às vezes, podemos provar uma invariante qualquer I (também conhecida como invariante ordinária) mostrando que uma invariante indutiva Inv implica nela ($Inv \Rightarrow I$). Para isso, verificamos três coisas:

1. $Init \Rightarrow Inv$, ou seja, verificar a propriedade Inv com `--max-steps=0` e garantir que Inv é verdade em todos os estados iniciais
2. $Inv \wedge Next \Rightarrow Inv'$, ou seja, verificar a propriedade Inv a partir de um estado inicial que satisfaça Inv, fazendo um passo (`--max-steps=1`)
3. $Inv \Rightarrow I$, ou seja, verificar a propriedade I em todos os estados onde Inv é verdade (`--max-steps=0`).

Invariância indutiva é muito poderosa porque conseguimos uma prova completa mesmo em model-checking limitado.

Referências

DE MOURA, L.; BJØRNER, N. **Z3: an efficient smt solver**Proceedings of the theory and practice of software, 14th international conference on tools and algorithms for the construction and analysis of systems. **Anais**.: Tacas'08/etaps'08.Budapest, Hungary: Springer-Verlag, 2008

KONNOV, I.; KUKOVEC, J.; TRAN, T.-H. Tla+ model checking made symbolic. **Proc. acm program. lang.**, v. 3, n. OOPSLA, Oct. 2019.

LAMPORT, L. **Specifying systems: The tla+ language and tools for hardware and software engineers**. Boston: Addison-Wesley, 2002.

Model Checking

Aula para disciplina de Métodos Formais

Gabriela Moreira

Departamento de Ciência da Computação - DCC
Universidade do Estado de Santa Catarina - UDESC

30 de junho de 2025