Model Checking & Program Analysis

Markus Müller-Olm
Dortmund University

Overview

- Introduction
- Model Checking
- Flow Analysis
- Some Links between MC and FA
- Conclusion

Apology for not giving proper credit to other researchers' work in this tutorial!
Purposes of Automatic Analysis

- Optimizing compilation
- Validation/Verification
  - Type checking
  - Functional correctness
  - Security properties
  - ...
- Debugging

Fundamental Limit

Rice's Theorem [Rice, 1953]:
All interesting semantic questions about programs from a universal programming language are undecidable.
Example: Detection of Constants

\[
\text{read(new);} \quad ? \quad \text{read(new)}; \\
\pi; \quad \pi; \\
\text{write(new)} \quad \text{write(k)}
\]

write(new) can be replaced by write(k) for some constant k

\[\iff \pi \text{ terminates}\]

Hence: Constant Detection is undecidable

Two Solutions

Weaker formalisms
- analyze abstract models of systems
- e.g.: automata, labelled transition systems,...

Approximate analyses
- yield sound but, in general, incomplete results
- e.g.: detects some instead of all constants

Model checking
Flow analysis
Abstract interpretation
Type checking
Weaker Formalisms

Overview

- Introduction
- **Model Checking**
- Flow Analysis
- Some Links between MC and FA
- Conclusion
Model Checking

Finite-state structure
G(φ → Fψ)
Model Checker
OK
Error trace
or

Temporal logic formula

What is a Model Checker?

an automatic procedure for deciding

\[ M \models \phi \]

where
• \( M \) is a model structure
• \( \phi \) is a (temporal-logic) formula
• \( \models \) means satisfaction
Model Structures

\[ M \models \phi \]

Kripke structure

Labeled Transition System

Kripke Transition System
Kripke Structures

$K = (S, R, I)$, where

- $S$ states
- $R \subseteq S \times S$ transition relation, total
- $I : S \rightarrow 2^{AP}$ interpretation
- $AP$ atomic propositions

Temporal Logics

$M \models \phi$
Temporal Logics

- Linear-Time Logics
  - formulas specify properties of program paths
  - state satisfies property if all paths starting in this state do
- Branching-Time Logic
  - can specify properties sensitive to branching
  - has (or can simulate) path quantifiers A and E

Branching vs. Linear-Time Logics

- LT logics cannot distinguish P and Q:
  - P and Q have same execution paths
- BT logics can:
  - \( P \models [\text{coin}] <\text{tea}> \) true but \( Q \not\models [\text{coin}] <\text{tea}> \) true
A Linear-Time Logic: PLTL

1. own the resource before using it
   \[ \lnot \text{use} \cup \text{own} \]
2. release the resource in finite time
   \[ \text{own} \Rightarrow F(\lnot \text{own}) \]

PLTL formulas
\[ \phi := p \mid \lnot \phi \mid \phi_1 \lor \phi_2 \]
\[ X\phi \mid \phi U \phi_2 \]

Abbreviations
\[ F\phi := \text{true} U \phi \]
\[ G\phi := \lnot F(\lnot \phi) \]
\[ \phi WU \psi := \phi U \psi \lor G\phi \]

Linear-Time Modalities

\[ X\phi : \quad \bullet \quad \bullet \quad \bullet \quad \bullet \quad \bullet \quad \cdots \quad \text{neXt} \]
\[ G\phi : \quad \phi \quad \phi \quad \phi \quad \phi \quad \phi \quad \cdots \quad \text{Generally} \]
\[ F\phi : \quad \bullet \quad \bullet \quad \bullet \quad \bullet \quad \phi \quad \cdots \quad \text{Finally} \]
\[ \phi U \psi : \quad \phi \quad \phi \quad \phi \quad \psi \quad \cdots \quad \text{Until} \]
\[ \phi WU \psi : \quad \phi \quad U \psi \quad \text{or} \quad G\phi \quad \text{Weak Until} \]
**Until**

- Linear-time: blue nodes satisfy 
  \[ v \models \phi \mathbf{U} \psi \]

- Branching-time: blue nodes satisfy 
  \[ v \models \mathbf{A}(\phi \mathbf{U} \psi) \]

**Weak Until**

- Linear-time: blue nodes satisfy 
  \[ v \models \phi \mathbf{WU} \psi \]

- Branching-time: blue nodes satisfy 
  \[ v \models \mathbf{A}(\phi \mathbf{WU} \psi) \]
A Branching-Time Logic: CTL

