Verificação de Modelos (Model Checking)

Estes slides são baseados nas notas de aula da Profa. Corina Cîrstea

Lista de Leitura para a Parte Teórica

- M. Huth and M. Ryan, *Logic in Computer Science – Modelling and Reasoning about Systems* (second edition), Cambridge University Press, 2004.
  (Chapter 3)
- E.M. Clarke, O. Grumberg, and D.A. Peled, *Model Checking*, MIT Press, 1999.
  (Chapters 1-4)
- D.A. Peled, *Software Reliability Methods*, Springer, 2001.
  (Chapter 6)

Verificação de Modelos - Motivação

- requisitos de alta confiabilidade (e.g. Ariane 5)
  $\Rightarrow$ necessidade de assegurar a corretude do projeto/código (nas fases iniciais do projeto)

- abordagens tradicionais para validar o projeto/código:
  - simulação:
    - executado em *abstrações*, ou *modelos*, dos sistemas
    - somente oferece validação parcial, pois somente alguns comportamentos do sistema são explorados
  - teste:
    - efetivo nas fases iniciais de depuração, mas se torna demorado em fases posteriores
    - não é escalável
    - difícil para sistemas concorrentes e distribuídos
    - somente oferece validação parcial, pois nem todos os cenários são explorados

Verificação de Modelos - Motivação

- verificação formal oferece *garantias de corretude*:
  - verificação dedutiva (prova):
    - a descrição do sistema e propriedade de corretude são conjuntos de formulas
    - axiomas e regras de prova são usados para provar a corretude (e.g. propriedades *invariante*)
    - tecnologia de provador de teorema é difícil e não é completamente automático
    - funciona para sistemas de estado infinito
  - verificação de modelos:
    - involve a checagem das propriedades dos *modelos*
    - executa exploração exaustiva de todos os possíveis comportamentos
    - tipicamente funciona em modelos *finitos* somente
    - completamente automático

Lógica Proposicional

- A sintaxe das fórmulas em lógica proposicional é definida pela seguinte regra:

$$\begin{array}{rcl} \mathit{Fml} & ::= & \mathit{Fml} \ \wedge \ \mathit{Fml} \mid \neg \mathit{Fml} \mid (\mathit{Fml}) \mid \mathit{Atom} \\ \mathit{Atom} & ::= & \mathit{Variable} \mid \mathit{true} \mid \mathit{false} \end{array}$$

- Usando os operadores lógicos de conjunção ($\wedge$) e negação ($\neg$), a potência total da lógica proposicional é obtida.
  - $\phi_1 \vee \phi_2 \equiv \neg(\neg\phi_1 \wedge \neg\phi_2)$
  - $\phi_1 \Rightarrow \phi_2 \equiv \neg\phi_1 \vee \phi_2$
  - $\phi_1 \Leftrightarrow \phi_2 \equiv (\phi_1 \Rightarrow \phi_2) \wedge (\phi_2 \Rightarrow \phi_1)$
  - $\phi_1 \oplus \phi_2 \equiv (\phi_1 \wedge \neg\phi_2) \vee (\phi_2 \wedge \neg\phi_1)$
  - $\mathit{ite}(\theta, \phi_1, \phi_2) \equiv (\theta \wedge \phi_1) \vee (\neg\theta \wedge \phi_2)$
- Esta fórmula é satisfeita $(A \Rightarrow B) \wedge A \wedge \neg B$?

Exercício

Estes dois fragmentos de código são equivalentes?

```
if (!a&&!b) h();
else
  if (!a) g();
  else f();
```

```
if (a) f();
else
  if (b) g();
  else h();
```

Verificação de Modelos

- técnica para verificar automaticamente as propriedades de corretude dos sistemas de *estado finito*;
- útil para verificar sistemas de hardware e sistemas de software (concorrente/distribuído)
- requer:
  1. **abordagem de modelagem:** para modelar os sistemas
  2. **linguagem de especificação:** para formalizar as propriedades de corretude
  3. **verificador de modelos:** ferramenta automatizada para checar *exaustivamente* as propriedades do modelo.

