Model Checking for Software

Willem Visser
Charles Pecheur

RIACS / NASA Ames

{wvisser,pecheur}@ptolemy.arc.nasa.gov
Part I
Overview

• Explicit State Model Checking
  – What is model checking?
  – Kripke structures
    • Describing the systems we want to check
  – Temporal logic
    • Describing the properties we want to check
  – Automata-theoretic model checking
  – State-explosion problem
    • What can we do?

• Model Checking Programs
  – A brief history of the field
  – Java PathFinder
Model Checking
The Intuition

- Calculate whether a system satisfies a certain behavioral property:
  - Is the system deadlock free?
  - Whenever a packet is sent will it eventually be received?
- Testing?
  - Look at all possible behaviors of a system
- Automatic, if the system is finite-state
  - Potential for being a push-button technology
  - Almost no expert knowledge required
- How do we describe the system?
- How do we express the properties?
Kripke Structures

- \( K = (\text{props}, S, R, s_0, L) \)
  - \( \text{props} \): (finite) set of atomic propositions
  - \( S \): (finite) set of states
  - \( R \): binary transitive relation (total)
  - \( s_0 \): set of initial states
  - \( L \): maps each state to the set of propositions true in the state

- Often \( M = (S, R, L) \) with \( \text{props} \) and \( s_0 \) implicit
Example Kripke Structure

\[ K = (\{ p, \neg p \}, \{ x, y, z, k, h \}, R, \{ x \}, L) \]
Property Specifications

• Temporal Logic
  – Express properties of event orderings in time

• Linear Time
  – Every moment has a unique successor
  – Infinite sequences (words)
  – Linear Time Temporal Logic (LTL)

• Branching Time
  – Every moment has several successors
  – Infinite tree
  – Computation Tree Logic (CTL)
CTL*

\[ S ::= \text{true} \mid \text{false} \mid q \mid \neg q \mid S \lor S \mid S \land S \mid AP \mid EP \]
\[ P ::= S \mid P \lor P \mid P \land P \mid XP \mid P \mid P \mid P \]

- \( A \) (for all) and \( E \) (there exists) are path quantifiers
- \( X \) (until), \( U \) (until) and \( V \) (release) are path operators
- CTL: Every path operator is preceded by a path quantifier
- LTL: Path formulas of the form \( AP \) where the state sub-formulas are atomic propositions
- \( Fp = \text{true} \ U p \) - “eventually \( p \)” or “finally \( p \)”
- \( Gp = \text{false} \ V p \) - “always \( p \)” or “globally \( p \)”
Computation Tree Logic (branching):
Consider the tree of possible executions

Always ...

\( \text{AX} \ p \)

\( \text{AG} \ p \) (In all states)

Sometimes ...

\( \text{EX} \ p \)

\( \text{EG} \ p \) (In some state)

... Next \( p \) ...

... Globally \( p \) ...

... Finally \( p \) ...

... \( p \) Until \( q \) ...

\( \text{AF} \ p \)

\( \text{EF} \ p \) (duals)

\( \text{AG} \ p \)

\( \text{EG} \ p \)

\( \text{AF} \ p \)

\( \text{EF} \ p \) (duals)

\( \text{AG} \ p \)

\( \text{EG} \ p \)
Mutual Exclusion Example

- Two process mutual exclusion with shared semaphore
- Each process has three states
  - Non-critical (N)
  - Trying (T)
  - Critical (C)
- Semaphore can be available ($S_0$) or taken ($S_1$)
- Initially both processes are in the Non-critical state and the semaphore is available --- $N_1 N_2 S_0$

\[
\begin{align*}
N_1 \quad \rightarrow \quad T_1 \\
T_1 \land S_0 \quad \rightarrow \quad C_1 \land S_1 \\
C_1 \quad \rightarrow \quad N_1 \land S_0 \\
N_2 \quad \rightarrow \quad T_2 \\
T_2 \land S_0 \quad \rightarrow \quad C_2 \land S_1 \\
C_2 \quad \rightarrow \quad N_2 \land S_0
\end{align*}
\]
Mutual Exclusion Example

- Mutual Exclusion: $K \_ AG \neg (C_1 \land C_2)$
- Response: $K \not/ AG (T_1 \rightarrow AF (C_1))$
- Reactive: $K \_ AG EF (N_1 \land N_2 \land S_0)$
Mutual Exclusion Example

K _ AG EF (N₁ ∧ N₂ ∧ S₀)
Mutual Exclusion Example

\[ K \_ AG \ EF (N_1 \land N_2 \land S_0) \]
Mutual Exclusion Example

\[ K \_ AG \ EF \left( N_1 \land N_2 \land S_0 \right) \]
Mutual Exclusion Example

\[ K \land AG EF (N_1 \land N_2 \land S_0) \]
Mutual Exclusion Example

\[ K \_ \text{AG } EF \left( N_1 \land N_2 \land S_0 \right) \]
Mutual Exclusion Example

\[ K \_ \text{AG EF } (N_1 \land N_2 \land S_0) \]
Mutual Exclusion Example

\[ K \not\models AG (T_1 \rightarrow AF (C_1)) \]
Mutual Exclusion Example

\[
K \not\models AG ( \neg T_1 \lor AF (C_1))
\]
Mutual Exclusion Example

\[ K \models AG ( \sim T_1 \lor AF (C_1)) \]
Model Checking

• Given a Kripke structure $M = (S,R,L)$ that represents a finite-state concurrent system and a temporal logic formula $f$ expressing some desired specification, find the set of states in $S$ that satisfy $f$:

$$\{ \ s \in S \mid M,s \models f \}$$

• Normally, some states of the concurrent system are designated as initial states. The system satisfies the specification provided all the initial states are in the set. We often write: $K \models f$
Model Checking Complexity

