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

\[\Leftrightarrow\]
\[\pi\] 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

Program

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

Abstract model

Exact analyzer for abstract model

Approximate

Exact

Overview

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

Finite-state structure

G(Φ → FΨ)

Temporal logic formula

Model Checker

OK

or

Error trace

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

\[ M \models \phi \]

Model Structures

- 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
- \( I : S \to 2^{AP} \) 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**:
  - $P \models [\text{coin}] <\text{tea}> \text{true}$ but $Q \not\models [\text{coin}] <\text{tea}> \text{true}$
A Linear-Time Logic: PLTL

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

Abbreviations
\[ F \phi ::= true U \phi \]
\[ G \phi ::= \neg F (\neg \phi) \]
\[ \phi WU \psi ::= \phi U \psi \lor G \phi \]

Linear-Time Modalities

\[ X \phi::= \phi U \psi::= \phi \lor G \phi \]

Abbreviations:
\[ F \phi ::= true U \phi \]
\[ G \phi ::= \neg F (\neg \phi) \]
\[ \phi WU \psi ::= \phi U \psi \lor G \phi \]
**Until**

- Linear-time: blue nodes satisfy \( v \models \phi U \psi \)
- Branching-time: blue nodes satisfy \( v \models A(\phi U \psi) \)

---

**Weak Until**

- Linear-time: blue nodes satisfy \( v \models \phi WU \psi \)
- Branching-time: blue nodes satisfy \( v \models A(\phi WU \psi) \)
A Branching-Time Logic: CTL

State formulas $\phi$

$$\phi ::= p | \neg \phi | \phi_1 \lor \phi_2 | E\psi \lor A\psi$$

Path formulas $\psi$

$$\psi ::= X \phi | \phi U \phi_2 | G\phi \lor F\phi$$

1. own the resource before using it

$$A(\neg \text{use } U \text{ own})$$

2. release the resource in finite time

$$\text{own } \Rightarrow AF(\neg \text{own})$$

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):
  - $\neg (\phi U \psi) = \neg \psi WU (\neg \phi \land \neg \psi)$
  - $\neg (\phi WU \psi) = \neg \psi U (\neg \phi \land \neg \psi)$
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

- $[a] \phi$:
- $<a> \phi$:
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
  - $\nu : \top, f(\top), f(f(\top), f(f(f(\top))))$, \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.:

  $A(\phi U \psi) = \mu X. (\psi \lor (\phi \land X \land \Diamond \text{true}))$
  $E(\phi U \psi) = \mu X. (\psi \lor (\phi \land \Diamond X))$
  $A(\phi WU \psi) = \nu X. (\psi \lor (\phi \land X \land \Box \text{false}))$
  $E(\phi WU \psi) = \nu X. (\psi \lor (\phi \land (\Diamond X \lor \Box \text{false})))$
Model Checking Approaches

\[ M \models \varphi \]

Global vs. Local Model Checking

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

- **Local** model checking problem
  - Given: finite model structure \( M \), formula \( \varphi \), and state \( s \) in \( M \)
  - Determine, whether \( s \models \varphi \) 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 $A(\phi \text{WU} \psi)$

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 $M$

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

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

\[
\text{AND} \quad \frac{s \vdash \phi_1 \land \phi_2}{s \vdash \phi_1 \quad s \vdash \phi_2}
\]

\[
\text{OR1} \quad \frac{s \vdash \phi_1 \lor \phi_2}{s \vdash \phi_1} \quad \quad \text{OR2} \quad \frac{s \vdash \phi_1 \lor \phi_2}{s \vdash \phi_2}
\]

\[
\text{BOX} \quad \frac{s \vdash [a] \phi}{s \vdash \phi_1, \ldots, s_n \vdash \phi} \quad \text{if} \quad \{s_1, \ldots, s_n\} = \{t \mid s \xrightarrow{a} t\}
\]

\[
\text{DIAMOND} \quad \frac{s \vdash \langle a \rangle \phi}{t \vdash \phi} \quad \text{if} \quad s \xrightarrow{a} t
\]