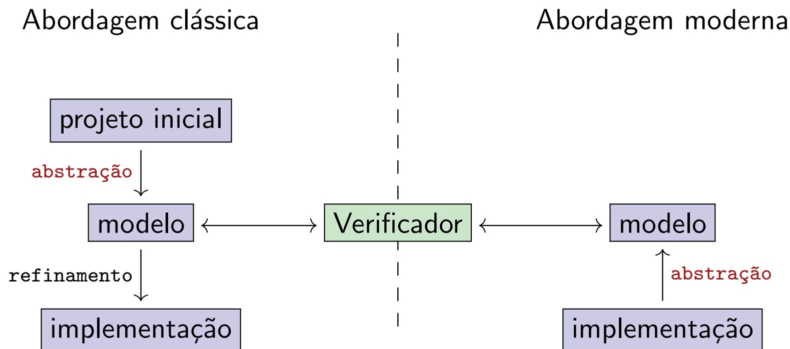
Verificação de Modelos - o Processo

- modelagem: constrói uma *abstração* do sistema sendo analisado, como um modelo matemático que possa ser compreendido por um verificador de modelos (e.g. *Kripke structures*)
  - possivelmente executa simulações para validar o modelo
- especificação: especifica as propriedades desejadas do modelo em um formalismo adequado (e.g. *lógica temporal*)
- verificação com um verificador de modelo:
  - executa exploração exaustiva de todos os possíveis comportamentos, para verificar se as propriedades especificadas fazem parte do modelo
  - um contra-exemplo (trace do erro) é produzido sempre que o modelo não satisfaz a especificação $\Rightarrow$ revisar modelo e/ou especificação

Verificação do Modelo - Vantagens e Desvantagens

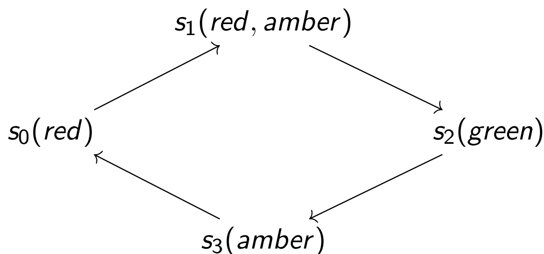
- vantagens:
  - processo completamente automático (diferente da prova de teorema)
  - pode ser usado para executar verificação parcial (somente certas características são modeladas, somente certas propriedades são verificadas)
  - termina com uma resposta sim/não
  - traces do erro são informativos
  - lógicas usadas para especificação podem expressar uma grande classe de propriedades de corretude (geralmente temporal) dos sistemas
- desvantagens:
  - problema da explosão do espaço de estado (quando o sistema sendo verificado tem vários componentes de interação, ou estruturas de dados com vários possíveis valores)
  - tradicionalmente somente funciona para sistemas de estado finito

Verificação de Modelos de Hardware/Software



Estruturas Kripke, Informalmente

- modelos matemáticos para descrever o comportamento dos sistemas
- grafos com vértices representando os estados e as arestas modelando mudanças de estado atômica
- e.g. estados descritos por cores de um semáforo, modelo de transição chaveando de uma cor para outra:



Estruturas Kripke, Formalmente

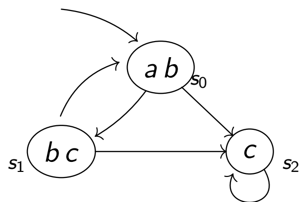
Uma **estrutura Kripke** M sobre um conjunto $Prop$ de *proposições atômicas* é dado por:

- um conjunto *finito* S de estados
- um subconjunto $S_0 \subseteq S$ de estados iniciais
- uma relação de transição $R \subseteq S \times S$ entre os estados
- uma avaliação V dando, para cada estado $s \in S$, as proposições atômicas que são verdadeiras naquele estado: $V(s) \subseteq Prop$

Pense em S como a modelagem dos estados de um sistema, em R como a modelagem dos passos da computação *atômica*, e em V como descrevendo as propriedades básicas dos estados.

Caminhos através da estrutura Kripke então correspondem a possíveis execuções do sistema!