K _ f

• CTL
  – \( O(|K| \times |f|) \)
• LTL
  – \( O(|K| \times 2^{|f|}) \)
• But, for CTL the whole transition relation must be kept in memory!
  – Binary Decision Diagrams (BDDs) often allows the transition relation to be encoded efficiently
• The formulas are seldom very complex, hence \(|f|\) is not too troublesome.
Automata-Theoretic Model Checking

- Linear time temporal logic
  - Nondeterministic automata over infinite words
- Branching time temporal logic
  - Alternating automata over infinite trees
- Automata-theoretic LTL model checking
- Basic idea:
  - Translate both Kripke structure and LTL property into automata and show language containment
- See papers by Vardi and Wolper
Büchi Automata

• Accepts infinite words
• $B = (\Sigma, S, \rho, s_0, F)$
  – $\Sigma$ is a finite alphabet
  – $S$ is a finite set of states
  – $\rho : S \times \Sigma \rightarrow 2^S$ is the transition function
  – $s_0 \in S$ is the initial state (or states)
  – $F \subseteq S$ is the set of accepting states
• Given an infinite word $\omega = a_0, a_1, \ldots$ over $\Sigma$ then a run of $B$ is the sequence $s_0, s_1, \ldots$ where $s_{i+1} \in \rho(s_i, a_i)$
• Let $\inf(\pi)$ be the set of states that occur infinitely often on the run $\pi$, then $\pi$ is accepting iff $\inf(\pi) \cap F \neq \emptyset$
Example Büchi Automaton

\[ \begin{align*}
\text{Example Büchi Automaton} & \\
\begin{array}{c}
\begin{tikzpicture}
\node (1) [fill=red] at (0,0) {$1$};
\node (2) [fill=green] at (2,0) {$2$};
\draw [->] (1) to [loop above] node {$\sim p$} (1);
\draw [->] (1) to node {$p$} (2);
\draw [->] (2) to node {$p$} (1);
\draw [->] (2) to [loop above] node {$\sim p$} (2);
\end{tikzpicture}
\end{array} \\
\end{align*} \]

- \( B = (\{\{p\}, \{\sim p\}\}, \{1,2\}, \rho, 1, \{2\}) \)
  
- Example accepting words:
  - \((12)^\omega\)
  - \(1112^\omega\)

- Example rejecting word: \(121212111^\omega\)

- LTL property: GFp – “infinitely often p”
Kripke to Büchi Automaton

- \( K = (\text{props}, S, R, s_0, L) \) can be viewed as
- \( A_K = (2^{\text{props}}, S, \rho, s_0, S) \) where
  - \( s_{i+1} \in \rho(s_i, a) \) \iff \( (s_i, s_{i+1}) \in R \) and \( a = L(s) \)
  - Note every state is in the accepting set, hence all runs of the automaton is accepting
  - The language of the automaton, \( L(A_K) \), is the set of all behaviors of \( K \)
Kripke to Büchi Example
Kripke to Büchi Example
Translating LTL Formulas to Büchi Automata

- Exponential in the length of the formula
  - Many heuristics optimizations are used
  - Multitude of papers: CAV, LICS, etc.

\[
\begin{align*}
\text{p} & \quad \text{T} & \quad \text{q} \\
\text{p U q} & \\
\text{T} & \quad \text{T} & \quad \text{T} & \quad \text{p} & \quad \text{~p} \lor q
\end{align*}
\]

\[
\begin{align*}
\text{Fp} & \\
\text{Gp} & \quad \text{G(p \rightarrow Fq)}
\end{align*}
\]
Model Checking with Büchi Automata

• K \_ f
• Translate K and f to Büchi Automata
• Language containment
  - \( L(A_K) \subseteq \overline{L(A_f)} \)
  - \( L(A_K) \cap L(A_f) = \emptyset \)
  - \( L(\overline{A_f}) = L(A_{\neg f}) \) and \( L(A_K \times A_{\neg f}) = L(A_K) \cap L(A_{\neg f}) \)
• Algorithm
  - Negate formula f and create \( A_{\neg f} \)
  - Construct the product \( A_{K,\neg f} = K \times A_{\neg f} \)
  - If \( L(A_{K,\neg f}) = \emptyset \) report YES else report NO
Model Checking Example

- $K \_ AFG\neg p$
  - For all paths from some moment onwards $p$ is always false

- Where $K$ is given by

\[
\begin{align*}
  K & = x \oplus y \oplus z \\
  & = (x \land \neg p) \lor (y \land \neg p) \lor (z \land \neg p) \\
  & = k \lor \neg p \\
  & = y \lor \neg p \\
  & = h \lor \neg p \\
  & = z \lor \neg p
\end{align*}
\]
Step 1

- Negate FG~p
  - GFp

- Construct Büchi Automaton for GFp
Step 2

• Construct the product automaton
Step 3

- Check if the language is empty
- It is nonempty since there is a cycle through an accepting state, hence $K \not\models \text{AFG}\neg\phi$
  - $(xkhz)^0$ is an accepting run
- The accepting run is also a counter-example to the property being true
Checking Nonemptiness

- A Büchi automaton accepts some word iff there exists an accepting state reachable from the initial state and from itself.
- Can be checked in linear time.
- Model Checking complexity for LTL:
  - $O(|K| \times 2^{|f|})$
Efficient Nonemptiness Checking

Dfs (state s)
   Add (s,0) to VisitedStates;
   FOR each successor t of s DO
      IF (t,0) \notin VisitedStates THEN Dfs(t) END
   END
   IF s \notin F THEN seed := s; 2Dfs(s) END
END

2Dfs (state s)
   Add (s,1) to VisitedStates;
   FOR each successor t of s DO
      IF (t,1) \notin VisitedStates THEN 2Dfs(t) END
      ELSEIF t = seed THEN report nonempty END
   END
END

Efficiency

- VisitedStates as HashTable
- Change Recursion to Iteration
- Generate successor states on-the-fly
SPIN Model Checker

- Automata based model checker
  - Efficient nonemptiness algorithm
- Translates LTL formula to Büchi automaton
- Kripke structures are described as “programs” in the PROMELA language
  - Kripke structure is generated on-the-fly during nonemptiness checking
  - Relevant theoretical papers can be found here
State-Explosion?

