Today

- Infinite families of finite-state systems
- Bounded model checking
  - HW verification
- Compositional reasoning
Infinite / Bounded Model Checking

- Markov chains
- Timed automata
- Labelled transition system
- Kripke structure

Model

- open
- start
- close
- empty
- heat

Property specification:

\[ AG(\text{start} \rightarrow AF \text{heat}) \]

Model checker

Error report

Property satisfied

Property violated
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
  - ...

Infinitive families

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

• Formally – infinitle 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: $\bigwedge_i f(i)$ and $\bigvee_i f(i)$
  - and also: $\bigwedge_{j \neq i} f(j)$ and $\bigvee_{j \neq i} f(j)$
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 \( n \)
Synchronous $Q \parallel P$ composition, natural synchronizing on $s$ and $r$ resulting in $\tau$
Token ring family: \( \mathcal{F} = \{ Q \parallel P_i \}_{i=1}^{\infty} \)

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

- Let $\mathcal{F} = \{M_i\}_{i=1}^{\infty}$ 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 $\|\|$ be a composition operator that is monotonic w.r.t. $\geq$. If $I \geq P$ and $I \geq I \| P$, then $\forall i: I \geq P^i$, where $\mathcal{F} = \{P^i\}_{i=1}^\infty$.

$\|\|$ is monotonic w.r.t. $\geq \iff \forall P_1, P_1', P_2, P_2'$:

$$P_1 \geq P_1' \land P_2 \geq P_2' \Rightarrow P_1 \| P_2 \geq P_1' \| P_2'$$
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
Let $\mathcal{M} = \{S, I, R, L\}$ be Kripke structure

Define predicate $\text{Reach}(s, s')$ iff $R(s, s')$

Define $\llbracket \mathcal{M} \rrbracket^k = \land_{i=0}^{k-1} \text{Reach}(s_i, s_{i+1})$

$\llbracket \mathcal{M} \rrbracket^k$ contains states reachable in $k$ steps

Idea: Look for counterexamples made of $k$ states
Bounded model checking

\[ M, \neg \varphi \]

\[ k = 0 \]

\[ \neg \varphi \text{ satisfiable in } [M]^k \]

\[ \text{NO} \]

\[ \text{YES} \]

\[ \text{inc}(k) \]

\[ k < \text{threshold} \]

\[ \text{NO} \]

\[ \text{YES} \]

\[ M \models \neg \varphi \]

\[ M \not\models_k \neg \varphi \]
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

\[ k = 5 \]
Bounded model checking for programs

- 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 $\rightarrow$ harder problem
  - NP-complete problem
  - if satisfiable $\rightarrow$ satisfying assignment
  - if not $\rightarrow$ unsat core (subset of formula’s clauses)
Example

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: `}`
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: }

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);
...
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: ...

\[ \begin{align*}
  f_1 &: (pc_1 = 1) \land (i_2 = 4) \land (pc_2 = 2) \\
  f_2 &: (pc_2 = 2) \land (i_3 = i_2) \land (s_3 = 0) \land (pc_3 = 3) \\
  f_3 &: (pc_3 = 3) \land (i_4 = i_3) \land (s_4 = s_3) \land (pc_4 = 4) \\
  f_4 &: (pc_4 = 4) \land (i_5 = i_4) \land (s_5 = s_4 + i_4) \land (pc_5 = 5) \\
  f_5 &: (pc_5 = 5) \land (i_6 = i_5) \land (s_6 = s_5) \land (pc_6 = 6) \\
  f_6 &: (pc_6 = 6) \land \left( ((i_6 > 0) \land (i_7 = i_6 - 1)) \lor ((i_6 \leq 0) \land (i_7 = i_6)) \right) \\
       & \land (s_7 = s_6) \land (pc_7 = 7) \\
  f_7 &: (pc_7 = 7) \land (s_7 \geq 10) \land (pc_8 = 8) \\
  f_8 &: (pc_8 = 8) \land (i_9 = i_8) \land (s_9 = s_8 + i_8) \land (pc_9 = 9) \\
  f_9 &: (pc_9 = 9) \land (i_{10} = i_9) \land (s_{10} = s_9) \land (pc_{10} = 10) \\
  f_{10} &: (pc_{10} = 10) \land \left( ((i_{10} > 0) \land (i_{11} = i_{10} - 1)) \lor ((i_{10} \leq 0) \land (i_{11} = i_{10})) \right) \\
       & \land (s_{11} = s_{10}) \land (pc_{11} = 11) \\
  f_{11} &: (pc_{11} = 11) \land (s_{11} \geq 10) \land (pc_{12} = 12)
\end{align*} \]
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: ...

\[ f_1: (pc_1 = 1) \land (i_2 = 4) \land (pc_2 = 2) \]
\[ f_2: (pc_2 = 2) \land (i_3 = i_2) \land (s_3 = 0) \land (pc_3 = 3) \]
\[ f_3: (pc_3 = 3) \land (i_4 = i_3) \land (s_4 = s_3) \land (pc_4 = 4) \]
\[ f_4: (pc_4 = 4) \land (i_5 = i_4) \land (s_5 = s_4 + i_4) \land (pc_5 = 5) \]
\[ f_5: (pc_5 = 5) \land (i_6 = i_5) \land (s_6 = s_5) \land (pc_6 = 6) \]
\[ f_6: (pc_6 = 6) \land ((i_6 > 0) \land (i_7 = i_6 - 1)) \lor ((i_6 \leq 0) \land (i_7 = i_6)) \land (s_7 = s_6) \land (pc_7 = 7) \]
\[ f_7: (pc_7 = 7) \land (s_7 \geq 10) \land (pc_8 = 8) \]
\[ f_8: (pc_8 = 8) \land (i_9 = i_8) \land (s_9 = s_8 + i_8) \land (pc_9 = 9) \]
\[ f_9: (pc_9 = 9) \land (i_{10} = i_9) \land (s_{10} = s_9) \land (pc_{10} = 10) \]
\[ f_{10}: (pc_{10} = 10) \land ((i_{10} > 0) \land (i_{11} = i_{10} - 1)) \lor ((i_{10} \leq 0) \land (i_{11} = i_{10})) \land (s_{11} = s_{10}) \land (pc_{11} = 11) \]
\[ f_{11}: (pc_{11} = 11) \land (s_{11} \geq 10) \land (pc_{12} = 12) \]
• 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

\[ X_3 X_2 X_1 X_0 \]

... \[ \rightarrow \] \[ A \rightarrow B \]

\[ \text{adder} \]

\[ C_{in} \rightarrow C_{out} \]

\[ C_0 = 0 \]

\[ X_3 X_2 X_1 X_0 \]

... \[ \leftarrow \] \[ A \leftarrow B \]
HW implementation of addition operation (1-bit):

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

\[
\begin{align*}
((A \land B \land C_{in}) & \implies (S \land C_{out})) \land \\
((\neg A \land B \land C_{in}) & \implies (\neg S \land C_{out})) \land \\
((A \land \neg B \land C_{in}) & \implies (\neg S \land C_{out})) \land \\
((A \land B \land \neg C_{in}) & \implies (\neg S \land C_{out})) \land \\
((\neg A \land \neg B \land C_{in}) & \implies (S \land \neg C_{out})) \land \\
((\neg A \land B \land \neg C_{in}) & \implies (S \land \neg C_{out})) \land \\
((A \land \neg B \land \neg C_{in}) & \implies (S \land \neg C_{out})) \land \\
((\neg A \land \neg B \land \neg C_{in}) & \implies (\neg S \land \neg C_{out}))
\end{align*}
\]

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C_{in}</td>
<td>S</td>
<td>C_{out}</td>
</tr>
<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 $S$

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

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

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

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

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

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

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

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

\[
S_0 = A_0 \text{xor} B_0
\]
\[
C_{\text{out},0} = A_0 \text{ and } B_0
\]
\[
S_1 = A_1 \text{xor} B_1 \text{xor} C_{\text{out},0}
\]
\[
C_{\text{out},1} = (A_1 \text{xor} B_1) \text{ and } C_{\text{out},0} \text{ or } (A_1 \text{ and } B_1)
\]
\[
S_2 = A_2 \text{xor} B_2 \text{xor} C_{\text{out},1}
\]
\[
C_{\text{out},2} = (A_2 \text{xor} B_2) \text{ and } C_{\text{out},1} \text{ or } (A_2 \text{ and } B_2)
\]
\[
S_3 = A_3 \text{xor} B_3 \text{xor} C_{\text{out},2}
\]
\[
C_{\text{out},3} = (A_3 \text{xor} B_3) \text{ and } C_{\text{out},2} \text{ or } (A_3 \text{ and } B_3)
\]
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. For example, 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.
Bounds are not that limiting...

- 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
• 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 $⟨g⟩M⟨f⟩$ 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 $⟨g⟩M'⟨f⟩$ and $⟨true⟩M⟨g⟩$
  - then we have $⟨true⟩M \parallel M'⟨f⟩$

- 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:

\[
\frac{\langle f \rangle M \langle g \rangle}{\frac{\langle g \rangle M' \langle f \rangle}{M \parallel M' \models f \land g}}
\]

• 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,...)
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
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);
    }
}

We need some guarantees from swapMin to prove this
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);
}
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
NAIL094

Summer semester 2020/2021

Inside into algorithms and techniques inside SAT (and SMT) solvers
• NSWI132
• Taught in Summer semester 2020/2021
• Insight into code verification tools
  ▪ Focus on practical experience with the code verifiers