1. **own the resource before using it**
   \[
   E(\neg \text{use} \mathrel{U} \text{own})
   \]

2. **release the resource in finite time**
   \[
   \text{own} \Rightarrow AF(\neg \text{own})
   \]

**State formulas** $\phi$

\[
\phi ::= p \mid \neg \phi \mid \phi_1 \lor \phi_2 \mid E\psi \mid A\psi
\]

**Path formulas** $\psi$

\[
\psi ::= X\phi \mid \phi_1 U \phi_2 \mid G\phi \mid F\phi
\]

Duality

- **Path quantifiers are duals:**
  - $\neg A \phi = E \neg \phi$
  - $\neg E \phi = A \neg \phi$
- **Until and weak until are almost duals (on paths):**
  - $\neg (\phi \mathrel{U} \psi) = \neg \psi \mathrel{WU} (\neg \phi \land \neg \psi)$
  - $\neg (\phi \mathrel{WU} \psi) = \neg \psi \mathrel{U} (\neg \phi \land \neg \psi)$
Modal mu-Calculus

- a branching-time logic with **local** modalities...
  - $\Box \phi$: all successor states satisfy $\phi$
  - $\Diamond \phi$: some successor state satisfies $\phi$
- ... and fixpoint formulae.
  - $\mu X. \phi(X)$: minimum fixpoint formula
  - $\nu X. \phi(X)$: maximum fixpoint formula
- Fixpoint formulae can be nested
  - Alternation: $\mu X.(\nu Y.X \land Y)$

Local Modalities for Labelled Edges

- $[a] \phi$: $\phi$ is true in all successor states
- $<a> \phi$: $\phi$ is true in some successor state
Computing Fixpoint Formulae

- On finite structures, meaning of fixpoint formulae can be computed by computing Kleene chains until stabilization:
  - $\mu: \bot, f(\bot), f(f(\bot)), f(f(f(\bot))), \ldots$
  - $\nu: T, f(T), f(f(T)), f(f(f(T))), \ldots$

  $f$ is a function on state sets derived from the body of the fixpoint formula.

CTL and Modal mu-Calculus

- CTL-formulae can inductively be transformed to modal mu-calculus formulae
- Global modalities can be expressed by fixpoint formulae, e.g.:
  \[
  A(\phi U \psi) = \mu X.(\psi \lor (\phi \land X \land \Diamond \text{true}))
  \]
  \[
  E(\phi U \psi) = \mu X.(\psi \lor (\phi \land \Diamond X))
  \]
  \[
  A(\phi WU \psi) = \nu X.(\psi \lor (\phi \land \Box X))
  \]
  \[
  E(\phi WU \psi) = \nu X.(\psi \lor (\phi \land (\Diamond X \lor \Box \text{false})))
  \]
Model Checking Approaches

\[ M \vDash \phi \]

Global vs. Local Model Checking

- **Global model checking problem**
  - Given: finite model structure \( M \), formula \( \phi \)
  - Determine: \( \{ s \mid s \vDash \phi \} \)

- **Local model checking problem**
  - Given: finite model structure \( M \), formula \( \phi \), and state \( s \) in \( M \)
  - Determine, whether \( s \vDash \phi \) or not
Model Checking Approaches

- Iterative model checking
- Automata-theoretic model checking
- Tableau-based model checking

Iterative Model Checking

- Good for global model-checking of branching-time logics
- Idea: Compute semantics of formula on given model by structural induction on the formula
  - Reduce modalities to their fixpoint definition
  - Compute the fixpoints by Kleene chains
  - Alternating fixpoints lead to backtracking

(→ proceedings)
Iterative Model Checking of CTL

- Annotate each state with those subformulas $\psi$ of $\phi$ with $s \models \psi$.
- Use structural induction on $\phi$.
- Use backwards propagation to compute modalities $A(\phi U \psi)$ and $A(\phi WU \psi)$.

Model Checking $A(\phi U \psi)$

1. Mark all nodes $v$ with $v \models \psi$
2. Mark all unmarked nodes $w$ with $w \models \phi$ and all successors marked
3. Iterate 2. until stabilization
Model Checking $A(\phi WU \psi)$

1. Mark all nodes with $v \models \phi$ or $v \models \psi$
2. Unmark all nodes $v$ with $v \not\models \psi$ and some unmarked successor
3. Iterate 2. until stabilization

Automata-theoretic MC

- Good for linear-time logics
- Idea: reduce model-checking to non-emptiness problem of an automaton
Automata-theoretic MC