- $n$ concurrent processes with $m$ states each
  - Has $m^n$ states
  - Worst-case, an on-the-fly model checker has to enumerate all of them

- What can we do to reduce $m^n$?
  - Reduce $m$
    - Abstraction
  - Reduce the effect of $n$
    - Partial-order reductions
  - Reduce $n$
    - Symmetry reductions

We’ll consider these 2 here
Partial-Order Reductions

- Reduce the number of interleavings of independent concurrent transitions

- $x := 1 \ || \ y := 1$ where initially $x = y = 0$

No Reductions | Transitions Reduced | States Reduced
Basic Ideas

• Independence
  – Independent transitions cannot disable nor enable each other
  – Enabled independent transitions are commutative

• Partial-order reductions only apply during the on-the-fly construction of the Kripke structure

• Based on a selective search principle
  – Compute a subset of enabled transitions in a state to execute

• Sleep sets (Reduce transitions)

• Persistent sets (Reduce states)
Persistent Set Reductions

- A subset of the enabled transitions is called *persistent*, when any transition can be executed from outside the set and it will not interact or affect the transitions inside the set
  - Use the static structure of the system to determine what goes into the persistent set
  - Note, all enabled transitions are trivially persistent
- Only execute transitions in the persistent set
- Persistent set algorithm is used within SPIN
- See papers by Godefroid and Peled
Abstraction

- Type based abstractions
  - Abstract Interpretation
  - Replace integer variable with odd-even range
  - Or Signs abstraction: negative, zero, positive
  - Replace all operations on the concrete variable with corresponding abstract operations
    - add(pos,pos) = pos
    - subtract(pos,pos) = negative | zero | pos
    - eq(pos,pos) = true | false

- Predicate Abstraction (Graf, Saïdi see also Uribe)
  - Create abstract state-space w.r.t. set of predicates defined in concrete system
Predicate Abstraction

Mapping of a concrete system to an abstract system, whose states correspond to truth values of a set of predicate.

- Create abstract state-graph during model checking, or,
- Create an abstract transition system before model checking.

\[
\alpha : \text{int} \times \text{int} \rightarrow \text{bool}
\]
Example
Predicate Abstraction

Predicate: \( B \equiv (x = y) \)

Concrete Statement
\( y := y + 1 \)

Abstract Statement

Step 1: Calculate pre-images
\[
\{ x = y + 1 \} \quad y := y + 1 \quad \{ x = y \} \\
\{ x \neq y + 1 \} \quad y := y + 1 \quad \{ x \neq y \}
\]

Step 2: Rewrite in terms of predicates
\[
\{ x = y + 1 \} \quad y := y + 1 \quad \{ B \} \\
\{ B \} \quad y := y + 1 \quad \{ \neg B \}
\]

Step 3: Abstract Code
\[
\text{IF } B \text{ THEN } B := \text{false} \\
\text{ELSE } B := \text{true} | \text{false}
\]
The Story so Far

• Explicit State Model Checking
  – Kripke structures
    • Describing the systems we want to check
  – Temporal logic
    • Describing the properties we want to check
  – Automata-theoretic model checking
  – State-explosion problem
    • Partial-order reductions
    • Predicate Abstraction

• Next - Model Checking Programs
  – A brief history of the field
  – Java PathFinder
Model Checking Programs

• Model checking usually applied to designs
  – Some errors get introduced after designs
  – Design errors are missed due to lack of detail
  – Sometimes there is no design

• Can model checking find errors in real programs?
  – Yes, many examples in the literature

• Can model checkers be used by programmers?
  – Only if it takes real programs as input
Main Issues

• Memory
  – Explicit-state model checking’s Achilles heel
  – State of a software system can be complex
  – Require efficient encoding of state, or,
  – State-less model checking

• Input notation not supported
  – Translate to existing notation
  – Custom-made model checker

• State-space Explosion
State-less Model Checking

- Must limit search-depth to ensure termination
- Based on partial-order reduction techniques
- Annotate code to allow verifier to detect “important” transitions
- Examples include
  - VeriSoft
    - [http://www1.bell-labs.com/project/verisoft/](http://www1.bell-labs.com/project/verisoft/)
  - Rivet
    - [http://sdg.lcs.mit.edu/rivet.html](http://sdg.lcs.mit.edu/rivet.html)
Traditional Model Checking

• Translation-based using existing model checker
  – Hand-translation
  – Semi-automatic translation
  – Fully automatic translation

• Custom-made model checker
  – Fully automatic translation
  – More flexible
Hand-Translation

- Hand translation of program to model checker’s input notation
- “Meat-axe” approach to abstraction
- Labor intensive and error-prone
Hand-Translation Examples

• Remote Agent – Havelund, Penix, Lowry 1997
  – [http://ase.arc.nasa.gov/havelund](http://ase.arc.nasa.gov/havelund)
  – Translation from Lisp to Promela (most effort)
  – Heavy abstraction
  – 3 man months

• DEOS – Penix et al. 1998/1999
  – [http://ase.arc.nasa.gov/visser](http://ase.arc.nasa.gov/visser)
  – C++ to Promela (most effort in environment)
  – Limited abstraction - programmers produced sliced system
  – 3 man months
Semi-Automatic Translation

• Table-driven translation and abstraction
  – Feaver system by Gerard Holzmann
  – User specifies code fragments in C and how to translate them to Promela (SPIN)
  – Translation is then automatic
  – Found 75 errors in Lucent’s PathStar system

