Verification Tools for Safety and Security Systems

Introduction
Dr. DONG, Jin–Dong, Dr. SUN Jun and Dr. LIU Yang
School of Computing, NUS

Outline

- System Dependability
  - Safety and Security
- Formal Methods
  - Early History
- Formal Specification
  - State and Event based Languages
- Formal Verification
  - Theorem Proving and Model Checking
System dependability

- For critical systems, it is usually the case that the most important system property is the dependability of the system.
- The dependability of a system reflects the user’s degree of trust in that system. It reflects the extent of the user’s confidence that it will operate as users expect and that it will not ‘fail’ in normal use.
- Dimensions of dependability are:
  - Availability;
  - Reliability;
  - Safety;
  - Security

Dimensions of dependability

- **Availability**: The ability of the system to deliver services when requested
- **Reliability**: The ability of the system to deliver services as specified
- **Safety**: The ability of the system to operate without catastrophic failure
- **Security**: The ability of the system to protect itself against accidental or deliberate intrusion
Availability and reliability

- Reliability
  - The probability of failure-free system operation over a specified time in a given environment for a given purpose
- Availability
  - The probability that a system, at a point in time, will be operational and able to deliver the requested services
- Both of these attributes can be expressed quantitatively

Safety

- Safety is a property of a system that reflects the system’s ability to operate, normally or abnormally, without danger of causing human injury or death and without damage to the system’s environment
- It is increasingly important to consider software safety as more and more devices incorporate software-based control systems
- Safety requirements are exclusive requirements i.e. they exclude undesirable situations rather than specify required system services
Security

- The security of a system is a system property that reflects the system’s ability to protect itself from accidental or deliberate external attack.
- Security is becoming increasingly important as systems are networked so that external access to the system through the Internet is possible.
- Security is often an essential prerequisite for availability, reliability, and safety.

Formal Methods for Safety/Security Systems

- Major goal of software engineers
  - Develop reliable (esp. for safety/security) systems
- Formal Methods
  - Mathematical languages, techniques, and tools
  - Used to specify and verify systems
  - Goal: to construct more reliable systems
- A mean to examine the entire state space of a design (whether hardware or software)
  - Establish a correctness or safety/security property that is true for all possible inputs.
Formal Methods Early History

- History can trace back to Goldstine, Neumann and Turing
- Floyd, Hoare, and Dijkstra

Beyond Program Reasoning

- Formal methods can be applied at various points through the development process
  - Specification (state & event based)
  - Verification (theorem proving & model checking)
- Specification: Give a description of the system to be developed, and its properties
- Verification: Prove or disprove the correctness of a system with respect to the formal specification or property
**Data/State based Specification**

- Formal methods for specification of the sequential systems
  - Z (Oxford, Abrial/Woodcock ...1985)
  - VDM (IBM, Jones 1986)
  - Alloy (MIT, Jackson 2006)
- **States** are described in rich math structures (set, relation, function)
- **Transition** are described in terms of predicates

**Event Based Specification**

- Formal methods for specification of the concurrent systems
  - CCS (Cambridge, Milner 1980)
  - CSP (Oxford, Hoare 1985)
  - Temporal Logic (Stanford, Pnueli 1981)
- **States** range over simple domains, like integers
- **Behavior** is defined in terms of sequences, trees, partial orders of events
Verification

- Two well established approaches to verification
  - Model Checking
  - Theorem Proving
- Model checking
  - Build a finite model of system and perform an exhaustive search
- Theorem Proving
  - Mechanization of a logical proof

Theorem Proving

- Both the system and its desired properties are expressed in some mathematical logic
- Theorem proving is the process of finding a proof from the axioms of the system
- In contrast to model checking, it can deal with infinite space and relies on techniques like reduction
- E.g. HOL/Isabelle (Cambridge), PVS (SRI), Coq (INRIA) etc
Model Checking

- Determining whether a model satisfies a property by the means of exhaustive searching (fully automatic)
- SMV(CUM), SPIN(Bell–lab), FDR(Oxford) … etc
  - The main disadvantage: state explosion problem!

Model Checking Success!

- Three researchers won Turing Award 2007 for their pioneer work on model checking!
- Intel i7 processor is verified by symbolic model checking completely without executing a single test case!
- The Slam project from Microsoft successfully detected many bugs in many driver software!

- In 2002, Bill Gates pointed out Microsoft's use of formal methods, saying that:
  
  "For things like software verification, this has been the Holy Grail of computer science for many decades. But now, in some very key areas for example driver verification we're building tools that can do actual proofs of the software and how it works in order to guarantee the reliability. ".

"For things like software verification, this has been the Holy Grail of computer science for many decades. But now, in some very key areas for example driver verification we're building tools that can do actual proofs of the software and how it works in order to guarantee the reliability. ".
Alloy
Abstract Design and Analysis

J. S. Dong, Y. Liu, J. Sun
School of Computing
National University of Singapore

(thanks D. Jackson for providing some of the notes)

the atlantic divide

American school of formal methods
› emphasis on verification algorithms
› eg, SMV, SPIN, Murphi

European school
› emphasis on modelling
› eg, Z, VDM, B

Alloy brings together
› automatic analysis (like SMV)
› logical notation (like Z)
first order effects

Alloy is first order
› to allow exhaustive search

design implications
› no constructors: composites by projection
› no need to distinguish scalars from singleton sets

novel features
› no scalars or sets: all expressions are relation-valued
› generalized relational join operator
› finite interpretation

atoms

structures are built from
› atoms & relations

atoms are
› indivisible
  can’t be broken into smaller parts
› immutable
  don’t change over time
› uninterpreted
  no built-in properties

what’s atomic in the real world?
› very little -- a modelling abstraction
types

universe
› contains all atoms
› a finite (but perhaps big) set
› partitioned into basic types, each a set

- \( \text{DATE} = \{\text{JAN1, JAN2, \ldots, DEC31}\} \)
- \( \text{PERSON} = \{\text{ALICE, BOB, CAROL}\} \)
- \( \text{STATE} = \{\text{STATE0, STATE1, STATE2}\} \)
- \( \text{FILESYSTEM} = \{\text{FILESYSTEM0, FILESSTEM2}\} \)

no subtyping, so
› atoms that share properties share a type

- \( \text{Employer} = \{\text{ALICE}\} \)
- \( \text{Employee} = \{\text{BOB, CAROL}\} \)
- \( \text{Employer, Employee in PERSON} \)

relations

definition
› a tuple is a list of atoms
› a relation is a set of tuples

- \( \text{birthday} = \{(\text{ALICE,MAY1}), (\text{BOB,JAN4}), (\text{CAROL,DEC9})\} \)
- \( \text{likes} = \{(\text{ALICE,BOB}), (\text{BOB,CAROL}), (\text{CAROL,BOB})\} \)

typing
› a relation type is a non-empty list of basic types
› if \( i \)-th type is \( T \), then \( i \)-th atom in each tuple is in \( T \)