- Construct (Büchi-) automaton, $A_\phi$, from $\phi$
  - $A_\phi$ accepts paths satisfying $\phi$
- Construct (Büchi-) automaton, $A_M$, from $M$
  - $A_M$ accepts paths exhibited by $M$

$$M \models \phi \iff L(A_M) \subseteq L(A_\phi)$$
$$\iff L(A_M) \cap \overline{L(A_\phi)} = \emptyset$$
$$\iff L(A_M \times \overline{A_\phi}) = \emptyset$$

Automata-theoretic MC

- Construct (Büchi-) automaton, $A_\phi$, from $\phi$
  - $A_\phi$ accepts paths satisfying $\phi$
- Construct (Büchi-) automaton, $A_M$, from $M$
  - $A_M$ accepts paths exhibited by $M$
- Compute automaton $A_M \times \overline{A_\phi}$
  - Complementation & product construction
- Decide $L(A_M \times \overline{A_\phi}) = \emptyset$ by reachability analysis
Automaton for $F(P) \land G(Q)$
(Finite Maximal Paths Interpretation)

[A Successful Model-Check for formula $F(P) \land G(Q)$]

Kripke structure:

Corresponding automaton:

Product automaton:
A Failing Model-Check for formula $F(P) \land G(Q)$

Tableau-Based Model-Checking

- Idea: solve local model checking problem by sub-goaling
  - Try to construct a proof tree that witnesses $s \models \phi$
  - If no proof tree can be found, then $s \not\models \phi$
  - For proof tree construction, tableau rules are given that reduce goals to sub-goals
Some Tableau Rules

\[
\begin{align*}
\text{AND} & : \frac{s \vdash \phi_1 \land \phi_2}{s \vdash \phi_1, s \vdash \phi_2} \\
\text{OR1} & : \frac{s \vdash \phi_1 \lor \phi_2}{s \vdash \phi_1} \\
\text{OR2} & : \frac{s \vdash \phi_1 \lor \phi_2}{s \vdash \phi_2} \\
\text{BOX} & : \frac{s \vdash [a] \phi}{s_1 \vdash \phi, \ldots, s_n \vdash \phi} \quad \text{if } \{s_1, \ldots, s_n\} = \{t \mid s \xrightarrow{a} t\} \\
\text{DIAMOND} & : \frac{s \vdash \langle a \rangle \phi}{t \vdash \phi} \quad \text{if } s \xrightarrow{a} t
\end{align*}
\]

Tableau Rules for Fixpoints

- Fixpoint formulas are analyzed with unfolding rules, e.g.,

\[
\mu\text{-UNFOLD} \quad \frac{s \vdash \mu X. \phi(X)}{s \vdash \phi(\mu X. \phi(X))}
\]

- If fixpoint formula sequent re-appears later:
  - \(\mu\) : unsuccessful leaf
  - \(\nu\) : successful leaf
- Special care needed for nested fixpoints
  \(\rightarrow\) proceedings
A Model Check by Tableau

\[ \phi = \nu X.(\psi \land [b]X) \]
\[ \psi = \mu X.(a)\text{true} \lor (b)X \]

\[ s \vdash \phi \]
\[ s \vdash \psi \land [b]\phi \]
\[ s \vdash [b]\phi \]
\[ s \vdash (a)\text{true} \lor (b)\psi \]
\[ s \vdash (a)\text{true} \]
\[ t \vdash \text{true} \]

Typical Profile of Model Checking Techniques

<table>
<thead>
<tr>
<th></th>
<th>Branching-time</th>
<th>Linear-time</th>
<th>Global</th>
<th>Local</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterative</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Automata-theoretic</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Tableau methods</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>
State-Explosion Problem

- State-Explosion
  - number of states grows exponentially in number of components or variables
- Techniques for fighting state-explosion
  - symbolic model-checking
  - incremental construction of state space
  - abstraction
  - symmetry reductions
  - partial-order methods
  - compositional methods
  - ...

Other Classes of Systems

- Infinite-state systems
- Timed systems
- Hybrid systems
- Probabilistic systems
- ...