• Advantages
  – Can be reused when program changes
  – Works well for programs with long development and only local changes
Fully Automatic Translation

- **Advantage**
  - No human intervention required

- **Disadvantage**
  - Limited by capabilities of target system

- **Examples**
  - **Java PathFinder** - [http://ase.arc.nasa.gov/havelund/jpf.html](http://ase.arc.nasa.gov/havelund/jpf.html)
    - Translates from Java to Promela (Spin)
    - Translates from Java to Promela (or dSpin)
    - Translates from Java bytecode to Promela, SMV or dSpin
Custom-made Model Checkers

- Allows efficient model checking
  - Often no translation is required
  - Algorithms can be tailored
- Translation-based approaches
  - dSpin
    - Spin extended with dynamic constructs
    - Essentially a C model checker
  - Java Model Checker (from Stanford)
    - Translates Java bytecode to SAL language
    - Custom-made SAL model checker
    - http://sprout.stanford.edu/uli/
Java PathFinder 2

- Based on new Java Virtual Machine
  - Handle all of Java, since it works with bytecodes
- Written in Java
  - 1 month to develop version with only integers
- Efficient encoding of states
  - Complex states are translated to integer vector
  - Garbage collection
  - Canonical heap representation
- [http://ase.arc.nasa.gov/jpf](http://ase.arc.nasa.gov/jpf)
JPF2 Core Decisions

- Explicit-state model checking
- Build own Java Virtual Machine
  - Emphasis on memory management not speed
  - Bytecode level assures language coverage
- Modular design to allow flexible system
  - Different search algorithms from testing to model checking
Our JVM

• Written in Java
  – *Java-in-Java* at SUN and *Rivet* at MIT

• Use *JavaClass* package
  – Class loading, Internal class file structure, etc.

• Initial implementation took 1 month!

• State encoded in complex data-structures

• Exploit Java in Java at every opportunity
Memory Management

- **SystemState**
  - KernelState, i.e. JVM state
  - Scheduler, used for exploring all paths

- **SystemState goes on Stack to allow backtracking**
  - Use “clone” operation to store states on the stack
  - It can be slow

- **KernelState goes in HashTable to record states**
  - Generic part of system, stores byte-vectors
  - Use “Pools” to reduce complex state to byte-vector
Reducing State Size

- Use intermediate tables/pools for each major class
  - Threads, Objects, Fields, Monitors
- Each time an object of the class must be stored, see if it is in the table, if so return the index, else insert it into next open slot and return the index
- Works well if tables don’t become too big
- Optimization
  - Only calculate index when object has changed.
Reducing the State Space

- Partial-order reductions
  - Vital for efficient explicit-state model checking
  - Must be able to identify independent transitions
    - Static analysis

- Abstraction
  - Under-approximations
    - Slicing, i.e. a cultured “meat-axe”
  - Over-approximations
    - Predicate abstraction
    - Type-based abstraction
Slicing in JPF

- JPF uses Bandera’s slicer
- Bandera slices w.r.t.
  - Deadlock - i.e. communication statements
  - Variables occurring in temporal properties
  - Variables participating in race-violations
    - Used with JPF’s runtime analysis
- More examples of slicing for model checking
  - Slicing for Promela (Millet and Teitelbaum)
  - Slicing for Hardware Description Languages (Shankar et al.)
JPF Abstraction Technique

- Find abstraction mapping ($\alpha$) by user guidance
- Use decision procedures to automatically compute abstract interpretation of concrete transitions
- Validity checking of pre-images
- Over approximation with nondeterminism

EQ := EQ ? F : T or F
EQ \equiv (x = y)

x = m
y = n
y++
x = m
y = n+1
JPF’s Java Abstraction

• Annotations used to indicate abstractions
  - Abstract.remove(x);
  - Abstract.remove(y);
  - Abstract.addBoolean("EQ", x==y);

• Tool generates abstract Java program
  - Using Stanford Validity Checker (SVC)
  - JVM is extended with nondeterminism to handle overapproximation

• Abstractions can be local to a class or global across multiple classes
  - Abstract.addBoolean("EQ", A.x==B.y);
  - Dynamic predicate abstraction, since it works across instances
Runtime Analysis

• Execute program once, and accumulate information to be analyzed
• Analysis can reveal an error potential although an error did not occur during the run
  – Looking for “footprints” of errors
• Data race violations with Eraser algorithm
• Lock order violations that lead to deadlock
Example Usage

- Deep-Space 1 software deadlocked in flight
- Create a Java program from the Lisp
- Too large to model check \(10^{60}\) states
- Do race analysis and find violation
- Create slice (with Bandera) w.r.t. variable for which race violation occurred
- Model Check slice and find error (instantly)
Conclusions

• Low-hanging fruit principle
  – Errors always obvious (in hindsight!)
  – Model checkers are good at finding obvious errors

• Combine many different techniques
  – Abstraction, slicing, runtime analysis, etc.

• Current work
  – Adding property specification language
  – Finding “real” counter-examples during abstraction
  – Defining “environments” for Java programs

• http://ase.arc.nasa.gov/jpf
Part II
Symbolic Model Checking

- Principles
  - BDDs
  - Symbolic MC algorithm

- Tools: SMV
  - Principles
  - Language
  - Variants

- Applications in Software
  - Model-based autonomy: Livingstone
  - Robot control: TCL

Based on material from:
- Edmund Clarke
- Marius Minea
- Reid Simmons
Symbolic Model Checking Principles
What is it?

Instead of considering each individual state, Symbolic model checking...
What is it?

Instead of considering each individual state,
Symbolic model checking...

• Manipulates sets of states,
What is it?

Instead of considering each individual state,
Symbolic model checking...