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
  (→ proceedings)
A Model Check by Tableau

\[ \phi = \nu X. (\psi \land [b] X) \]

\[ \psi = \mu X. \langle a \rangle \text{true} \lor \langle b \rangle X \]

\[ s \vdash \phi \]
\[ s \vdash \psi \land [b] \phi \]
\[ s \vdash \psi \land [b] \phi \]
\[ s \vdash [b] \phi \]
\[ s \vdash \phi \]

\[ \nu \Rightarrow \text{successful} ! \]

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></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

```c
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:
\[ \text{DeadIn}[i] = (\text{DeadOut}[i] \cup \text{Mod}[i]) \setminus \text{Use}[i] \]
\[ \text{DeadOut}[i] = \bigcap_{j \in \text{dead}(i)} \text{DeadIn}[i] \]

Equations may be combined…
\[ \text{DeadOut}[i] = \bigcap_{j \in \text{dead}(i)} (\text{DeadOut}[j] \cup \text{Mod}[j]) \setminus \text{Use}[j] \]

…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 \(f: D \rightarrow D\)
  - \(\text{Id} \in F\)
  - 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>(\sqcap)</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 \(\{f: D \rightarrow D \mid \exists d_1, d_2: f(d) = (d \cup d_1) \setminus d_2\}\)

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

- **Forward Analysis**
  - Computes maximal solution w.r.t. \((D, \sqsubseteq)\) of
    \[
    \text{In}[i] = \begin{cases} 
    \text{init} & i \in \text{Entry} \\
    \mathcal{P} \text{\text{\text{\text{Pre}}} } (i) \text{ } f_i (\text{In}[j]) & \text{otherwise}
    \end{cases}
    \]

- **Backward Analysis**
  - Computes maximal solution of
    \[
    \text{Out}[i] = \begin{cases} 
    \text{init} & i \in \text{Exit} \\
    \mathcal{P} \text{\text{\text{\text{Succ}}} } (i) \text{ } f_i (\text{Out}[j]) & \text{otherwise}
    \end{cases}
    \]

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

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

\[ \text{MOP}[v] = \{x, y\} \cap \{x\} = \{x\} \]
Coincidence Theorem

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

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

Theorem:
All bitvector frameworks are distributive.

---

Lattice for Constant Propagation

\[
\begin{array}{c}
\top \\
\downarrow \\
-2 & -1 & 0 & 1 & 2 & \ldots \\
\downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
\inconsistent & \inconsistent & \inconsistent & \inconsistent & \inconsistent \\
\end{array}
\]

unknown value
Constant Propagation Framework

lattice $D$ Var $\to \mathbb{Z} \cup \{\bot, \top\} = \text{Var} \to \text{ConstVal}$

$\sqsubseteq \rho \sqsubseteq \rho' : \forall x : \rho(x) \sqsubseteq \rho'(x)$

$\sqcap$ pointwise meet

$\top \top (x) = \top \text{ f.a. } x \in \text{Var}$

control flow program graph

initial value $\bot$

function space $\{f : D \to D \mid f \text{ monotone}\}$

$f_i$ $f_i(d) = \begin{cases} d[x \mapsto \llbracket e \rrbracket^{CP}(d)] & \text{if } i \text{ annotated with } x := e \\ d & \text{otherwise} \end{cases}$

$\{(\rho(x), \rho(y), \rho(z))\}$

$(x := 2, y := 3, z := x + y)$

$\text{MOP}[v] = (\bot, \bot, 5)$
Correctness Theorem

Definition:
A framework is monotone if f(x) ≤ f(y) for all x, y ∈ D, x ≤ y.

Theorem:
In any monotone framework, MFP[i] ⊆ MOP[i] for all program points i.

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

Where Flow Analysis Looses 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

Design & Programming

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:

  \[ y := x + y \]

- Formula specifying "x is dead":

  \[ \text{Dead}_x = X (\neg \text{use}(x) \text{ WU } (\text{mod}(x) \wedge \neg \text{use}(x))) \]

- Note: more direct specification than in the data flow framework
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 $\text{E}(\phi \text{ U } \psi)$

lattice $D$

$\mathbb{B} = \{0, 1\}$

$\subseteq$

$1 \subseteq 0$

$\sqcap$

$0 \sqcap 1 = 1$

$\top$

$0$

control flow

Kripke structure $(S, R, I)$

initial nodes

Nodes $s$ with $s \models \psi$

initial value

1

function space

$\{\lambda x.0, \lambda x.1, \lambda x.x\}$

$f_i$

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

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

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

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
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?