Estruturas Kripke - um Exemplo



- conjunto de estados: $S = \{ s_0, s_1, s_2 \}$
- conjunto de estados iniciais: $\{ s_0 \}$
- relação de transição:
  $\{ (s_0, s_1), (s_0, s_2), (s_1, s_0), (s_1, s_2), (s_2, s_2) \}$
  $$s_0 \longrightarrow s_2 \qquad s_2 \not\longrightarrow s_0$$
- avaliação V fornece a rotulagem dos estados com proposições atômicas:
  $$V(s_0) = \{a, b\}$$
  $$V(s_1) = \{b, c\}$$
  $$V(s_2) = \{c\}$$

Exemplo: Protocolo de Exclusão Mútua

bool *turn*;
$P \ = \ m :$ **cobegin** $P_0 \parallel P_1$ **coend** m'

$P_0 \ = \ n_0 :$ **while** *True* **do**
  $t_0 :$ **wait** $(turn = 0)$;
  $c_0 :$ `use resource;` $turn := 1$;
endwhile ; n_0'

$P_1 \ = \ n_1 :$ **while** *True* **do**
  $t_1 :$ **wait** $(turn = 1)$;
  $c_1 :$ `use resource;` $turn := 0$;
endwhile ; n_1'

Note: **wait** (c) repetidamente testa c até que o mesmo se torne verdadeiro.

Extrair uma estrutura Kripke que modele $P_0 \parallel P_1$:

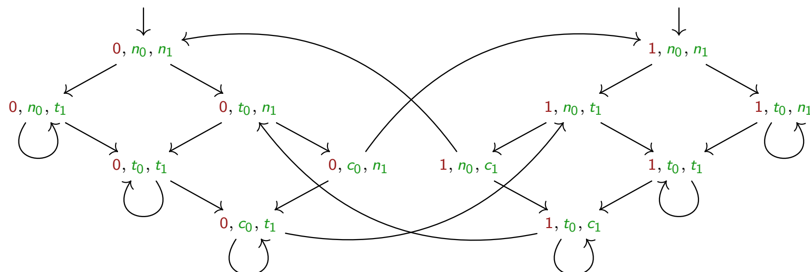
- estados são dados por pares de estados de P_0 e P_1 (valores dos contadores de processo), junto com o valor da variável compartilhada *turn*
- transições correspondem a passos de execução atômica em P_0 *ou* P_1 (sistema assíncrono)
- *Prop* e *V* dependerão das propriedades que nós queremos verificar . . .

Exclusão Mútua: o Modelo

bool *turn*;
$P \;=\; m$: **cobegin** $P_0 \parallel P_1$ **coend** m'

$P_0 \;=\; n_0$: **while** *True* **do**
  t_0 : **wait** ($turn = 0$);
  c_0 : `use resource;` $turn := 1$;
endwhile ; n_0'

$P_1 \;=\; n_1$: **while** *True* **do**
  t_1 : **wait** ($turn = 1$);
  c_1 : `use resource;` $turn := 0$;
endwhile ; n_1'



(Material sobre extração de estruturas Kripke a partir de programas concorrentes no capítulo 2 do Clarke, Grumberg e Peled.)

Exercício

Construa a estrutura Kripke associada ao seguinte algoritmo de exclusão mútua:

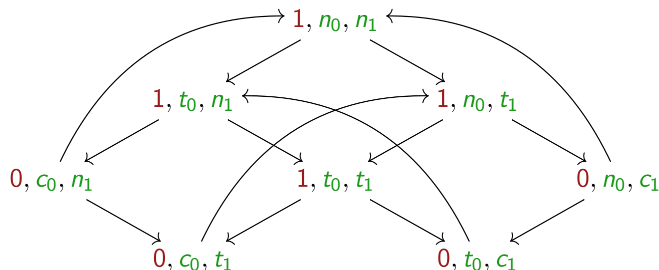
bool $turn$;
bool $flag0 = 0, flag1 = 0$;
P = **cobegin** $P_0 \parallel P_1$ **coend**

P_0 = **while** $True$ **do**
  $flag0, turn := 1, 1$;
  wait $((flag1 = 0) \,||\, (turn = 0))$;
  `use resource;` $flag0 := 0$;
endwhile ;

P_1 = **while** $True$ **do**
  $flag1, turn := 1, 0$;
  wait $((flag0 = 0) \,||\, (turn = 1))$;
  `use resource;` $flag1 := 0$;
endwhile ;

Suposições de Justiça: Exemplo

Um algoritmo de exclusão mútua (baseado em semáforo):



É "justo" que o segundo processo tenha infinitamente várias possibilidades de entrar na região crítica, mas *nunca entre nela*?