• Manipulates sets of states,

• Represented as boolean formulas,

\[ x = 2 \lor y = 1 \]
What is it?

Instead of considering each individual state, Symbolic model checking...

- Manipulates sets of states,
- Represented as boolean formulas,
- Encoded as binary decision diagrams.
What is it?

Instead of considering each individual state, Symbolic model checking...

• Manipulates sets of states,
  – Can handle very large state spaces ($10^{50} +$)

• Represented as boolean formulas,
  – Suited for boolean/abstract models

• Encoded as binary decision diagrams.
  – The limit is BDD size (hard to control)
Boolean Functions

- Represent a state as boolean variables
  \[ s = b_1, \ldots, b_n \]
  Non-boolean variables => use boolean encoding

- A set of states as a boolean function
  \[ s \text{ in } S \text{ iff } f(b_1, \ldots, b_n) = 1 \]

- A transition relation as a boolean function over two states
  \[ s \rightarrow s' \text{ iff } f(b_1, \ldots, b_n, b'_1, \ldots, b'_n) = 1 \]
Binary Decision Trees

• Encoding for boolean functions

• Notational convention:
  \[ c \begin{cases} e & \text{if } c \\ e' & \text{else} \end{cases} = (c ? e : e') \]

• Always exists but not unique

\[(a \mid b) \Rightarrow c\]
From Trees to Diagrams

- Fixed variable ordering

"layered" tree

\[(a | b) \Rightarrow c\]
From Trees to Diagrams

• Fixed variable ordering
  "layered" tree
• Merge equal subtrees

\[(a \mid b) \Rightarrow c\]
From Trees to Diagrams

- Fixed variable ordering
  "layered" tree
- Merge equal subtrees
- Remove nodes with equal subtrees

\[(a \mid b) \Rightarrow c\]

=> Ordered Binary Decision Diagram
[Ordered] Binary Decision Diagrams

- [O]BDDS for short
- Unique normal form
  - for a given ordering and
  - up to isomorphism
  => compare in constant time
    (using hash table)

(a ∣ b) ⇒ c
Computations with BDDs

• Negation \( \neg f \):
  swap leaves 0 and 1.

• Boolean combinator \( f \# g \):
  \[(b ? f' : f'') \# (b ? g' : g'') = (b ? f'\# g' : f''\# g'') \]
  cache results \(\rightarrow O(|f|.|g|) \) time

• Instantiation \( f[b=1], f[b=0] \):
  \[(b ? f' : f'')[b=1] = f' \]

• Quantifiers \( \textbf{exists } b . f, \textbf{forall } b . f \):
  \[\textbf{exists } b . f = f[b=1] \mid f[b=0] \]
Variable Ordering

• Must be the same for all BDDs
• Size of BDDs depends critically on ordering
• Worst case: exponential w.r.t. #variables
  – sometimes exponential for any ordering
    e.g. middle output bit of n-bit multiplier
• Finding optimum is hard (NP-complete)
  => optimization uses heuristics
Transition Systems with BDDs

Given boolean state variables \( v = b_1, \ldots, b_n \)

a set of states as a BDD \( p(v) \)

a transition relation as a BDD \( T(v, v') \)

we can compute the predecessors and successors of \( p \)

as BDDs:

\[
(p \text{pred } p)(v) = \exists v' . T(v, v') \land p(v')
\]

\[
(p \text{succ } p)(v) = \exists v' . p(v') \land T(v', v)
\]
Checking Formulas with BDDs

Functional evaluation as set of states:

- for every formula $p$, build the BDD $p(v)$ of the set of states that satisfy $p$
- Top level: for a set of initial states $I$, $I$ satisfy $p$ iff $\neg p \& I = 0$
- $p = \text{op}(q,r) \Rightarrow$ build $p(v)$ based on $q(v)$, $r(v)$
CTL temporal logic

Computation Tree Logic (branching):
Consider the tree of possible executions

Always ...

\[ AX \ p \] \quad \text{AG} \ p \quad \text{AG} \ p \quad \text{AG} \ p \\
\text{In all states}

Sometimes ...

\[ EX \ p \] \quad \text{EG} \ p \quad \text{EG} \ p \quad \text{EG} \ p \\
\text{In some state}

\[ \text{Next} \ p \] \quad \text{Globally} \ p \quad \text{Finally} \ p \quad \text{p Until} \ q \\
\text{duals}

\[ A[p \ U \ q] \] \quad \[ E[p \ U \ q] \]
CTL operators
as BDDs

\( (EX \ p)(v) = (\text{pred } p)(v) = \exists v’. T(v, v’) \land p(v’) \)

\( (EG \ p)(v) = (\text{gfp } U . \ p \land EX \ U)(v) \)

\( (E[p \ U \ q])(v) = (\text{lfp } U . \ q \mid (p \land EX \ U))(v) \)

All others can be expressed as \( EX/EG/EU \)

\[ \begin{align*}
EF \ p &= E[1 \ U \ p] \\
AX \ p &= !EX \ !p \\
AG \ p &= !EF \ !p \\
AF \ p &= !EG \ !p \\
A[p \ U \ q] &= !E[!q \ U \ !p \land !q] \land !EG \ !q
\end{align*} \]
Evaluating Fixpoints with BDDS

Compute \( \text{lfp } U \cdot F[U] \) as a BDD:

\[
U_0(v) = 0 \\
U_1(v) = F[U_0](v) = F[0](v) \\
... \\
U_{n+1}(v) = F[U_n](v) = F^n[0](v)
\]

until \( U_n(v) = U_{n+1}(v) = (\text{lfp } U \cdot F[U])(v) \)