Learn more about such systems this week!
Overview

- Introduction
- Model Checking
- Flow Analysis
- Some Links between MC and FA
- Conclusion

From Programs to Flow Graphs

```plaintext
main()
{ x=17;
  if (x>63)
    { y=17; x=10; x=x+1; }
  else
    { x=42;
      while (y<99)
        { y=x+y; x=y+1; }
      y=11;
      x=y+1;
      out(x);
  }
}
```
Dead Code Elimination

- **Goal:**
  - find and eliminate assignments that compute values which are never used
- **Fundamental problem:**
  - undecidability
  - hence: use approximate algorithm; ignore guards
- **Technique:**
  - propagate set of non-needed variables backwards through the flow graph
Remarks

- Forward vs. backward analyses
- Bitvector analyses
  - backward: live/dead variables, very busy expressions,
  - forward: reaching definitions, available expressions
- Computation strategies
Flow Equations for Dead Code

\[ x := y + z \]

\[ \text{DeadIn}[i] = (\text{DeadOut}[i] \cup \{x\} \setminus \{y, z\}) \]

\[ \text{DeadOut}[i] = \bigcap_{j \in \text{Succ}(i)} \text{DeadIn}[i] \]

General equations:

\[ \text{DeadIn}[i] = (\text{DeadOut}[i] \cup \text{Mod}[i]) \setminus \text{Use}[i] \]

\[ \text{DeadOut}[i] = \bigcap_{j \in \text{Succ}(i)} \text{DeadIn}[i] \]

Equations may be combined…

\[ \text{DeadOut}[i] = \bigcap_{j \in \text{Succ}(i)} (\text{DeadOut}[j] \cup \text{Mod}[j]) \setminus \text{Use}[j] \]

…or replaced by inequations:

\[ \text{DeadOut}[i] \subseteq (\text{DeadOut}[j] \cup \text{Mod}[j]) \setminus \text{Use}[j], \text{ for } j \in \text{Succ}(i) \]

Data-Flow Frameworks

- Correctness
  - generic properties of frameworks can be studied and proved

- Implementation
  - efficient, generic implementations can be constructed
**Data-Flow Frameworks**

- a complete lattice \((D, \sqsubseteq)\) of data-flow facts
- a space \(F\) of transfer functions \(f: D \to D\)
  - \(\text{Id} \in F\)
  - closed under composition
- initial value \(\text{init} \in D\)
- a control-flow graph with set of entry/exit nodes
- mapping assigning transfer functions \(f_i\) to flow graph nodes \(i\)

**Framework for dead variables**

<table>
<thead>
<tr>
<th>lattice (D)</th>
<th>(2^{\text{Var}})</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\sqsubseteq)</td>
<td>(\subseteq)</td>
</tr>
<tr>
<td>(\cap)</td>
<td>(\cap)</td>
</tr>
<tr>
<td>(\top)</td>
<td>(\text{Var})</td>
</tr>
</tbody>
</table>

control flow program graph

initial value \(\emptyset\)

function space \(\{f : D \to D | \exists d_1, d_2; f(d) = (d \cup d_1) \setminus d_2\}\)

\(f_i\)

\(f_i(d) = (d \cup \text{Mod}[i]) \setminus \text{Use}[i]\)
What Data Flow Algorithms Compute

- **Forward Analysis**
  - Computes maximal solution w.r.t. \((D, \sqsubset)\) of
    \[
    \text{In}[i] = \begin{cases} 
    \text{init} & i \in \text{Entry} \\
    \bigcap_{j \in \text{Pred}(i)} f_j(\text{In}[j]) & \text{otherwise}
    \end{cases}
    \]

- **Backward Analysis**
  - Computes maximal solution of
    \[
    \text{Out}[i] = \begin{cases} 
    \text{init} & i \in \text{Exit} \\
    \bigcap_{j \in \text{Succ}(i)} f_j(\text{Out}[j]) & \text{otherwise}
    \end{cases}
    \]
  - This is called the maximal fixpoint solution \(\text{MFP}[i]\)

Assessing Data Flow Frameworks

- **Execution Semantics**
- **Abstraction**
- **MOP-solution**
- **MFP-solution**

- sound? how precise?
- sound? precise?
Meet-Over-All-Paths Solution

- Forward Analysis
  \[ \text{MOP}[i] := \bigsqcap_{p \in \text{Paths}(\text{entry},i)} F_p(\text{init}) \]

- Backward Analysis
  \[ \text{MOP}[i] := \bigsqcap_{p \in \text{Paths}(i,\text{exit})} F_p(\text{init}) \]
Coincidence Theorem

Definition:
A framework is distributive if
\[ f(\sqcap X) = \sqcap \{ f(x) \mid x \in X \} \]
for all \( X \subseteq D, f \in F \).

Theorem:
In any distributive framework,
\( MOP[i] = MFP[i] \) for all program points \( i \).

Theorem:
All bitvector frameworks are distributive.

---

Lattice for Constant Propagation

\[ \begin{array}{cccccc}
\top & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \bot \\
\ldots & -2 & -1 & 0 & 1 & 2 & \ldots \\
\leftarrow & \leftarrow & \leftarrow & \leftarrow & \leftarrow & \\
\bot & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \bot \\
\end{array} \]

inconsistent value
unknown value
Constant Propagation Framework

lattice $D$ \quad Var \to (\mathbb{Z} \cup \{\bot, \top\}) = Var \to ConstVal \\
\sqsubseteq \quad \rho \sqsubseteq \rho' :\Rightarrow \forall x : \rho(x) \sqsubseteq \rho'(x) \\
\sqcap \quad \text{pointwise meet} \\
\top \quad \top(x) = \top \quad \text{f.a. } x \in \text{Var} \\
control flow \quad \text{program graph} \\
initial value \quad \bot \\
function space \quad \{f : D \to D \mid f \text{ monotone}\} \\
\text{if } i \text{ annotated with } x := e \\
\text{otherwise} \\

\begin{align*}
\rho(x), \rho(y), \rho(z) \\
(2,3,5) \quad \text{out}(x) \\
(3,2,5) \quad \text{MOP}[v] = (\bot, \bot, 5)
\end{align*}
**Correctness Theorem**

**Definition:**
A framework is monotone if $f(x) \sqsubseteq f(y)$ for all $x, y \in D$, $x \sqsubseteq y$.

**Theorem:**
In any monotone framework, $\text{MFP}[i] \sqsubseteq \text{MOP}[i]$ for all program points $i$.

**Remark:**
Any "reasonable" framework is monotone.
Assessing Data Flow Frameworks

Where Flow Analysis Loses Precision
Overview

- Introduction
- Model Checking
- Flow Analysis
- Some Links between MC and FA
- Conclusion

Some Links between MC and FA

- Flow Analysis via Model Checking
- Model Checking via Flow Analysis
- Synergy of MC and FA
Flow Analysis via Model Checking

- yes/no data-flow problem
  - flow graph
    - Flow graph transformer
      - formula
        - Kripke structure
          - Model-Checker
            - yes/no annotation of Kripke structure nodes
              - Interpretation
                - data flow information for flow graph nodes

Dead-Code Elimination

- Flow Graph Transformation:
  - \( y := x+y \) to \( \text{Mod}(y), \text{Use}(x), \text{Use}(y) \)

- Formula specifying "x is dead":
  \[
  \text{Dead}_x = X (\neg \text{use}(x) \text{ WU } (\text{mod}(x) \land \neg \text{use}(x)))
  \]

- Note: more direct specification than in the data flow framework
Check: $\text{Dead}_x = X (\neg \text{use}(x) \text{ WU } \text{mod}(x) \land \neg \text{use}(x))$

Model Checking via Flow Analysis

- Evaluation of (CTL) modalities can be seen as a data flow analysis
- CTL model-checking is iterated flow analysis
Framework for $E(\phi \mathbf{U} \psi)$

- lattice $D = \{0, 1\}$
  - $\bot 1 \sqsubseteq 0$
  - $\sqcap 0 \sqcap 1 = 1$
  - $\top 0$

- control flow Kripke structure $(S, R, I)$
- initial nodes Nodes $s$ with $s \models \psi$
- initial value 1
- function space $\{\lambda x.0, \lambda x.1, \lambda x.x\}$

$$f_i, f_i(d) = \begin{cases} 
\lambda x.x & \text{if } i \not\models \psi, i \models \phi \\
\lambda x.0 & \text{if } i \not\models \psi, i \not\models \phi
\end{cases}$$

---

Why the Framework computes $E(\phi \mathbf{U} \psi)$

$\text{MFP}[^{\downarrow}v] = \text{MOP}[^{\downarrow}v] = 1$ iff $s \models E(\phi \mathbf{U} \psi)$

$\begin{align*}
\phi, \neg \psi & \phi, \neg \psi & \phi, \neg \psi & \psi & 0 \\
\lambda x.x & \lambda x.x & \lambda x.x & 1
\end{align*}$
The Model-Extraction Problem in Software Model-Checking

Program

Abstract model

Model Checker

Idea: Use techniques from abstract interpretation and program analysis to extract a finite-state model from program
Bandera: [Dwyer, Hatcliff, et. al.]
An open tool set for model-checking Java source code

Conclusion

- Overview on fundamentals of
  - Model Checking
  - Flow Analysis
  - Some links
Just the Beginning. . .

- More complex properties
- Theory of Abstract Interpretation
- Interprocedural Flow Analysis
- Parallel Programs

Any Questions?