Behavior Models and Verification

Lecture 8

Jan Kofroň, František Plášil
• Infinite families of finite-state systems
• Bounded model checking
  ▪ HW verification
• Compositional reasoning
Infinite / Bounded Model Checking

Kripke structure

Model

Property specification

AG(start → AF heat)

Model checker

Property satisfied

Property violated

Error report

Jan Kofroň, František Plášil, Lecture 8
Infinite families of finite states systems
Infinite families

- Finite model checking is fine
  - we know that generally model checking infinite state spaces is undecidable

- **But:** Protocols and circuits specification can be parameterized, e.g.:
  - size of int in multiplication unit of CPU
  - number of processors connected to bus
  - ...

Jan Kofroň, František Plášil, Lecture 8
Infinite families

- Indeed, it would be handy to reason about such parameterized designs (models)

- Formally – infinite family of systems:

\[ \mathcal{F} = \{ M_i \}_{i=1}^{\infty} \]

- For temporal formula \( f \) verify that:

\[ \forall i: M_i \models f \]

- This is generally still undecidable, though
Temporal logic for infinite families

- Having indexed specification, indexed formulae would be fine as well

- **Indexed CTL (ICTL)**
  - formula indexed by integer
  - \(i\)-th formula applies to \(i\)-th component

- ICTL allows for expressing: \(\wedge_i f(i)\) and \(\vee_i f(i)\)
  - and also: \(\wedge_{j \neq i} f(j)\) and \(\vee_{j \neq i} f(j)\)
Token ring example

- Simple token ring algorithm
  - n – non-critical section, t – keeping token,
    c – critical section, r – receive token, s – send token

- One process $Q$ and several $P_i$ of this type
  - $Q$ initially in $t$, $P_i$ initially in $n$
Token ring example

- Synchronous $Q \ || \ P$ composition, natural synchronizing on $s$ and $r$ resulting in $\tau$
• Token ring family: $\mathcal{F} = \{Q \parallel P_i\}_{i=1}^\infty$

• Desired property: $\bigwedge_i \text{AG} (c_i \Rightarrow \bigwedge_{j \neq i} \neg c_j)$
  i.e., if process $i$ is in critical section, then no other process is...
Invariants

Let \( F = \{ M_i \}_{i=1}^{\infty} \) be family of structures
Let \( \geq \) be reflexive, transitive relation on structures
Invariant \( I \) is structure such that
\[
\forall i: I \geq M_i
\]
properties that can be checked are determined by \( \geq \):
- bisimulation (strong preservation: \( I \models f \iff M \models f \))
- simulation preorder (weak preservation: \( I \models f \Rightarrow M \models f \))
- language equivalence (strong preservation)
- language preorder (weak preservation)
Token rings of size $n$ and of size 2 are in simulation preorder.

So for any CTL property $f$ it is sufficient to verify:

$$(P \parallel Q) \models f$$
Lemma: Let $\geq$ be a reflexive, transitive relation and let $\parallel$ be a composition operator that is monotonic w.r.t. $\geq$. If $I \geq P$ and $I \geq I \parallel P$, then $\forall i: I \geq P^i$, where $\mathcal{F} = \{P^i\}_{i=1}^\infty$.

$\parallel$ is monotonic w.r.t. $\geq \iff \forall P_1, P_1', P_2, P_2':
\begin{align*}
P_1 \geq P_1' \land P_2 \geq P_2' \Rightarrow P_1 \parallel P_2 \geq P_1' \parallel P_2'
\end{align*}
More systematic approach

• This is more like:
  “This holds once we have the relation”
  than
  “How to find the relation”

• Finding a suitable relation is hard, not possible in automatic way
  ▪ recall: the problem is undecidable in general
Bounded model checking
Bounded model checking

- Let $M = \{S, I, R, L\}$ be Kripke structure
- Define predicate $Reach(s, s')$ iff $R(s, s')$
- Define $\lceil M \rceil^k = \bigwedge_{i=0}^{k-1} Reach(s_i, s_{i+1})$
- $\lceil M \rceil^k$ contains states reachable in $k$ steps
- Idea: Look for counterexamples made of $k$ states
Bounded model checking

\( M, \neg \varphi \)

\( k = 0 \)

\( \neg \varphi \) satisfiable in \([M]^k\)

\( inc(k) \)

\( k < \text{threshold} \)

\( M \models \neg \varphi \)

\( M \not\models \neg \varphi \)
Bounded model checking

$k=0$
Bounded model checking

k=1
Bounded model checking

\[ k=2 \]
Bounded model checking

\[ k=3 \]
Bounded model checking

k=4
Bounded model checking

$k=5$
Mean: Construction of formula describing the transitions in the program
- and trying to reach assertion violation, i.e., violation of $AG\ p$
- checking for satisfiability of the formula
  - using SAT solver
SAT solvers

- Tools taking logical formula and deciding whether it is satisfiable
  - whether there is satisfying assignment of free variables
  - formula in conjunctive normal form (CNF)
  - can contain quantifiers → harder problem
  - NP-complete problem
  - if satisfiable → satisfying assignment
  - if not → unsat core (subset of formula’s clauses)
Example

1:  int  i=4;
2:  int  s=0;
3:  while (i) {
4:    s+=i;
5:    if (i>0)
6:      i--;
7:    assert(s<10);
8:  }
First step is unwinding loops (to cover the bound)

```
1: int i=4;
2: int s=0;
3: while (1) {
4:   s+=i;
5:   if (i>0) 
6:     i--;
7:   assert(s<10);
8: }
```
Example

1: int i=4;
2: int s=0;
3: f_1: (pc_1 = 1) ∧ (i_2 = 4) ∧ (pc_2 = 2)
4: s+=i;
5: if (i>0)
6: i--;
7: assert(s<10);
8: s+=i;
9: if (i>0)
10: i--;
11: assert(s<10);

...
Example

1: int i=4;
2: int s=0;
3:    s+=i;
4:    if (i>0)
5:       i--;
6:    assert(s<10);
7:    s+=i;
8:    if (i>0)
9:       i--;
10:   assert(s<10);
11:   ...
• Assertion expressions negated
• Main formula:

\[ \bigwedge_{i=0..k} f_i \]

• Satisfying assignment is found \( \rightarrow \) assertion is violated
• If not, we know that there is no assertion violation in \( k \) steps
HW application: Four-bit adder
HW implementation of addition operation (1-bit):

- $A, B$ – input bits, $C_{in}, C_{out}$ – carry bits, $S$ – output
Logical representation of BIT-ADDER

\[
((A \land B \land C_{in}) \Rightarrow (S \land C_{out})) \land \\
((\neg A \land B \land C_{in}) \Rightarrow (\neg S \land C_{out})) \land \\
((A \land \neg B \land C_{in}) \Rightarrow (\neg S \land C_{out})) \land \\
((A \land B \land \neg C_{in}) \Rightarrow (\neg S \land C_{out})) \land \\
((\neg A \land \neg B \land C_{in}) \Rightarrow (S \land \neg C_{out})) \land \\
((\neg A \land \neg B \land \neg C_{in}) \Rightarrow (S \land \neg C_{out})) \land \\
((A \land \neg B \land \neg C_{in}) \Rightarrow (S \land \neg C_{out})) \land \\
((\neg A \land \neg B \land \neg C_{in}) \Rightarrow (\neg S \land \neg C_{out}))
\]

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C_{in}</th>
<th>S</th>
<th>C_{out}</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Evaluation of $S$

$S_0 = A_0 \text{xor} B_0$

$C_{out,0} = A_0 \text{ and } B_0$

$S_1 = A_1 \text{xor} B_1 \text{xor} C_{out,0}$

$C_{out,1} = \left( (A_1 \text{xor} B_1) \text{ and } C_{out,0} \right) \text{ or } (A_1 \text{ and } B_1)$

$S_2 = A_2 \text{xor} B_2 \text{xor} C_{out,1}$

$C_{out,2} = \left( (A_2 \text{xor} B_2) \text{ and } C_{out,1} \right) \text{ or } (A_2 \text{ and } B_2)$

$S_3 = A_3 \text{xor} B_3 \text{xor} C_{out,2}$

$C_{out,3} = \left( (A_3 \text{xor} B_3) \text{ and } C_{out,2} \right) \text{ or } (A_3 \text{ and } B_3)$
Evaluation of $S$

$S_0 = A_0 \text{ xor } B_0$

$C_{out,0} = A_0 \text{ and } B_0$

$S_1 = A_1 \text{ xor } B_1 \text{ xor } C_{out,0}$

$C_{out,1} = \left( (A_1 \text{ xor } B_1) \text{ and } C_{out,0} \right) \text{ or } (A_1 \text{ and } B_1)$

$S_2 = A_2 \text{ xor } B_2 \text{ xor } C_{out,1}$

$C_{out,2} = \left( (A_2 \text{ xor } B_2) \text{ and } C_{out,1} \right) \text{ or } (A_2 \text{ and } B_2)$

$S_3 = A_3 \text{ xor } B_3 \text{ xor } C_{out,2}$

$C_{out,3} = \left( (A_3 \text{ xor } B_3) \text{ and } C_{out,2} \right) \text{ or } (A_3 \text{ and } B_3)$
Evaluation of $S$

\[ S_0 = A_0 \oplus B_0 \]
\[ C_{out,0} = A_0 \land B_0 \]
\[ S_1 = A_1 \oplus B_1 \oplus C_{out,0} \]
\[ C_{out,1} = \left( (A_1 \oplus B_1) \land C_{out,0} \right) \lor (A_1 \land B_1) \]
\[ S_2 = A_2 \oplus B_2 \oplus C_{out,1} \]
\[ C_{out,2} = \left( (A_2 \oplus B_2) \land C_{out,1} \right) \lor (A_2 \land B_2) \]
\[ S_3 = A_3 \oplus B_3 \oplus C_{out,2} \]
\[ C_{out,3} = \left( (A_3 \oplus B_3) \land C_{out,2} \right) \lor (A_3 \land B_3) \]
Evaluation of $S$

\[ S_0 = A_0 \text{ xor } B_0 \]
\[ C_{out,0} = A_0 \text{ and } B_0 \]
\[ S_1 = A_1 \text{ xor } B_1 \text{ xor } C_{out,0} \]
\[ C_{out,1} = (A_1 \text{ xor } B_1) \text{ and } C_{out,0} \text{ or } (A_1 \text{ and } B_1) \]
\[ S_2 = A_2 \text{ xor } B_2 \text{ xor } C_{out,1} \]
\[ C_{out,2} = (A_2 \text{ xor } B_2) \text{ and } C_{out,1} \text{ or } (A_2 \text{ and } B_2) \]
\[ S_3 = A_3 \text{ xor } B_3 \text{ xor } C_{out,2} \]
\[ C_{out,3} = (A_3 \text{ xor } B_3) \text{ and } C_{out,2} \text{ or } (A_3 \text{ and } B_3) \]
Evaluation of $S$

\[
S_0 = A_0 \text{xor} B_0
\]

\[
C_{out,0} = A_0 \text{ and } B_0
\]

\[
S_1 = A_1 \text{xor} B_1 \text{xor} C_{out,0}
\]

\[
C_{out,1} = \left( (A_1 \text{xor} B_1) \text{ and } C_{out,0} \right) \text{ or } (A_1 \text{ and } B_1)
\]

\[
S_2 = A_2 \text{xor} B_2 \text{xor} C_{out,1}
\]

\[
C_{out,2} = \left( (A_2 \text{xor} B_2) \text{ and } C_{out,1} \right) \text{ or } (A_2 \text{ and } B_2)
\]

\[
S_3 = A_3 \text{xor} B_3 \text{xor} C_{out,2}
\]

\[
C_{out,3} = \left( (A_3 \text{xor} B_3) \text{ and } C_{out,2} \right) \text{ or } (A_3 \text{ and } B_3)
\]
In each step, one bit of $S$ and one carry bit are computed.

To reason about any bit, four steps are enough.

- E.g., if we are interested in $C_{out,3}$ setting some flags.

That means that from the model of hw we can easily set the threshold for bounded model checking.
• We obtain a minimal counterexamples
  ▪ always the shortest found first
• Connected with loop invariants, properties of
  infinite paths can be verified
  ▪ this way, unbounded (infinite) models can be
    analyzed
  ▪ though not really model-checked
• If we manage to traverse entire state space, it
  is actually equal to unbounded MC
A step further

- If we manage to prove that particular number of steps covers all the states, we can even verify!
- This is called “unbounded model checking”
Compositional reasoning
Efficient verification algorithms can extend applicability of formal methods

Many systems can be decomposed into parts

- verifying each properties of parts separately
- if conjunction of parts properties implies overall specification, we are done
- the entire system never analyzed as whole
Example

- Communication protocol: Sender, network, receiver
- Overall specification:
  - Data correctly transmitted from sender to receiver
- Partial specifications
  - Data correctly sent from sender to network
  - Data correctly transmitted via network
  - Data correctly transmitted from network to receiver
- Verification of partial specifications typically much easier
  - sum of the state spaces much smaller than state space of entire system (recall state space explosion)
Assume-guarantee for composition reasoning

- Verifies each component separately
- Based on specification of
  - **Assumptions** – requirements on behavior of environment
  - **Guarantees** – provisions offered to environment if assumptions are met
- environment = the other components

- By combining assumptions and guarantees of particular parts, it is possible to establish correctness of entire system
  - Full transition graph never constructed
Formally

- Formula is triple $\langle g \rangle M \langle f \rangle$ where $g$ and $f$ are temporal formulae and $M$ is a program
  - true if whenever $M$ is part of system satisfying $g$ system also guarantees $f$

- Typically: proof for $\langle g \rangle M' \langle f \rangle$ and $\langle true \rangle M \langle g \rangle$
  $\rightarrow$ then we have $\langle true \rangle M \parallel M' \langle f \rangle$