– Convergence assured because finite domain
– Dual construction for \( \text{gfp} \)
process P(id) {
    repeat {
        x=getFlag();
    } until x=false;
    setFlag();
    CS(id);
    resetFlag();
}

start P(1);
start P(2);

\[ \text{EF } p = \text{lfp } U \cdot p \mid \text{EX } U \]
\[ U_0 = 0 \]
process P(id) {
    repeat {
        x=getFlag();
    } until x=false;
    setFlag();
    CS(id);
    resetFlag();
}

\[ \text{EF} \ p = \text{lfp} \ U . p \ | \ \text{EX} \ U \]
\[ U_0 = 0 \]
\[ U_1 = p \ | \ \text{EX} \ U_0 = p \]
process P(id) {
    repeat {
        x=getFlag();
    } until x=false;
    setFlag();
    CS(id);
    resetFlag();
}

start P(1);
start P(2);

EF \( p = \text{lfp} \ U \cdot p \mid \text{EX} U \)

\( U_0 = 0 \)
\( U_1 = p \mid \text{EX} U_0 = p \)
\( U_2 = p \mid \text{EX} U_1 \)
process P(id) {
  repeat {
    x=getFlag();
  } until x=false;
  setFlag();
  CS(id);
  resetFlag();
}

start P(1);
start P(2);

\[
\begin{align*}
\text{EF } p &= \text{lfp } U . p \mid \text{EX } U \\
U_0 &= 0 \\
U_1 &= p \mid \text{EX } U_0 = p \\
U_2 &= p \mid \text{EX } U_1 \\
\vdots \\
U_5 &= p \mid \text{EX } U_4 \\
U_6 &= p \mid \text{EX } U_5 = U_5 \\
\Rightarrow \text{EF } p &= U_5 \\
\Rightarrow \text{EF } p \land I \neq 0 \\
\Rightarrow \text{AG } !p \text{ does not hold}
\end{align*}
\]
Fairness, LTL

• CTL+fairness:
  – Only check executions where fairness conditions $c_1, \ldots, c_n$ hold infinitely often
  – Symbolic evaluation: express $c_1, \ldots, c_n$ as BDDs, modified algorithms for $\text{EX}, \text{EG}, \text{EU}$.

• Symbolic model checking of LTL
  – Convert LTL formula to Büchi automaton
  – Encode automaton in transition relation
  – Express acceptance condition in CTL+fairness
Bounded Model Checking

- Principle:
  - $n+1$ copies of state variables $v_0, .., v_n$
  - Unroll transition relation $n$ times $T(v_{k-1}, v_k)$
  - Embed property to be satisfied
  - Verify satisfiability with SAT procedure
- Verifies traces up to length $n$
  - Iterate over values of $n$ => breadth-first search
- No state space explosion (polynomial space)
- Usually fast (but worst case is exponential time)
Symbolic Model Checking

Summary

• Principle: compute over sets of states encoded as BDDs.
• Can handle huge state spaces.
• CTL + fairness, LTL.
• Some tweaking may be needed.
  – variable ordering
• Some models blow up nevertheless.
• New alternative: SAT-based (bounded).

*The seminal paper on Binary Decision Diagrams.*


*Survey paper on the principles of symbolic model checking.*


*Paper on SAT-based bounded model checking.*
Symbolic Model Checking

References (cont'd)


*Symbolic model checking of CTL with fairness.*


*Verifying LTL using symbolic model checking.*
Symbolic Model Checking Tools:
SMV
Overview

- **SMV** = **S**ymbolic **M**odel **V**erifier.
- Developed by Ken McMillan at Carnegie Mellon University.
- Modeling language for transition systems based on parallel assignments.
- Specifications in temporal logic **CTL**.
- **BDD-based symbolic model checking**: can handle very large state spaces.
What SMV Does

```
MODULE user(...) ...

MODULE main
VAR turn: {1, 2};
  user1: user(...);
...

SPEC AG !(user1.state = c) & (user2.state = c)
```

-- specification AG ... is false
-- as demonstrated by ...
state 1.1:
  turn = 1
  user1.state = n
  user2.state = n
state 1.2:
...
resources used: ...

```
mutex.smv <stdout>
```
MODULE user(turn, id, other)
VAR state: {n, t, c};
DEFINE my_turn :=
    (other=n) | ((other=t) & (turn=id));
ASSIGN
init(state) := n;
next(state) := case
    (state = n) : {n, t};
    (state = t) & my_turn: c;
    (state = c) : n;
    1 : state;
esac;

SPEC AG((state = t) → AF (state = c))
MODULE main
VAR turn: {1, 2};
    user1: user(turn, 1, user2.state);
    user2: user(turn, 2, user1.state);
ASSIGN
init(turn) := 1;
next(turn) := case
    (user1.state=n) & (user2.state=t): 2;
    (user2.state=n) & (user1.state=t): 1;
1: turn;
esac;

SPEC AG !((user1.state=c) & (user2.state=c))
SPEC AG !(user1.state=c)
Diagnostic Trace Example

-- specification AG (state = t -> AF state = c) (in module user1) is true
-- specification AG (state = t -> AF state = c) (in module user2) is true
-- specification AG (!(user1.state = c & user2.state = c)... is true
-- specification AG (!user1.state = c) is false
-- as demonstrated by the following execution sequence
state 1.1:
  turn = 1
  user1.state = n
  user2.state = n

state 1.2:
  user1.state = t

state 1.3:
  user1.state = c
• The SMV program defines:
  – a finite transition model $M$ (Kripke structure),
  – a set of possible initial states $I$ (may be several),
  – specifications $P_1 \ldots P_m$ (CTL formulas).