É "justo" que o segundo processo tenha infinitamente várias possibilidades de entrar na região crítica, mas *somente entre nela finitamente várias vezes*?

Suposições de Justiça

- captura a idéia de que o escalonamento dos processos é “justo”
  - cada processo tem o direito de executar infinitas vezes
- isto é *assumido*, não exigido:
  - somente execuções “justas” são exploradas pelo verificador de modelos
- tipicamente necessárias para provar as propriedades de vivacidade (liveness)
  - geralmente algumas execuções (aquelas que são “justas”) refutam as propriedades de vivacidade

Suposições de Justiça (Continuação)

- diferentes graus de justiça: um processo tem o direito de executar infinitas vezes ...
  - ...sempre $\Rightarrow$ unconditional fairness
  - ...se é habilitado infinitas vezes $\Rightarrow$ strong fairness
  - ...se é continuamente habilitado a partir de algum ponto $\Rightarrow$ weak fairness
- qual suposição é *suficiente* depende do exemplo em particular
  - no exemplo anterior, strong fairness é necessário
- ...mas a suposição deve ser satisfeita pelo escalonador !
- este é a *justiça baseada em processo*, nós também estudaremos mais adiante a *justiça baseado em ação* ...

Alguns Problemas em Sistemas Concorrentes e Distribuídos

Competição por recursos compartilhados leva a:

- deadlock: dois ou mais processos param e esperam um pelo outro
- livelock: dois ou mais processos continuam a executar, mas não realizam nenhum progresso rumo ao objetivo final
- starvation: algum processo fica adiado para sempre/um evento particular é impedido de acontecer

Propriedades de Corretude

- propriedades de segurança: nada de “ruim” jamais acontecerá (ao longo de qualuqer possível execução do sistema)
  - ausência de deadlock: um processo *nunca* entra em um estado que não possa sair
  - invariantes do sistema: alguma propriedade (desejável) é verdadeira *em todos os estados futuros*
- propriedades de vivacidade: alguma coisa “boa” eventualmente acontece (ao longo de todas as possíveis execuções do sistema)
  - ausência de livelock/starvation
  - progresso: um evento particular acontece *eventualmente*/*infinitas vezes*
- propriedades temporais mais complexas dos sistemas
  - capacidade de resposta: toda solicitação é eventualmente seguida por uma resposta

Exclusão Mútua: Propriedades de Segurança

- exclusão mútua: no máximo um processo na seção crítica a qualquer momento

Propriedades de segurança são violadas em tempo finito.

(Isto não é verdade para propriedades de vivacidade.)

Exclusão Mútua: Propriedades de Vivacidade

- eventualmente: cada processo entrará eventualmente em sua região crítica
- repetida eventualmente: cada processo entrará na sua região crítica *infinitas vezes*
- starvation freedom: cada processo de *espera* entrará eventualmente na sua região crítica

Propriedades de vivacidade são violadas em tempo infinito.

Exclusão Mútua: a Especificação

Propriedades de Corretude:

- **exclusão mútua** (segurança): no máximo um processo na região crítica *a qualquer momento*
- **starvation freedom** (vivacidade): *sempre que* um processo tenta entrar na sua região crítica, ele *eventualmente* irá suceder
- **sem bloqueio** (vivacidade): um processo pode sempre, em algum ponto no futuro, solicitar para entrar na sua região crítica

Exercício

Considere os modelos anteriores dos protocolos de exclusão mútua.

Verifique (por inspeção os modelos/protocolos) se as três propriedades de corretude fazem parte neste modelo.

Isto depende de quaisquer *suposições de justiça*?