- Can be expressed as inference rule:

\[
\frac{
\langle true \rangle M \langle g \rangle \\
\langle g \rangle M' \langle f \rangle
}{
\langle true \rangle M \parallel M' \langle f \rangle
} 
\]
Formally

- Necessary to avoid circular dependencies:
  \[
  \langle f \rangle M \langle g \rangle \\
  \langle g \rangle M' \langle f \rangle \\
  \hline
  M \parallel M' \models f \land g
  \]

- This is unsound!
- Should be avoided
Each component specifies not only provided (implemented) interfaces,
  - as objects do
But also required ones
  - in addition to objects

Syntactic (type) information
  - may or may not consider interface/type inheritance

Semantic (behavior) specification
  - verification techniques, usually model checking or equivalence checking (simulation, bisimulation, ...)

Applications – SW components
Can cover various aspects of the components
  - functional – sequences of messages/calls
  - extra-functional
    - timing
    - reliability
    - resource usage
    - security
    - ...

Composability verification based on the same principle
  - component should provide at least as much (as good, fast, reliable, ...) as its counterpart requires
Applications – Code

• Types – usually checked by compiler
  ▪ no additional effort required

• Semantics – code contracts
  ▪ at level of functions/methods
  ▪ assumptions – preconditions
  ▪ guarantees – postconditions
  ▪ usually also invariants
    ▪ helps with verification
    ▪ loop invariants
• Verification is then **modular**

  - each function is verified separately – whether the code really guarantees postcondition once precondition is satisfied at function entry
  - if function is called from within other function, its contract is assumed (precondition is checker, postcondition is assumed)
public class ArrayList {
    public void add(int index, Object obj) { ... }
    public int size() { ... }
}

• “Value of the index parameter has to be greater than or equal to zero. Successful call of add increases the size of the array by one.”

• Formally:

public void add(int index, Object obj)
    
    requires index >= 0;
    
    ensures size = old(size) + 1;

    { ... }
Contract specification languages
- Spec#, JML, Code Contracts, ...

There are tools to verify contracts
- model checkers, SAT/SMT solvers, theorem provers

NSWI132 – Program analysis and code verification
Code level – Remarks

- It is not easy to specify the contracts
  - preconditions
    - too weak to guarantee postconditions
    - too strong to be satisfied by caller
  - postconditions
    - too strong to be proven
    - too weak to “satisfy” caller

- One has to know and tune...
int[N] field;

int swapMin(int from)
{
    swaps the min value beyond from with the one at from and return the index
}

int main()
{
    // sorted
    ensures (forall int i : 0<i<N-2 : field[i] <= field[i+1]);
    // the original values
    ensures (forall int i : 0<i<N-1 && exists int j : old(field[j]) == field[i]);
    
    for (int i = 0; i < N; i++)
        swapMin(i);
}
```c
int[N] field;

int swapMin(int from)
{
    ensures ((field[return] == old(field[from]) && old(field[from]) == field[return]));
    { swaps the min value beyond from with the one at from and return the index }
}

int main()
{ // sorted
    ensures (forall int i : 0<i<N-2 : field[i] <= field[i+1]);
    // the original values
    ensures (forall int i : 0<i<N-1 && exists int j : old(field[j]) == field[i]);
    { for (int i = 0; i < N; i++)
        swapMin(i);
    }
}
```

Holds, but too weak!
int[N] field;

int swapMin(int from)
    ensures (forall int i<from: field[i]<=field[from]);
{
    swaps the min value beyond from with the one at from and return the index
}

int main()
// sorted
ensures (forall int i : 0<i<N-2 : field[i] <= field[i+1]);
// the original values
ensures (forall int i : 0<i<N-1 && exists int j : old(field[j]) == field[i]);
{
    for (int i = 0; i < N; i++)
        swapMin(i);
}
Based on checking model level specification with code
- whether code complies to model spec
- again modular – at granularity of
  - functions/methods
  - objects
  - sw components
- similar to code contracts but usually coarser granularity
  - e.g., limited to sequences of method calls

Allows for checking compositionality at model level
- which is usually easier than at code level
- handling components as annotated black boxes
- if strong enough, entire problem **undecidable**
  - code model checking \( \rightarrow \) halting problem
Decision Procedures and Verification

- NAIL094
- Summer semester
- Inside into algorithms and techniques inside SAT (and SMT) solvers
• NSWI132
• Summer semester
• Insight into code verification tools
  ▪ Focus on practical experience with the code verifiers
If you like area of verification, do not hesitate to contact us!

- Jan Kofroň: jan.kofron@d3s.mff.cuni.cz
- Pavel Parízek: pavel.parizek@d3s.mff.cuni.cz