• For each specification $P$, SMV checks that

$$\forall s_o \in I . \ M, s_o \models P$$

Note: SPEC $\neg P$ is not the negation of SPEC $P$:
both can be false (in some initial states),
both can be true (vacuously when $I=\emptyset$).
VAR state: \{n, t, c\};
ASSIGN
init(state) := n;
next(state) := case
  (state = n) : \{n, t\}; ...
esac;

• Finite data types (incl. numbers and arrays).
• Usual operations \(x \& y, x+y, \text{ etc.}, \) case statement.
• All assignments are evaluated in parallel.
• No control flow (must be simulated with vars).
• SMV detects circular and duplicate assignments.
Defined Symbols

DEFINE my_turn :=
    other=n | (other=t & turn=id);

ASSIGN
next(state) := case ...
    (state = t) & my_turn:  c; ...
esac;

• Defines an abbreviation (macro definition).
• No new state variable is created
  => no added complexity for model checking.
• No type declaration is needed.
MODULE user (turn, id, other)

VAR ...

ASSIGN ...

MODULE main

VAR user1: user (turn, 1, user2.state);

...

• Parameters passed by reference.

• Top-level module main.

• Composition is synchronous by default: all modules move at each step.
Records

Modules without parameters and assignments.

MODULE point
VAR x : \{0,1,2,3,4,5\};
  y : \{0,1,2,3,4,5\};

MODULE main
VAR  p : point;
ASSIGN
  init(p.x) := 0; init(p.y) := 0;
  ...

Processes

VAR node1: process node(1);
node2: process node(2);

• Composition of processes is asynchronous: one process moves at each step.
• Boolean variable running in each process
  – running=1 when that process is selected to run.
  – Used for fairness constraints (see later).
Specifications

SPEC AG ((state = t) -> AF (state = c))

"Whenever state t is reached, state c will always eventually be reached."

• Standard CTL syntax:
  AX p, AF p, AG p, A[p U q], EX p, ...

• Can be added in any module.

• Each specification is verified separately.
Fairness

MODULE user(turn,id,other)
VAR ... 
ASSIGN ...
SPEC AG AF (state = c)
FAIRNESS (state = t)

• Check specifications, assuming fairness conditions hold repeatedly (infinitely often).
• Useful for liveness properties.
• Fair scheduling: FAIRNESS running
Variables and Transitions
(Constraint Style)

VAR pos: \{0,1,2,3,4,5\};
INIT pos < 2
TRANS (next(pos)-pos) in \{+2,-1\}
INVAR !(pos=3)

• Any propositional formula is allowed
  => flexible for translation from other languages.

• INVAR p is equivalent to INIT p
  TRANS next(p)
  but implemented more efficiently.

• Risk of inconsistent models (TRANS p & !p).
Well-Formed Programs?

• In assignment style, by construction:
  – always at least one initial state,
  – all states have at least one next state,
  – non-determinism is apparent (unassigned variables, set assignments, interleaving).

• In constraint style:
  – INIT and TRANS constraints can be inconsistent,
  – the level of non-determinism is emergent from the conjunction of all constraints.
Inconsistency

- **Inconsistent** $\text{INIT}$ constraints
  $\Rightarrow$ **inconsistent model**: no initial state.
  - $\text{SPEC } 0$ (or any $\text{SPEC } p$) is vacuously true.

- **Inconsistent** $\text{TRANS}$ constraints
  $\Rightarrow$ **deadlock state**: state with no next state
  $\Rightarrow$ transition relation is **not complete**.
  - SMV **does not work** correctly in this case.
  - SMV will **detect and report** it.
Variable Ordering

• BDDs require a fixed variable ordering.
  – Critical for performance (BDD size).
  – Best one is hard to find (NP-complete).

• SMV does not optimize by default but
  – can read, write ordering in a file,
  – can search for better ordering on demand.
Re-ordering Variables

Using command line options:

```
smv -o demo.var
```

Outputs variable ordering to `demo.var`. 
`demo.var` is text, can be re-ordered manually.

```
smv -i demo.var
```

Inputs variable ordering from `demo.var`.

```
smv -reorder
```

Does variable re-ordering when BDD size exceeds a certain (configurable) limit.

```
smv -reorder -oo demo.var
```

Outputs to `demo.var` after each change.
Re-ordering Variables
Method for Tough Cases

Problem (Livingstone ISPP model):

```
smv ispp.smv
-> Memory overflow.
smv -reorder ispp.smv
-> keeps re-ordering again and again...
```

Solution:

```
smv -reorder -oo ispp.var ispp.smv
Wait until "enough" re-ordering (statistics).
^C
smv -i ispp.var ispp.smv
-> Goes to completion (10^{50} states).
```
Availability

- Freely downloadable.
- Source or binaries for Unix (SunOS4, SunOS5, Linux x86, Ultrix).
- Windows NT port (Dong Wang).
- see [http://www.cs.cmu.edu/~modelcheck/smv.html](http://www.cs.cmu.edu/~modelcheck/smv.html)
NuSMV

• From ITC-IRST (Trento, Italy) and CMU.
• New version of SMV, completely rewritten:
  – Same language as SMV.
  – Modular, documented APIs, easily customized.
  – Specifications in CTL or LTL.
  – Graphical User Interface.
  – Usually faster but uses more memory.
• See http://sra.itc.it/tools/nusmv/index.html
Other Related Tools

• **Cadence SMV** (Cadence Berkeley Labs)
  – From [Ken McMillan](http://www-cad.eecs.berkeley.edu/~kenmcmil/smv/), original author of SMV.
  – Supports refinement, compositional verification.
  – New language but accepts CMU SMV.
  – see [http://www-cad.eecs.berkeley.edu/~kenmcmil/smv/](http://www-cad.eecs.berkeley.edu/~kenmcmil/smv/)

• **BMC = Bounded Model Checker** (CMU)
  – Uses SAT procedures instead of BDDs: bounded depth but usually faster, less memory.
  – Simple SMV-like language (no modules).
  – Early beta version.
  – see [http://www.cs.cmu.edu/~modelcheck/bmc.html](http://www.cs.cmu.edu/~modelcheck/bmc.html)
SMV
Summary