- \( \text{birthday: (PERSON, DATE)} \)
- \( \text{likes: (PERSON, PERSON)} \)
relations as tables

can view relation as table
  → atoms as entries, tuples as rows
  → order of columns matters, but not order of rows
  → can have zero rows, but not zero columns
  → no blank entries

example

\[ \text{birthday} = \{(\text{ALICE, MAY1}), (\text{BOB, JAN4}), (\text{CAROL, DEC9})\} \]

<table>
<thead>
<tr>
<th>PERSON</th>
<th>DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALICE</td>
<td>MAY1</td>
</tr>
<tr>
<td>BOB</td>
<td>JAN4</td>
</tr>
<tr>
<td>CAROL</td>
<td>DEC9</td>
</tr>
</tbody>
</table>

dimensions

arity
  → number of columns
  → relation of arity \( k \) is a \( k \)-relation
  → unary, binary, ternary for \( k = 1, 2, 3 \)
  → finite, \( > 0 \)

size
  → number of rows
  → finite, \( \geq 0 \)

\#p is an integer expression giving the size of \( p \)

homogeneity
  → relation of type \( (T, T, \ldots T) \) is homogeneous
  → else heterogeneous
relations as graphs

can view 2-relation as graph
› atoms as nodes
› tuples as arcs

eexample

likes = \{(ALICE, BOB), (BOB, CAROL), (CAROL, BOB)\}

sets and scalars

sets and scalars
› represented as relations
› set: a unary relation
› scalar: a unary, singleton relation

\[
\begin{align*}
\text{PERSON} &= \{(ALICE), (BOB), (CAROL)\} \quad \text{-- note (')s!} \\
\text{Employee} &= \{(BOB), (CAROL)\} \\
\text{Employer} &= \{(ALICE)\} \\
\text{Alice} &= \{(ALICE)\}
\end{align*}
\]

unlike standard set theory
› no distinction between
\[a, (a), \{a\}, \{(a)\}\]
ternary relations

for relationships involving 3 atoms

salary: [PERSON, COMPANY, SALARY]
salary = {((ALICE,APPLE,$60k), (BOB,BIOGEN,$70k))}

for associating binary relations with atoms

birthdayRecords: [BIRTHDAYBOOK, PERSON, DATE]
birthdayRecords =
{((BOB, ALICE, MAY1), (BOB, BOB, JAN4), (RR1, CAROL, DEC9))}

---

left and right

left and right sets
- left (right) set of p is set of atoms in left-(right-)most column

left and right types
- left (right) type of p is the first (last) basic type of p’s type

examples

likes = {((ALICE,BOB), (BOB,CAROL), (CAROL,BOB))
left-set(likes) = {((ALICE,BOB,CAROL))
right-set(likes) = {((BOB,CAROL))
left-type(likes) = right-type(likes) = PERSON
set operators

standard set operators

- union \( p + q \) contains tuples of \( p \) and tuples of \( q \)
- intersection \( p \& q \) contains all tuples in both \( p \) and \( q \)
- difference \( p - q \) contains tuples in \( p \) but not in \( q \)

interpretation of +

- for scalars, makes a set \( \text{Alice} + \text{Bob} \)
- for sets, makes a new set \( \text{Employer} + \text{Employee} \)
- for relations, combines maps \( \text{likes} + \text{Alice} \rightarrow \text{Bob} \)

subset and equality

- subset \( p \in q \) \( q \) contains every tuple \( p \) contains
- equality \( p = q \) \( p \) and \( q \) contain same set of tuples

product

definition

- if \( p \) contains \( (p_1, \ldots, p_n) \)
- and \( q \) contains \( (q_1, \ldots, q_m) \)
- then \( p \rightarrow q \) contains \( (p_1, \ldots, p_n, q_1, \ldots, q_m) \)

puns

- for sets \( s \) and \( t \), \( s \rightarrow t \) is cartesian product
- for scalars \( a \) and \( b \), \( a \rightarrow b \) is tuple

examples

- \( \text{birthday} = \text{Alice} \rightarrow \text{May1} + \text{Bob} \rightarrow \text{Jan4} + \text{Carol} \rightarrow \text{Dec9} \)
- \( \text{Employee} \rightarrow \text{Employee in likes} \)
join

definition

if p contains (p1,...,pn-1,pn)
and q contains (q1,...,qm)
and pn = q1
then p . q contains (p1,...,pn-1,q2,...,qm)

constraints

arity(p) + arity(q) > 2
right-type(p) = left-type(q)

join, examples

given

Alice = { (ALICE) }, bb0 = { (BB0) }
likes = { (ALICE,BOB), (BOB, CAROL), (CAROL, BOB) }
birthday = { (ALICE, MAY1), (BOB, JAN4), (CAROL, DEC9) }
birthdayRecords =
{ (BB0,ALICE,MAY1), (BB0,BOB,JAN4), (BB1, CAROL, DEC9) }

we have

Alice.likes = { (BOB) }; likes.Alice = {}
likes.birthday = { (ALICE, JAN4), (BOB, DEC9), (CAROL, JAN4) }
bb0.birthdayRecords = { (ALICE, MAY1), (BOB, JAN4) }
Alice.(bb0.birthdayRecords) = { (MAY1) }
join, puns

puns

for set s and binary relation r, s.r is image of s under r
for binary relations p and q, p.q is standard join of p and q
for binary relation r of type (S,T),
S.r is right-set of r
r.T is left-set of r

join variants

for non-binary relations, join is not associative

3 syntactic variants of join

\[ p.q = p::q = q[p] \]

binding power: :: most, then .., then []

\[ p.q:r = p.(q.r) \]

\[ p.q[r] = r.(p.q) \]

equivalent expressions

Alice.(bb0.birthdayRecords)
Alice.bb0::birthdayRecords
bb0.birthdayRecords [Alice]
**transpose**

for relation \( r: (S,T) \)

\( \sim r \) contains \((b,a)\) whenever \( r \) contains \((a,b)\)

\( \sim r \) has type \((T,S)\)

a theorem

for set \( s \) and binary relation \( r \),

\[ r.s = s.\sim r \]

**override**

for relations \( p,q: (S,T) \)

\( p++q \) contains \((a,b)\) whenever

\( q \) contains \((a,b)\), or

\( p \) contains \((a,b)\) and \( q \) does not map \( a \)

given

\( Alice = \{(ALICE)\}, \ March3 = \{(MAR3)\} \)

\( birthday = \{(ALICE,MAY1), (BOB,JAN4), (CAROL,DEC9)\} \)

we have

\( birthday ++ Alice->March3 = \)

\( \{(ALICE,MAR3), (BOB,JAN4), (CAROL,DEC9)\} \)
closure

for relation \( r \): \((T,T)\)

\[ ^r = r + r.r + r.r.r + r.r.r.r + \ldots \]

is smallest **transitive** relation \( p \) containing \( r \)

\[ *r = \text{iden}[T] + r + r.r + r.r.r + r.r.r.r + \ldots \]

is smallest **reflexive & transitive** relation \( p \) containing \( r \)

examples

ancestor = \(^\text{parent}\)  
reaches = \(^\text{connects}\)  
precedes = \(^\sim\text{next}\)

operator types

<table>
<thead>
<tr>
<th>if</th>
<th>then</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p.q: (T_1, \ldots, T_n) )</td>
<td>( p+q, p.q, p&amp;q: (T_1, \ldots, T_n), p \text{ in } q )</td>
</tr>
<tr>
<td>( p: (S_1, \ldots, S_n), q: (T_1, \ldots, T_m), S_n = T_1 )</td>
<td>( p.q: (S_1, \ldots, S_{n-1}, T_2, \ldots, T_m) )</td>
</tr>
<tr>
<td>( p: (S,T) )</td>
<td>( \neg p: (T,S) )</td>
</tr>
<tr>
<td>( p: (T,T) )</td>
<td>( ^p, ^\neg p: (T,T) )</td>
</tr>
<tr>
<td>( p: (T) )</td>
<td>( \text{iden}[p]: (T,T) )</td>
</tr>
<tr>
<td>( p: (T_1, \ldots, T_n) )</td>
<td>( \text{none}[p], \text{univ}[p]: (T_1, \ldots, T_n) )</td>
</tr>
</tbody>
</table>
navigation expressions

from 2-relations and the operators

. + ^ * ~

interpret as path-sets

p.q follow p then q
p+q follow p or q
^p follow p once or more
*p follow p zero or more times
~p follow p backwards

eample

cousin = parent.sibling.^~parent

a navigation example

to say

all messages queued on links emanating from a node have a ‘from’ field of that node

we can write

all n: Node | n.^~source.queue.els.from = n

or equivalently

~source.queue.els.from in iden[Node]
**logical operators**

standard connectives

\[
\begin{align*}
\neg F & \quad \text{not } F \\
F \land G & \quad F \text{ and } G \\
F \lor G & \quad F \text{ or } G \\
F \implies G, H & \quad F \text{ implies } G \text{ else } H \\
F \iff G & \quad F \text{ iff } G
\end{align*}
\]

if-then-else expressions

\[
\text{if } F \text{ then } e \text{ else } e'
\]

negated operators

\[
e \notin e', \ e \neq e'
\]

**set declarations**

form

\[
\text{var} : [\text{set} \mid \text{option}] \text{ setexpr}
\]

meaning

\[
\begin{align*}
v : e & \quad v \in e \text{ and } \#v = 1 \\
v : \text{set} e & \quad v \in e \\
v : \text{option} e & \quad v \in e \text{ and } \#v \leq 1
\end{align*}
\]

examples

\[
\begin{align*}
p : \text{Person} & \quad p \text{ is a scalar in Person} \\
\text{Employee} : \text{set} \text{ Person} & \quad \text{Employee is a subset of Person} \\
bb : \text{Person} \rightarrow \text{Date} & \quad \text{not unary, so no scalar constraint}
\end{align*}
\]
relation declarations

form

\[ \text{var} : \text{expr} \ [\text{mult}] \rightarrow [\text{mult}] \text{expr} \]

multiplicity symbols

? zero or one
!

exactly one
+

one or more

meaning

\[ r: e0 \ m \rightarrow n \ e1 \]
\[ \text{means } r \ \text{in } e0 \rightarrow e1 \]
\[ \text{and } n \ e1's \ for \ each \ e0, \]
\[ m \ e0's \ for \ each \ e1 \]

examples

\[ r: A \rightarrow \mathbf{B} \]

\[ r \text{ is a partial function} \]

\[ r: A \rightarrow \mathbf{B} \]

\[ r \text{ is a total function} \]

\[ r: A \rightarrow \mathbf{B} \]

\[ r \text{ is an injective} \]

\[ r: A \rightarrow \mathbf{B} \]

\[ r \text{ is a bijection} \]

object models

what is an object model?

› set of declarations drawn as graph
› boxes denote sets, arcs relations
› parentless box has implicit type

Person: set PERSON
Company: set COMPANY
Employee: set Person
worksFor: Employee \( \rightarrow \) Company
comprehensions

general form
\[
\{ \text{var} : \text{setexpr} , \ldots | \text{formula} \}
\]

meaning
\[
\{ v0 : e0, v1 : e1, \ldots | F \}
\]
is the relation containing tuples \((a0,a1,\ldots)\)
such that \(F\) holds when \(v0 = \{(a0)\}, v1 = \{(a1)\}, \ldots\) and \(\{(a0)\}\) in \(e0\), \(\{(a1)\}\) in \(e1\), etc

example
\[
\text{ sibling } = \{a, b : \text{Person} | a.\text{parents} = b.\text{parents} \&\& a \neq b\}
\]

quantification

universal quantification
\[
\textbf{all} \ \text{var} : \text{setexpr} , \ldots | \text{formula}
\]

meaning
\[
\textbf{all} \ v0 : e0, v1 : e1, \ldots | F
\]
holds iff \(F\) holds whenever \(v0 = \{(a0)\}, v1 = \{(a1)\}, \ldots\) and \(\{(a0)\}\) in \(e0\), \(\{(a1)\}\) in \(e1\), etc

example
\[
\textbf{all} \ a : \text{Person} | a \notin a.\text{parents}
\]
other quantifiers

quantifiers

all x: e | F  \quad F \text{ holds for all } x \text{ in } e
some x: e | F  \quad F \text{ holds for some } x \text{ in } e
no x: e | F  \quad F \text{ holds for no } x \text{ in } e
sole x: e | F  \quad F \text{ holds for at most one } x \text{ in } e
one x: e | F  \quad F \text{ holds for exactly one } x \text{ in } e

note

all v0: e0, v1: e1,... | F  \quad \text{is equivalent to}
\quad all v0: e0 | all v1: e1 | ... | F

one v0: e0, v1: e1,... | F  \quad \text{is not equivalent to}
\quad one v0: e0 | one v1: e1 | ... | F

quantified expressions

for quantifier Q and expression e, make formula
\[ Q \ e \]

meaning

some e  \quad e \text{ is non-empty}  \quad \#e > 0
no e  \quad e \text{ is empty}  \quad \#e = 0
sole e  \quad e \text{ has at most one tuple}  \quad \#e \leq 1
one e  \quad e \text{ has one tuple}  \quad \#e = 1

example

no Man & Woman  \quad \text{no person is both a man and a woman}
sample quantifications

biological constraints
- all p: Person | one p.mother
- no p: Person | p in p.parents

cultural constraints
- all p: Person | sole p.spouse
- no p: Person | some p.spouse & p.siblings

biblical constraints
- one eve: Person | Person in eve. *~mother

summary: doing more with less

everything's a relation
- a->b in r  for (a, b) ∈ r and a × b ⊆ r

first-order operators
- r : A -> B  means r ⊆ A × B  replaces r ∈ P(A × B)

dot operator
  > plays many roles

intractable  tractable
expressive  inexpressive
Alloy Language and tool

- Main alloy language constructs
- Analysis commands
- Trace pattern
- Examples

Revisiting Paul Simon (1973) song: One Man’s Ceiling Is Another Man’s Floor

sig Platform {}
sig Man {ceiling, floor: Platform}
fact {all m: Man | some n: Man | Above (n,m)}
pred Above (m, n: Man) {m.floor = n.ceiling}

*Simon said "One Man’s Ceiling Is Another Man’s Floor".*
*Does it follow that "One Man’s Floor Is Another Man’s Ceiling"?*
Revisiting Paul Simon (1973) song: One Man’s Ceiling Is Another Man’s Floor

sig Platform {}  
sig Man (ceiling, floor: Platform)  
fact {all m: Man | some n: Man | Above (n,m)}  
pred Above (m, n: Man) {m.floor = n.ceiling}

//will have problem with BelowToo at this stage  
assert BelowToo {all m: Man | some n: Man | Above (m,n)}  
check BelowToo for 4 expect 1

Man1 has no living space,  
is this the problem?

pred Geometry (){no m: Man | m.floor = m.ceiling}  
assert BelowToo’ {Geometry() => all m: Man | some n: Man | Above (m,n)}  
check BelowToo’ for 2 expect 0

//but, still have problem with an increased scope  
check BelowToo’ for 3 expect 1

So, is sharing the problem?
Revisiting Paul Simon (1973) song:
One Man’s Ceiling Is Another Man’s Floor

//Daniel Jackson’s solution – no sharing (see his 2001-1004 example version):

pred NoSharing() {no disj m,n: Man | m.floor = n.floor || m.ceiling = n.ceiling}
assert BelowToo’ {NoSharing() => all m: Man | some n: Man | Above (m,n)}
check BelowToo’ for 6 expect 0
check BelowToo’ for 10 expect 0

//but, this is too restrictive and unreasonable

So, is sharing really the problem?
What about proper sharing

Revisiting Paul Simon (1973) song:
One Man’s Ceiling Is Another Man’s Floor

//Jin Song’s correction (2005) to Daniel’s solution:
//need a living space
fact {no m: Man | m.ceiling = m.floor}
//proper sharing
fact {all disj m,n: Man | m.floor = n.floor iff m.ceiling = n.ceiling}
assert BelowToo’ {all m: Man | some n: Man | Above (m,n)}
check BelowToo’ for 6 expect 0

This is not the end of the story?, the question is: is the artist realistic?
The `acyclic' constraint will show that the artist might not be logical with the current physical reality but could be abstract/imaginary.
Trace pattern:
module util/ordering[elem]

- Creates a single linear ordering over the atoms in elem. It also constrains all the atoms to exist that are permitted by the scope on elem. That is, if the scope on a signature S is 7, opening util/ordering[S] will force S to have 7 elements and create a linear ordering over those 7 elements.

- The predicates and functions below provide access to properties of the linear ordering, such as which element is first in the ordering, or whether a given element precedes another.

module util/ordering[elem]

one sig Ord {
  first_, last_: elem,
  next_, prev_: elem -> lone elem
}

// constraints that actually define the total order
prev_ = ~next_
one first_
one last_
no first_.prev_
no last_.next_

// either elem has exactly one atom, which has no predecessor or successor...
((one elem && no elem.prev_ && no elem.next_) ||
  all e: elem | {
    // ...each element (except the first) has one predecessor, and...
    (e = first_ || one e.prev_)
    // ...each element (except the last) has one successor, and...
    (e = last_ || one e.next_)
    // ...there are no cycles
    (e \in e.^next_))
  // all elements of elem are totally ordered
  elem in first_.*next_.
// first and last
fun first (): elem { Ord.first_ }
fun last (): elem { Ord.last_ }

// return the predecessor of e, or empty set if e is the first element
fun prev (e: elem): lone elem { e.(Ord.prev_) }

// return the successor of e, or empty set of e is the last element
fun next (e: elem): lone elem { e.(Ord.next_) }

// return elements prior to e in the ordering
fun prevs (e: elem): set elem { e.(Ord.prev_) }

// return elements following e in the ordering
fun nexts (e: elem): set elem { e.(Ord.next_) }

// e1 is less than e2 in the ordering
pred lt (e1, e2: elem) { e1 in prevs (e2) }

// e1 is greater than e2 in the ordering
pred gt (e1, e2: elem) { e1 in nexts (e2) }

// e1 is less than or equal to e2 in the ordering
pred lte (e1, e2: elem) { e1 = e2 || lt (e1, e2) }

// e1 is greater than or equal to e2 in the ordering
pred gte (e1, e2: elem) { e1 = e2 || gt (e1, e2) }

// returns the larger of the two elements in the ordering
fun larger (e1, e2: elem): elem { if lt (e1, e2) then e2 else e1 }

// returns the smaller of the two elements in the ordering
fun smaller (e1, e2: elem): elem { if lt (e1, e2) then e1 else e2 }

// returns the largest element in es or the empty set if es is empty
fun max (es: set elem): lone elem { es ^ Ord.prev_ }

// returns the smallest element in es or the empty set if es is empty
fun min (es: set elem): lone elem { es ^ Ord.next_ }
Bridge crossing with one umbrella

A family with a father, a mother, a child and an old grandmother, are going to cross a bridge (a shelter at each end) from the east side to the west side. The bridge is very narrow that allows at most two person to pass at the same time. It is raining hard that they cannot move without an umbrella (they don't want to get wet). However, the whole family has only one umbrella. So some of them need to pass the umbrella back and forth. It takes 1, 2, 5, and 10 minutes for the father, the mother, the child and the grandmother to overpass the bridge respectively, and if two of them walk together, the duration depends on the one who takes longer time.

Father do all the work is the fastest way?

The question is, can this be done less than 19, say 17 mins?
module examples/puzzles/bridgecrossing

open util/ordering[State] as ord
abstract sig Person { time: Int }
one sig Father, Mother, Son, Grandma extends Person {}

fact crossing {int Father.time=1 && int Mother.time=2 && int Son.time=5 && int Grandma.time=10}

sig State {
east: set Person,
west: set Person,
time, u: Int
}
// In the initial state, all objects are on the east side.
fact initialState {
let s0 = ord/first() |
s0.east = Person && no s0.west && int s0.time = 0 && int s0.u=0
}

pred crossBridge (from, from', to, to': set Person, t, t', u, u': Int)
{
(some disj p1, p2: from {
from' = from - p1 - p2
to' = to + p1 + p2
int p1.time > int p2.time => int t'= int t + int p1.time
int p1.time < int p2.time => int t'= int t + int p2.time
int u =0 => int u'=1
int u=1 => int u'=0
})||
(some p: from {
from' = from - p
to' = to + p
int t'= int t + int p.time
int u =0 => int u'=1
int u=1 => int u'=0
})
}
crossRiver transitions between states

fact stateTransition {
    all s: State, s': ord/next(s) {
        s.west != Person =>
            int s.u=0 =>
                crossBridge(s.east, s'.east, s.west, s'.west, s.time, s'.time, s.u, s'.u),
                crossBridge(s.west, s'.west, s.east, s'.east, s.time, s'.time,s.u, s'.u)),
                s' = s
    }
}

dec pred solvePuzzle () {
    ord/last().west = Person && int
    ord/last().time=<17
}

run solvePuzzle for 6 State, 9 Int, 6 int expect 1
Son and gradma

Alloy Applications to Security/Safety Systems

- A relational logic approach for representing secrecy models and detecting their inconsistencies, Waël Hassan, The 14th Nordic Conference on Secure IT Systems, Oct 14 2009
- A Security Domain Model for Implementing Trusted Subject Behaviors, Alan Shaffer, Mikhail Auguston, Cynthia Irvine, and Tim Levin, Workshop on Modeling Security (MODSEC08) held as part of the 2008 International Conference on Model Driven Engineering Languages and Systems (MODELS), Sep 28 2008
Outline

- Introduction
  - Model Checking
  - SPIN
- Modeling Language: Promela
  - Variables
  - Processes
  - Statements
- Verification using SPIN
  - Properties
  - XSPIN
Common Design Flaws

- Deadlock
- Livelock, starvation
- Underspecification
  - unexpected reception of messages
- Overspecification
  - Dead code
- Violations of constraints
  - Buffer overruns
  - Array bounds violations
- Assumptions about speed
  - Logical correctness vs. real-time performance

In designing distributed systems: network applications, data communication protocols, multithreaded code, client-server applications.

Designing concurrent (software) systems is so hard, that these flaws are mostly overlooked...

Fortunately, most of these design errors can be detected using model checking techniques!

Model Checking

State Space

Model M

Property 0

\[ \{ i \mid i < 3 \} \]

Model Checker

M |=

YES, property is satisfied

NO, trace to error
Verification vs. Debugging

- Two (extreme) approaches with respect to the application of model checkers.
  - verification approach: tries to ascertain the correctness of a detailed model $M$ of the system under validation.
  - debugging approach: tries to find errors in a model $M$.
- Model checking is most effective in combination with the debugging approach.

Automatic verification is not about proving correctness, but about finding bugs much earlier in the development of a system.

SPIN (Simple Promela Interpreter)

- One of the most powerful model checkers
  - State of the Art
  - Used by >2000 users
- A tool for analyzing the logical consistency of concurrent systems, specifically of data communication protocols.
  - Promela Language
- Some success factors of SPIN
  - “press on the button” verification (model checker)
  - very efficient implementation (using C)
  - nice graphical user interface (Xspin)
  - not just a research tool, but well supported
  - contains more than two decades research on advanced computer aided verification (many optimization algorithms)
Outline

- Introduction
  - Model Checking
  - SPIN
- Modeling Language: Promela
  - Variables
  - Processes
  - Statements
- Verification using SPIN
  - Properties
  - XSPIN
Promela

Promela (= Protocol/Process Meta Language)
- specification language to model finite-state systems
- loosely based on CSP
  - dynamic creation of concurrent processes
  - communication via message channels can be
    - synchronous (i.e. rendezvous), or
    - asynchronous (i.e. buffered)
- features from Dijkstra’s guarded command language
- features from the programming language C

Promela is a modelling language, not a programming language!

Promela Model (1)

- A Promela model consist of:
  - type declarations
  - channel declarations
  - global variable declarations
  - process declarations
  - [init process]

\[
\text{mtype, constants, typedefs (records)}
\]
\[
\text{chan ch = \{dim\} of \{type, ...\}}
\]
\[
\text{asynchronous: dim > 0}
\]
\[
\text{rendezvous: dim == 0}
\]
\[
\text{- simple vars}
\]
\[
\text{- structured vars}
\]
\[
\text{- vars can be accessed by all processes}
\]
\[
\text{behaviour of the processes: local variables + statements}
\]
\[
\text{initialises variables and starts processes}
\]
Promela Model (2)

- Promela model consist of:
  - type declarations
  - channel declarations
  - variable declarations
  - process declarations
  - \texttt{[init process]}

- A Promela model corresponds with a (usually very large, but) finite transition system, so
  - no unbounded data
  - no unbounded channels
  - no unbounded processes
  - no unbounded process creation

```c
mtype = \{ MSC, ACK \};
chan toS = ...;
chan toR = ...;
bool flag;

\textbf{proctype} Sender() {
  ...
\textcolor{olive}{\texttt{process body}}
}

\textbf{proctype} Receiver() {
  ...
}
\textbf{init} {
  ...
\textcolor{olive}{\texttt{creates processes}}
}
```

Processes (1)

- A \texttt{process} is defined by a \texttt{proctype} definition
- executes concurrently with all other processes, independently of speed or behaviour
- communicates with other processes using \texttt{global (shared) variables} using \texttt{channels}

- There may be several processes of the same type.
- Each process has its own \texttt{local state}:
  - process counter (location within the \texttt{proctype})
  - contents of the \texttt{local variables}
Processes (2)

- A process type (proctype) consist of
  - a name
  - a list of formal parameters
  - local variable declarations
  - body

```proctype Sender(chan in; chan out) {
  bit sndB, rcvB;
  do :: out ! MSG, sndB ->
      in ? ACK, rcvB;
    if :: sndB == rcvB -> sndB = 1-sndB
      else -> skip
    od
}
```

The body consist of a sequence of statements.

Processes (3)

- Process are created using the run statement (which returns the process id).
- Processes can be created at any point in the execution (within any process).
- Processes start executing after the run statement.
- Processes can also be created by adding active in front of the proctype declaration.

```proctype Foo(byte x) {
  ...
}
```

```init {
  int pid2 = run Foo(2);
  run Foo(27);
}
```

number of procs. (opt.)

```active[3] proctype Bar() {
  ...
}
```

parameters will be initialised to 0
Variables and Types (1)

- Five different (integer) basic types.
- Arrays
- Records (structs)
- Type conflicts are detected at runtime.
- Default initial value of basic variables (local and global) is 0.

Basic types
- `bit turn=1;`
- `bool flag;`
- `byte counter;`
- `short s;`
- `int msg;`

Arrays
- `byte a[27];`
- `bit flags[4];`

Array indexing start at 0

Typedef (records)
- `typedef Record {
  short f1;
  byte f2;
} Record rr;
rr.f1 = ...`

Variable declaration

Variables and Types (2)

- Variables should be declared.
- Variables can be given a value by:
  - Initialisation
  - Assignment
  - Argument passing
  - Message passing (see communication)
- Variables can be used in expressions.

```
int ii;
bit bb;
bb=1;
ii=2;
short s=-1;
typedef Foo {
  bit bb;
  int ii,
};
Foo f;
f.bb = 0;
f.ii = -2;
ii*s+27 == 23;
printf("value: %d", s*s);
```

Assignment =

Declaration & initialisation

Equality test ==

Most arithmetic, relational, and logical operators of C/Java are supported, including shift operators.
Statements (1)

- The body of a process consists of a sequence of statements. A statement is either
  - **executable**: the statement can be executed immediately.
  - **blocked**: the statement cannot be executed.
- An **assignment** is always executable.
- An **expression** is also a statement; it is **executable** if it evaluates to non-zero.
  
  2 < 3 always executable
  
  x < 27 only executable if value of \( x \) is smaller 27
  
  3 + x executable if \( x \) is not equal to -3

Statements (2)

- The **skip** statement is always executable.
  - "does nothing", only changes process’ process counter
- A **run** statement is only executable if a new process can be created (remember: the number of processes is bounded).
- A **printf** statement is always executable (but is not evaluated during verification, of course).

```c
int x;
procype Aap()
{
  int y=1;
  skip;
  run Noot();
  x=2;
  x>2 && y==1;
  skip;
}
```
Statements (3)

- **assert**(expr);
  - The **assert**-statement is always executable.
  - If expr evaluates to zero, SPIN will exit with an error, as the expr “has been violated”.
  - The **assert**-statement is often used within Promela models, to check whether certain properties are valid in a state.

```c
proctype monitor() {
  assert(n <= 3);
}

proctype receiver() {
  ...
  toReceiver ? msg;
  assert(msg != ERROR);
  ...
}
```

Hello World

```c
/* A “Hello World” Promela model for SPIN. */
active proctype Hello() {
  printf("Hello process, my pid is: %d\n", _pid);
}

init {
  int lastpid;
  printf("init process, my pid is: %d\n", _pid);
  lastpid = run Hello();
  printf("last pid was: %d\n", lastpid);
}
```

```
$ spin -n2 hello.pr
init process, my pid is: 1
last pid was: 2
Hello process, my pid is: 0
Hello process, my pid is: 2
3 processes created
```
if-statement (1)

```c
if :: choice_i -> stat_{i,1}; stat_{i,2}; stat_{i,3}; ...
:: choice_j -> stat_{j,1}; stat_{j,2}; stat_{j,3}; ...
:: ... ...
:: choice_n -> stat_{n,1}; stat_{n,2}; stat_{n,3}; ...
fi;
```

- If there is at least one `choice_i` (guard) executable, the `if`-statement is executable and GPIN non-deterministically chooses one of the executable choices.
- If no `choice_i` is executable, the `if`-statement is blocked.
- The operator "->" is equivalent to ";". By convention, it is used within `if`-statements to separate the guards from the statements that follow the guards.

if-statement (2)

```c
if :: (n \% 2 != 0) -> n=1
:: (n >= 0) -> n=n-2
:: (n \% 3 == 0) -> n=3
:: else -> skip
fi
```

- The `else` guard becomes executable if none of the other guards is executable.

**give n a random value**

```c
if :: skip -> n=0
:: skip -> n=1
:: skip -> n=2
:: skip -> n=3
fi
```

**skips are redundant, because assignments are themselves always executable...**

**non-deterministic branching**
**do-statement (1)**

```plaintext
do
  :: choice₁ -> stat₁₁; stat₁₂; stat₁₃; ...
  :: choice₂ -> stat₂₁; stat₂₂; stat₂₃; ...
  :: ...
  :: choiceₙ -> statₙ₁; statₙ₂; statₙ₃; ...
od;
```

- With respect to the choices, a `do`-statement behaves in the same way as an `if`-statement.
- However, instead of ending the statement at the end of the chosen list of statements, a `do`-statement repeats the choice selection.
- The (always executable) `break` statement exits a `do`-loop statement and transfers control to the end of the loop.

**do-statement (2)**

- Example – modelling a traffic light

```plaintext
mtype = { RED, AMBER, GREEN };

active proc type TrafficLight() {
  byte state = GREEN;
  do
    :: (state == GREEN) -> state = AMBER;
    :: (state == AMBER) -> state = RED;
    :: (state == RED) -> state = GREEN;
  od;
}
```

- If- and `do`-statements are ordinary Promela statements; so they can be nested.
- `mtype` (message type) models enumerations in Promela
- Note: this `do`-loop does not contain any non-deterministic choice.
Channels (1)

```
proc Sender { 
  s2r!MSG; 
  s2r?MSG; 
  r2s?ACK; 
}

proc Receiver { 
  s2r?MSG; 
  r2s!ACK; 
}
```

Channels (2)

- Communication between processes is via channels:
  - message passing
  - rendez-vous synchronisation (handshake)
- Both are defined as channels:
  ```
  chan <name> = [<dim>] of (<t_1>, <t_2>, ..., <t_n>);
  ```

- Also called: queue or buffer
- Number of elements in the channel: dim=0 is special case: rendez-vous

```
chan ch = [1] of (bit);
chan toR = [2] of (mtype, bit);
chan line[2] = [1] of (mtype, Record);
```
Channels (3)

! sending - putting a message into a channel
  ch ! <expr>, <expr>, ... <expr>;

? receiving - getting a message out of a channel
  ch ? <var>, <var>, ... <var>;
  ch ? <const>, <const>, ... <const>;
  ...

ch!2,27  ch?x,y  ch?2,z  ch?2,2 — will be blocked

Atomic

atomic { stat_1; stat_2; ... stat_n }

- can be used to group statements into an atomic sequence; all statements are executed in a single step (no interleaving with statements of other processes)
- is executable if stat_1 is executable
- if a stat_i (with i>1) is blocked, the “atomicity token” is (temporarily) lost and other processes may do a step

- (Hardware) solution to the mutual exclusion problem:

```c
#include <semaphore.h>

int main()
{
  int flag;

  sem_init(&sem, 0, 1);

  do {
    flag = sem_wait(&sem);
  } while (flag == -1);

  /* do critical section */

  sem_post(&sem);
}
```
**d_step**

```plaintext
\texttt{d\_step \{ \texttt{stat}_1; \texttt{stat}_2; \ldots \texttt{stat}_n \}}
```

- more efficient version of \texttt{atomic}: no intermediate states are generated and stored
- may only contain deterministic steps
- it is a run-time error if \texttt{stat}_i \texttt{i>1} blocks.
- \texttt{d\_step} is especially useful to perform intermediate computations in a single transition

- \texttt{atomic} and \texttt{d\_step} can be used to lower the number of states of the model

**goto**

```plaintext
\texttt{goto label}
```

- transfers execution to \texttt{label}
- each Promela statement might be labelled
- quite useful in modelling communication protocols

```plaintext
:: Rout?i(v) -> d\_step \{
    k++;
    e[k].ind = i;
    e[k].val = v;
    i=0; v=0 ;
\}
```

Part of a model of the “Bounded Retransmission Protocol (BRP)”, an ARP with timers.
Interleaving Semantics

- Promela processes execute concurrently.
- Non-deterministic scheduling of the processes.
- Processes are interleaved (statements of different processes do not occur at the same time).
  - exception: rendez-vous communication.
- All statements are atomic: each statement is executed without interleaving with other processes.
- Each process may have several different possible actions enabled at each point of execution.
  - only one choice is made, non-deterministically.

Example: ABP

- Alternating Bit Protocol
  - To every message, the sender adds a bit.
  - The receiver acknowledges each message by sending the received bit back.
  - The receiver only accepts messages with a bit that it expected to receive.
  - If the sender is sure that the receiver has correctly received the previous message, it sends a new message and it alternates the accompanying bit.
ABP: Prefect Line

```
mtype {MSG, ACK};

chan s2r = [2]of {mtype, bit};
chan r2s = [2]of {mtype, bit};

proctype Sender(chan in, out) {
    bit sendbit, recvbit;
    do
        :: out ! MSG, sendbit ->
            in ? ACK, recvbit;
        if
            :: recvbit == sendbit ->
            sendbit = 1-sendbit
            :: else
                fi
        od
    }  
```  

Outline

- Introduction
- Model Checking
- SPIN
- Modeling Language: Promela
  - Variables
  - Processes
  - Statements
- Verification using SPIN
  - Properties
  - XSPIN
Verification

- Model checking tools automatically verify whether $M|=\phi$ holds, where $M$ is a (finite-state) model of a system and property $\phi$ is stated in some formal notation.

![Xspin: Verification Options](image)

Properties (1)

- Safety property
  - “nothing bad ever happens”
  - invariance $x$ is always less than 5
  - deadlock freedom the system never reaches a state where no actions are possible
  - SPIN: find a trace leading to the “bad” thing. If there is no such trace, the property is satisfied.

- Liveness property
  - “something good will eventually happen”
  - termination the system will eventually terminate
  - response if action X occurs then eventually action Y will occur
  - SPIN: find a (infinite) loop in which the “good” thing does not happen. If there is no such loop, the property is satisfied.
Properties (2)

- LTL formulae are used to specify liveness properties.
  \[ LTL = \text{propositional logic + temporal operators} \]
  - \([\square] p\) always \(p\)
  - \([\diamondsuit] p\) eventually \(p\)
  - \(p \lor q\) \(p\) is true until \(q\) becomes true

- Some LTL patterns
  - invariance \([\square] (p)\)
  - response \([\square] ((p) \rightarrow (\diamondsuit (q)))\)
  - precedence \([\square] ((p) \rightarrow ((q) \lor (x)))\)
  - correlation \((\diamondsuit (p) \rightarrow (\diamondsuit (q)))\)

XSPIN contains a special "LTL Manager" to edit, save and load LTL properties.

Properties (3)

- Three types of labels have a special meaning when SPIN is run in verification mode:
  - end: to specify valid end-states
    Such states might otherwise be flagged as deadlocked states.
  - progress: to mark states that have to be visited in every potentially infinite execution cycle
    If SPIN finds an infinite cycle that does not pass through a progress, SPIN has found a cycle in which no progress has been made: a non-progress cycle.
  - accept: to mark states that should not be part of any potentially infinite execution cycle

Note that such labels just have to start with end, progress or accept.
Xspin in a nutshell

- Xspin allows the user to
  - edit Promela models (+ syntax check)
  - simulate Promela models
    - random
    - interactive
    - guided
  - verify Promela models
    - exhaustive
    - bitstate hashing mode
- additional features
  - Xspin suggest abstractions to a Promela model (slicing)
  - Xspin can draw automata for each process
  - LTL property manager
  - Help system (with verification/simulation guidelines)
SPIN Applications to Security Systems

- ...
Introduction to PAT 3.0
By Dr. SUN, Jun

Table of Content

- Background
  - Model Checking ABC
  - Why PAT?
- System Modeling in Easy in PAT
  - LTS
  - CSP#
- Verification Support in PAT
  - Reachability Analysis
  - Temporal Logic
  - Refinement Checking
Model Checking

- Determining whether a model satisfies a property by the means of exhaustive searching.

How Model Checking Work?

Model Checker

Counterexample!
Model Checking Works

- Three researchers won Turing Award 2007 for their pioneer work on model checking!
- Intel i7 processor is verified by symbolic model checking completely without a single test case!
  - 8 cores, millions of registers; functional verification!
- The Slam project from Microsoft successfully detected many bugs in many driver software!
  - Dozens of K lines of C codes; debugging.

How do we replicate the success of SLAM and Intel i7 everywhere?

PAT: Motivation

- Assume that you have interesting problems to be solved by model checking.
- Method A
  - Translate your system into a model of an existing model checker.
  - Translate your problem into the problem of verifying certain property against the model.
  - Interpreter and reflect model checking results back to your original problem.
Assume that you have interesting problems to be solved by searching.

Method B: Build a dedicated model checker with
- Dedicated modeling language
- Dedicated model checking algorithms

Building a model checker is highly nontrivial!
PAT: Pervasive Model Checking

- Distributed Algorithms, Web Services, Bio-systems, Security Protocols, Sensor Networks, etc.

| Modeling | Concurrent Module | Real-time Module | Web Service Module | Bio-system Module | ...
|-----------|-------------------|------------------|--------------------|------------------|--------
| Compiling | Domain-specific, Abstraction, data abstraction, zone abstraction, environment abstraction, etc. | Labeled Transition System (or Markov Decision Processes) | 
| System Analysis | Reachability Analysis, LTL Model Checking, Refinement Checking, Probabilistic Model Checking, etc. | Simulator |

Where Are We after 2.5 Years

- More than 600K lines of C# codes!
- 800+ registered users from 130+ organizations
- 20+ peer-reviewed conference papers (including CAV, FM) + multiple journal submissions
- Commercializing in Japan
- Adopted for teaching in multiple universities
- …
Background
- Model Checking ABC
- Why PAT?

System Modeling in Easy in PAT
- LTS
- CSP#

Verification Support in PAT
- Reachability Analysis
- Temporal Logic
- Refinement Checking

What is a model?
- A model is a finite-state transition system.
Where Models Come From?

- From user requirements
  - Building the model and verifying it help in identifying problems with the requirements
- From system design/specification
- From actual systems
  - For hardware, it’s more or less natural.
  - For software, it’s highly non-trivial!
    - The SLAM project builds an abstract model from driver software by data abstraction.
    - The SLAM project handles 10K lines of C codes.

CSP# Model

- Variables/Channels
  ```csharp
  var x = 5;
  var y[m][n];
  channel ch 1;
  ```

- Processes
  ```csharp
  System = ComponentA || ComponentB;
  ComponentA = ...
  ComponentB = ...
  ```
What is a state?

- A state in the model should capture all varying things in the systems.
  - Two states are different if and only if the system are in different configuration at the states.
- What is a state in a PAT model?
  - Valuation of the variables, current process expression, and contents of communication channel buffers.
- What is a state in a program?
  - Valuation of variables, PC, system stack, hard disk, etc.

What is a transition?

- A transition in the model is caused by some action which changes system configurations.
- What is an action in PAT?
  - Performing an event,
  - Performing an event while updating variables,
  - Channel input/output.
- What is an action in program?
  - Executing of one statement
    - What is “one statement” is related to the level of atomicity!
A state consists of the positions of the black one and the white ones. Initially, it is
- Black at (3,0); Whites at (2,3), (3,1), (3,5), (4,3)

A transition is caused by the movement of the black.
Non-determinism

- How is a PAT model different from a program?
  - There are many behaviors of a model due to non-determinism, e.g., choices.
  - A program has multiple behaviors if you consider different environment inputs or different system scheduling if it is concurrent.

The number of system states must be finite!

Table of Content

- Background
  - Model Checking ABC
  - Why PAT?
- System Modeling in Easy in PAT
  - LTS
  - CSP#
- Verification Support in PAT
  - Reachability Analysis
  - Temporal Logic
  - Refinement Checking
#import "PAT.Lib.Example";

#define NoOfFloors 2;
#define NoOfLifts 2;
#define NoOfUsers 2;

var extrequestsUP[NoOfFloors];
var extrequestsDOWN[NoOfFloors];
var intrequests[NoOfLifts][NoOfFloors];

door = [-1(NoOfLifts)]; //initiate an array of -1 with length NoOfLifts

LiftSystem() = (||| {NoOfUsers} @ User()) ||| (||| x:{0..NoOfLifts-1} @ Lift(x, 0, 1));

User() = []pos:{0..NoOfFloors-1}@ (ExternalPush(pos); UserWaiting(pos));

ExternalPush(pos) = case {
pos == 0 : pushup.pos{extrequestsUP[pos] = 1;} -> Skip
-pos == NoOfFloors-1 : pushdown.pos{extrequestsDOWN[pos] = 1;} -> Skip
-default : pushup.pos{extrequestsUP[pos] = 1;} -> Skip
[
] pushdown.pos{extrequestsDOWN[pos] = 1;} -> Skip,
};

UserWaiting(pos) = [] i:{0..NoOfLifts-1} @
([door[i] == pos]enter.i -> ([
]y:{0..NoOfFloors-1}@(push.y{intrequests[i][y] = 1;} ->
([door[i] == y]exit.i -> User())));

Lift(i, level, direction) =
if (intrequests[i][level] != 0 || (direction == 1 && extrequestsUP[level] == 1) || (direction == -1 && extrequestsDOWN[level] == 1)) {
opendoor.i.level{
door[i] = level; intrequests[i][level] = 0;
if (direction > 0) {
extrequestsUP[level] = 0;
} else {
extrequestsDOWN[level] = 0;
}
} -> close.i.level{door[i] = -1;} -> Lift(i, level, direction)
} else {
checkIfToMove.i.level ->
if (call(CheckIfToMove, level, direction, i, NoOfFloors, intrequests, extrequestsUP, extrequestsDOWN)) {
moving.i.level.direction ->
if (level+direction == 0 || level+direction == NoOfFloors-1) {
Lift(i, level+direction, -1*direction)
} else {
Lift(i, level+direction, direction)
}
} else {
if ((level == 0 && direction == 1) || (level == NoOfFloors-1 && direction == -1)) {
Lift(i, level, direction)
} else {
changedir.i.level -> Lift(i, level, -1*direction)
}
}
}
Modeling is vital to the success of the system development.
- The choice of the modeling language is essential!
- PAT supports CSP# for a system with non-determinism, concurrency, data operations, etc.

CSP# = Hoare’s CSP + Data + Assertions + Real-time + Probability

A model contains
- Constants
  ```c
  #define max 5;
  #define allpositive (x > 0 && y > 0 && z > 0);
  ```
- Variables
  - Boolean, integer (default), (multiple dimensional) arrays of Boolean or integer, user-defined data type, etc.
- