NSWI101: System Behaviour Models and Verification

9. Abstractions and Symmetries

Jan Kofroň

Faculty of Mathematics and Physics
Charles University

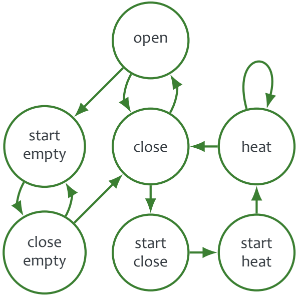
Department of Distributed and Dependable Systems

TODAY

- Abstractions
- Symmetries and Partial order reduction

Part I: Abstractions

ABSTRACTIONS



System model

AG (start $\rightarrow$ AF heat)

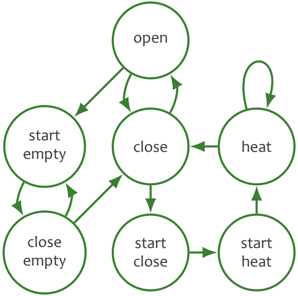
Property specification

Model Checker

Property satisfied

Property violated

Abstractions



System model

AG (start $\rightarrow$ AF heat)

Property specification

Model Checker

Property satisfied

Property violated

Abstractions

Ways to reduce (simplify) state space – optimizations – realized as elimination of some details of model

Particular abstractions:

- *Cone of influence reduction*
- *Data abstraction*

Cone of Influence Reduction (CoIR)

- Focus just on variables related to specification, i.e., those in formula to be model checked
- Variables not influencing values of variables in specification can be removed – they cannot affect whether the spec is valid or not

CoIR – Definition

- Let S be synchronous circuit described by set of equations $v_i = f_i(V)$
  - $v_i \in V$
  - f_i are Boolean functions
- Let specification contain set of variables $V' \subseteq V$
- Some $x \in V'$ can depend on $y \notin V'$
- We define set $C \supseteq V'$ of interest – C as *cone*

CoIR – Definition

The *cone of influence* C of V' is the minimal set of variables such that:

- $V' \subseteq C$, and
- $\forall k, j : (v_k \in C) \wedge (f_k \text{ depends on } v_j) \implies v_j \in C$

New (reduced) system is constructed from original by removing all equations whose left-hand-side variables do not appear in C

Correctness of Reduction

Theorem: Let f be CTL formula and M Kripke structure. Let M' be Kripke structure after CoIR of M with respect to f. Then $M \models f \Leftrightarrow M' \models f$.

Proof idea: Removing variables not in C and adding transitions existing in original states with identical values.

CoIR – Example

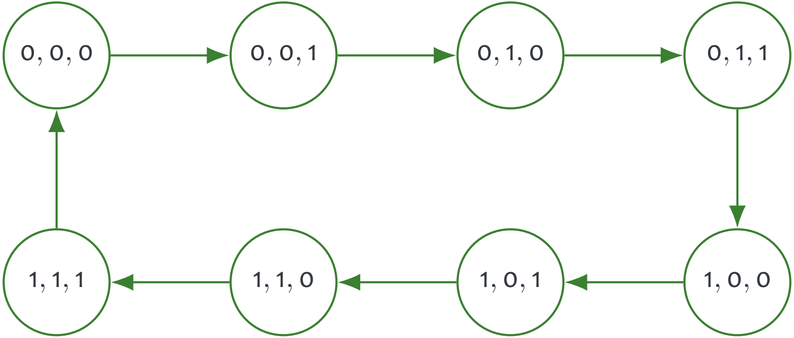
Specification comprising three variables:

- $v_1' = \neg v_1$
- $v_2' = v_1 \oplus v_2$
- $v_3' = (v_1 \wedge v_2) \oplus v_3$

Corresponding Kripke structure $M = (S, I, R, L)$ over variables $V = \{v_1, \ldots, v_n\}$:

- $S = \{0, 1\}^n$
- $I \subseteq S$
- $R = \bigwedge_{i=1}^{n} [v_i' = f_i(V)]$
- $L(s) = \{v_i | s(v_i) = 1, 1 \leq i \leq n\}$

Original State Space



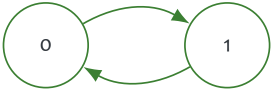
Reduced Model

Let $C = \{v_1, ..v_k\}$ for some k be CoI. Reduced model is $M' = (S', I', R', L')$:

- $S' = \{0, 1\}^k$
- $I' = \{(d'_1, ..d'_k) | \exists (d_1, .., d_n) \in I : d'_i = d_i, 1 \leq i \leq k\}$
- $R' = \bigwedge_{i=1}^{k} [v'_i = f_i(V)]$
- $L'(s') = \{v_i | s'(v_i) = 1, 1 \leq i \leq k\}$

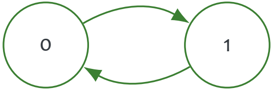
Reduced State Space

$$V' = \{v_1\} \rightarrow C = \{v_1\}$$

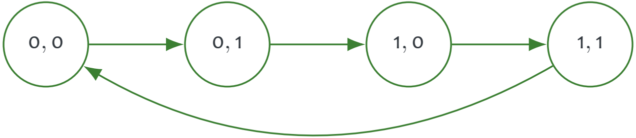


Reduced State Space

$$V' = \{v_1\} \rightarrow C = \{v_1\}$$

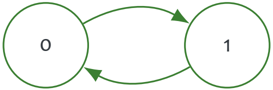


$$V' = \{v_2\} \rightarrow C = \{v_1, v_2\}$$

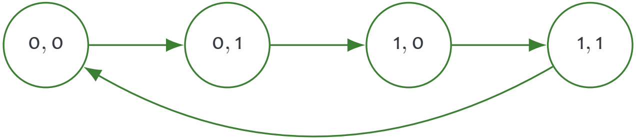


Reduced State Space

$V' = \{v_1\} \rightarrow C = \{v_1\}$



$V' = \{v_2\} \rightarrow C = \{v_1, v_2\}$



$V' = \{v_3\} \rightarrow C = \{v_1, v_2, v_3\}$ – original state space

Back to Proof Sketch

Let $B \subseteq S \times S'$ be defined as follows: $((d_1, .., d_n), (d'_1, .., d'_k)) \in B \Leftrightarrow d_i = d'_i, \forall 1 \leq i \leq k$
It suffices to show that B **is bisimulation**

- bisimulation implies $M \models f \Leftrightarrow M' \models f$
- first consider initial states and then all transitions and target states
- it is easy to see that it is bisimulation ;-)

Data Abstraction – Motivation

- Number of combinations of possible values of (user) input can be enormous
- Results in very large or sometimes even infinite (floating point numbers) state space
- Model checking hard or not possible in principle
- Solution: **Data abstraction**

Data Abstraction – Procedure

1. Define abstract domain(s) and map concrete values to abstract ones
2. Create reduced Kripke structure
   2.1 Replace concrete AP with abstract AP
   2.2 Merge states with same set of AP
3. Model checking

1. Abstract Domains

- Motivation is to significantly lower number of possible values for selected variables
- Abstraction means *hiding* some information
- Done by mapping each **concrete** value to **abstract** one, e.g., integer domain can be mapped to abstract domain
- Note that in Kripke structure data are encoded in (Boolean) atomic propositions

Data Abstraction – Example

Let $A = \{a_0, a_+, a_-\}$ be abstract domain and $h(x)$ mapping (abstraction) function

For *int i*: $h(i) = a_0$ if $i = 0$
$h(i) = a_+$ if $i > 0$
$h(i) = a_-$ if $i < 0$

Corresponding atomic propositions for concrete variables: a_0, a_+, a_-

2. Creating Reduced Kripke Structure

1. Create $M' = (S, I, R, L')$ such that it is identical to M except for L – L' labels states with **abstract** atomic propositions
2. Create reduced Kripke structure $M_r = (S_r, I_r, R_r, L_r)$:
   - $S_r = \{L'(s) | s \in S\}$ – merging states with identical set of AP
   - $s_r \in I_r \Leftrightarrow \exists s \in S : s_r = L'(s) \wedge s \in I$
   - $(s_r, t_r) \in R_r \Leftrightarrow \exists s, t \in S : (s, t) \in R \wedge s_r = L'(s) \wedge t_r = L'(t)$

3. Model Checking

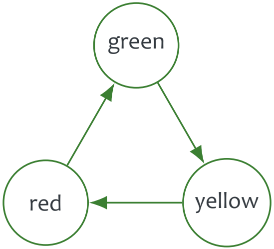
Perform model checking of M_r

Desired property: $M_r \models f_r \implies M \models f$. Does this hold for any CTL formula?

Perform model checking of M_r

Desired property: $M_r \models f_r \implies M \models f$. Does this hold for any CTL formula?

Example: Traffic lights

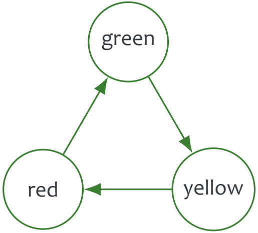


Data Abstraction – Example

Original atomic propositions $AP = \{red, yellow, green\}$, in each state exactly one is true

Abstract domain $A = \{stop, go\}$

Mapping function h: $h(red) = stop$, $h(yellow) = stop$, $h(green) = go$



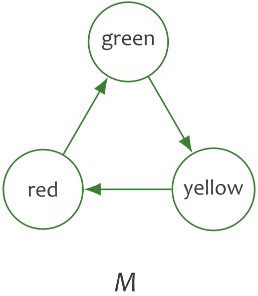
M

Data Abstraction – Example

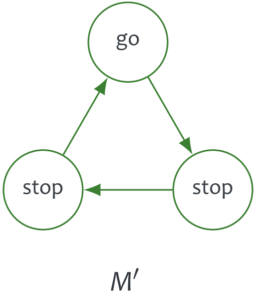
Original atomic propositions $AP = \{red, yellow, green\}$, in each state exactly one is true

Abstract domain $A = \{stop, go\}$

Mapping function h: $h(red) = stop$, $h(yellow) = stop$, $h(green) = go$



$\longrightarrow$

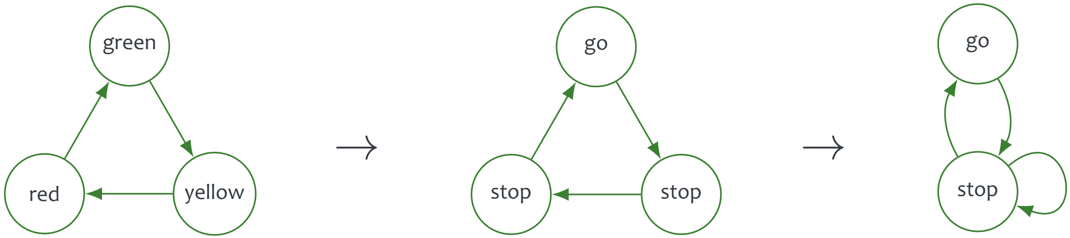


Data Abstraction – Example

Original atomic propositions $AP = \{red, yellow, green\}$, in each state exactly one is true

Abstract domain $A = \{stop, go\}$

Mapping function h: $h(red) = stop$, $h(yellow) = stop$, $h(green) = go$



M $\longrightarrow$ M' $\longrightarrow$ M_r

Our Property

Desired property: $M_r \models f_r \implies M \models f$. Does this hold for any CTL formula?

OUR PROPERTY

Desired property: $M_r \models f_r \implies M \models f$. Does this hold for any CTL formula? **NO!**

Our Property

Desired property: $M_r \models f_r \implies M \models f$. Does this hold for any CTL formula? **NO!**

Consider formula AG $(red \wedge \mathrm{EX}\, yellow)$ which is **not** satisfied in red state.

After abstraction formula reads AG $(stop \wedge \mathrm{EX}\, stop)$ and this formula **is** satisfied in stop state of M_r.

So what is it good for?!?

DATA ABSTRACTION – JUSTIFICATION

We cannot do arbitrary abstractions – this way we could reduce any KS into one with just one state and possibly self loop, which is apparently not correct.

Data Abstraction – Justification

We cannot do arbitrary abstractions – this way we could reduce any KS into one with just one state and possibly self loop, which is apparently not correct.

Two options:

1. Limit allowed abstractions

2. Limit language of formulae

Data Abstraction – Justification

We cannot do arbitrary abstractions – this way we could reduce any KS into one with just one state and possibly self loop, which is apparently not correct.

Two options:

1. Limit allowed abstractions
   - only *exact abstractions* allowed – those congruent with respect to primitive relations (transition relation, set of initial states)
2. Limit language of formulae

Data Abstraction – Justification

We cannot do arbitrary abstractions – this way we could reduce any KS into one with just one state and possibly self loop, which is apparently not correct.

Two options:

1. Limit allowed abstractions
   - only *exact abstractions* allowed – those congruent with respect to primitive relations (transition relation, set of initial states)
2. Limit language of formulae
   - using **ACTL** – formulae in negative normal form without existential quantification

Part II: Symmetries and Partial Order Reduction

Symmetries

Concurrent systems often exhibit a lot of symmetry:

- memories
- caches
- buses

Identification of symmetric states can lead to substantial reduction of states space

- by checking just one representative of each symmetry group

We need to define what *symmetric* means in particular context/example

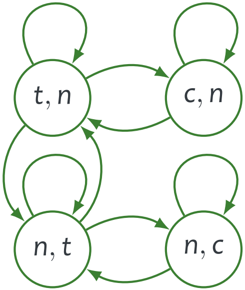
- property specification
- system model

Symmetries

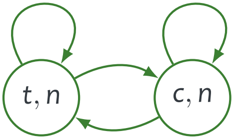
- From each set of symmetric states just one is picked, and its transitions are taken into account
- Formally, reduction is based on finding quotiens and invariant groups of automorphisms upon permutations
  - skipping algebraic theory here for sake of time :-)

Example – Token Ring Network

Token ring of two nodes



Reduced model based on symmetry:
$\{(t, n), (n, t)\}$ and $\{(c, n), (n, c)\}$

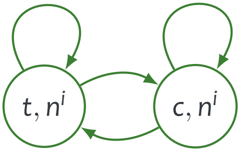


Example – Token Ring Network

We can apply reduction to larger configurations of i nodes

- obtaining again two symmetry groups:
  $\{(t, n^i), (n, t, n^{i-1}), \ldots, (n^i, t)\}$ and $\{(c, n^i), (n, c, n^{i-1}), \ldots, (n^i, c)\}$

This results to exactly the same state space

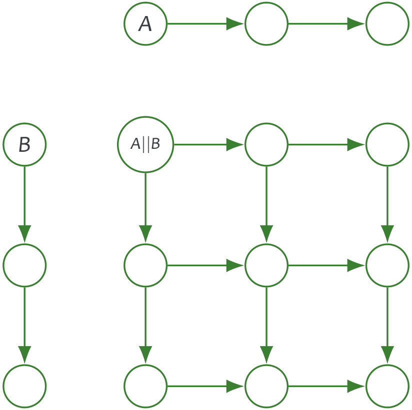


EXECUTION SYMMETRY: PARTIAL ORDER REDUCTION

Parallel composition (of processes) causes exponential growth of state space

If processes do not communicate too much, majority of states are equivalent to other states and do not need to be explored

This means exploring just some paths from initial to final state



Partial Order Reduction

Idea:

- Before model checking, reduced state graph is constructed
- Full state graph is never constructed
- Exploiting commutativity of concurrently executed transitions, which result in the same state when executed in different orders
- Formulated by Doron Peled in 1993

The name – Partial Order Reduction:

- Early versions were based on the partial order model of the program execution
- Better name: Model checking using representatives

State Transition System

It is convenient to formulate the algorithm upon STS rather than Kripke structure

- Kripke structure $M = (S, I, R, L)$
- Corresponding STS $N = (S, T, S_0, L)$:
  - $S, S_0 = I, L$ – identical to those in Kripke structure
  - T is set of transitions: $R(s, s') \Leftrightarrow \exists a \in T : a(s, s')$
  - Transitions are labelled
  - Transitions with the same label are considered the same transition

Transitions

- Transition a is *enabled* in state $s \Leftrightarrow \exists s' : a(s, s')$
- If transition is not enabled, it is *disabled*
- *enabled*(s) refers to all enabled transition is state s
- Transition a is deterministic
  $\Leftrightarrow \forall s : a$ is enabled in s, there exists at most one state $s' : a(s, s')$
  - We can write $s' = a(s)$
  - Only deterministic systems will be considered

Reduction Algorithm

```
expand_state(so)

function expand_state(s) {
  work_set = ample(s);
  while work_set is not empty {
    choose a from work_set
    work_set = work_set \ {a}
    t = a(s)
    if new(t)
      expand_state(t)

    create_edge(s, a, t);
  }
}
```

Ample Set: Requirements

Systematic way of computing ample sets required

Desired properties of function ample(s):

1. Sufficiently many behaviours must be present in reduced state graph, so that algorithm provides correct results
2. Reduced state graph should be significantly smaller than full graph
3. Overhead of computing ample(s) must be reasonably small

Important notions are *independence* and *invisibility* of transitions

INDEPENDENCE

Definition: *Independence relation* $I \subseteq T \times T$ is symmetric, anti-reflexive relation satisfying following two conditions:

- enabledness: $a, b \in enabled(s) \implies a \in enabled(b(s))$
- commutativity: $a, b \in enabled(s) \implies a(b(s)) = b(a(s))$

Definition exploits symmetry of relation

Dependency relation D is complement of independence relation I: $D = (T \times T) \setminus I$

Independence relation to be specified:

- obtained either from computational model
- or knowledge of modelled system

Even actions that cannot be executed in parallel, e.g., incrementing variable by several processes, can be independent

Independence in Spin

Let *a*, *b* be transitions performed by different processes. *a*, *b* are independent if:

- *a* accesses local variable of its process, *b* is arbitrary transition
- *a*, *b* access two different global variables or channels
  - Also including sending and receiving messages on different channels, and testing length of different channels
- *a*, *b* read one global variable (or test length of one channel)
- *a* is send operation on channel chan, *b* is receive operation on chan, provided that chan is asynchronous and default behaviour of send is used (i.e., send on full channel is blocked)

Invisibility And Stuttering Equivalence

Definition: Transition is called *invisible* if both origin and target states satisfy same set of atomic propositions.

- can be restricted to subset of AP
- invisible $\sim$ no visible change after executing the transition

Each path can be split into blocks, where each block contains states satisfying the same set of AP

Definition: Two paths are *stuttering equivalent* iff they contain the same blocks (w.r.t. AP) in the same order, possibly differing just in lengths.

- minimal length of each block is one
- block length is always finite

Stuttering Equivalence of Structures

Two structures M, M' (Kripke structures or state transition systems) are stuttering equivalent iff:

- M and M' have the same set of initial states
- for each path σ of M that starts from initial state s of M there exists path σ' of M' from the same initial state s such that $\sigma \sim_{st} \sigma'$
- for each path σ' of M' that starts from initial state s of M' there exists path σ of M from the same initial state s such that $\sigma' \sim_{st} \sigma$

Stuttering Equivalence in LTL

LTL formula is *invariant under stuttering* iff for each pair of paths π and π' such that $\pi \sim_{st} \pi' : \pi \models f \Leftrightarrow \pi' \models f$.

We denote LTL without next operator by LTL_{-X}

Theorem: Any LTL_{-X} property is invariant under stuttering.

Theorem: Every LTL property that is invariant under stuttering can be expressed in LTL_{-X}.

Theorem: Let M, M' be two stuttering equivalent structures. Then, for every LTL_{-X} property f, and every initial state s: $M, s \models f \Leftrightarrow M', s \models f$.

Idea: Partial Order Reduction generates stuttering equivalent structure and model-checks just this smaller structure

Partial Order Reduction for LTL$_{-X}$

Independence and invisibility are not enough, reduction has to avoid postponing transitions forever inside cycles

State s is *fully expanded* iff $ample(s) = enabled(s)$

Four conditions to be satisfied by *ample(s)* function:

C0. $ample(s) = \emptyset \Leftrightarrow enabled(s) = \emptyset$

C1. Along every path in full state graph that starts at s it holds that transition dependent on transition in $ample(s)$ cannot be executed without transition in $ample(s)$ occurring first

C2. If s is not fully expanded, then every $a \in ample(s)$ is invisible

C3. Cycle is not allowed if it contains state in which some transition a is enabled, but is never included in $ample(s)$ for any state s of the cycle

POR in Java PathFinder

Java PathFinder is explicit code model checker for Java programs

In principle special virtual machine executing “all” possible thread interleavings and “trying” all specified (random) input values

Since there are exponentially (in size and number of threads) many thread interleavings, switch only when it makes sense:

- For example, sequential update of local variables cannot affect other threads
- Consider just interesting instruction as re-scheduling points:
  - **scheduling-relevant instructions**
  - **non-deterministic instructions**

POR in Java PathFinder

Only about 10% are scheduling-relevant instructions:

- synchronization (monitorEnter, monitorExit, invokeX on synchronized methods)
- field access (putX, getX)
- array element access (Xaload, Xastore)
- thread instructions (start, sleep, yield, join)
- object methods (wait, notify)