• BDD-based symbolic model checker.
• Modeling language based on synchronous transition systems.
• Constraint style: more versatile, less strict => good for use as back-end tool.
• 1st generation: CMU
• 2nd generation: Cadence, NuSMV
• Variant: BMC (SAT based)
SMV

References


*Based on Ken McMillan's PhD thesis on SMV.*


[http://www.cs.cmu.edu/~modelcheck/smv/smvmanual.r2.2.ps](http://www.cs.cmu.edu/~modelcheck/smv/smvmanual.r2.2.ps)

*The (old) user manual provided with the SMV program.*


*Survey paper on NuSMV.*
Symbolic Model Checking
Applications in Software
Applications of Symbolic Model Checking

• Used in **industry** for **hardware design**
  – Commercial tools (Cadence)
  – Fits well with boolean modeling

• Some **success stories in protocol design**
  – Cache coherence of IEEE Futurebus+
  – HDLC

• **Research stage for software design**
  – **Gap** between **programming/design language**
    and **verification modeling language**.
Model-Based Autonomy

- Unattended control of a complex device (e.g. a spacecraft)
- Based on AI technology
- General reasoning engine + application-specific model
- Use model to respond to unanticipated situations

=> Verify the model!
Verification of Autonomy Models

Model-Based Autonomy

Verification

Model-Based Autonomous Controller

Domain Model

Domain Requirements

Domain Trace

SMV Model

SMV Specifications

SMV Trace

SMV
The Livingstone Diagnostic System

- Mode identification & recovery:
  - identify current state (including faults)
  - find path to goal state
- Model-based
- From NASA Ames
- Run in space (DS-1, May 1999)
Livingstone Models

- Models = concurrent transition systems
- Qualitative values \( \Rightarrow \) finite state
- Nominal/fault modes
- Probabilities on faults

Courtesy Autonomous Systems Group, NASA Ames
**Formalizing the Model**

<table>
<thead>
<tr>
<th>Livingstone model</th>
<th>SMV model</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic component</td>
<td>module</td>
</tr>
<tr>
<td>compound module</td>
<td>module</td>
</tr>
<tr>
<td>variables</td>
<td>scalar variables</td>
</tr>
<tr>
<td>structures</td>
<td>module variables</td>
</tr>
<tr>
<td>mode transitions</td>
<td>TRANS</td>
</tr>
<tr>
<td>model constraints</td>
<td>INVAR</td>
</tr>
<tr>
<td>initial state</td>
<td>INIT</td>
</tr>
</tbody>
</table>

*Main difficulty is translating Livingstone’s flat name space*
Translating Models

Livingstone Model

(defcomponent valve ()
  (:inputs (cmd :type valve-cmd))
  ...
  (Closed :type ok-mode
    :transitions
      ((do-open :when (open cmd)
        :next Open) ...))
  (StuckC :type :fault-mode ...)
  ...)

SMV Model

MODULE valve
VAR mode: {Open,Closed,
  StuckO,StuckC};
  cmd: {open,close};
DEFINE faults:={StuckO,StuckC};
TRANS
  (mode=Closed & cmd=open) ->
  (next(mode)=Open | next(mode) in faults)
Translating Requirements

Livingstone Requirement

(defverify ...
 (:specification
  (always (globally (implies
    (not (broken))
    (exists (eventually
      (high flow-in)))))))

SMV Requirement

SPEC AG (overflow or (!broken) ->
EF (ISPP.valve.flow-in = high))

- Declaration (defverify ...) added to the Livingstone model.
- Temporal logic formulas (CTL) in Livingstone syntax.
- Auxiliary predicates (e.g. failed component).
- High-level property patterns (e.g. reachability of modes).
Application

In-Situ Propellant Production

• Use atmosphere from Mars to make fuel for return flight.
• Livingstone controller developed at NASA Kennedy.
• Components are tanks, reactors, valves, sensors...
• Exposed improper flow modeling.
• Very "loose" state space:
  – $10^{50}$ states
  – all states reachable in 3 steps

\[
\text{Mars atmosphere} \quad \text{on-board} \\
\text{CO}_2 + 2\text{H}_2 \rightarrow \text{CH}_4 + \text{O}_2
\]
TDL: Task Description Language

• Extension of C++
• Task decomposition, task synchronization, monitoring, exception handling
• From Carnegie Mellon University
• Used for robot control architectures
## Formalizing TDL

<table>
<thead>
<tr>
<th>TDL model</th>
<th>SMV model</th>
</tr>
</thead>
<tbody>
<tr>
<td>task</td>
<td>module</td>
</tr>
<tr>
<td>task/subtask relationship</td>
<td>module variables</td>
</tr>
<tr>
<td>task state</td>
<td>scalar variables</td>
</tr>
<tr>
<td>state transitions</td>
<td>ASSIGN</td>
</tr>
<tr>
<td>temporal constraints</td>
<td>INVAR and parameters</td>
</tr>
<tr>
<td><em>asynchronous nature</em></td>
<td><em>PROCESS variables</em></td>
</tr>
<tr>
<td></td>
<td><em>and FAIRNESS constraints</em></td>
</tr>
</tbody>
</table>
Verifying Task Descriptions

Work by Reid Simmons (CMU)

• Can verify temporal properties of hierarchical tasks
  – deadlock, safety, liveness, …
  – can handle conditional execution

• In progress:
  – monitoring and exception handling
  – iteration and recursion
Applications of SMV
Summary

• Symbolic model checking: **OK** for hardware, **quid** for software?
• Needs translation from **programming language** to **verification language** and back!
• 2 examples for **autonomy software** using SMV.
Applications of SMV

References


*Verification of Livingstone with SMV.*


*Verification of TDL with SMV.*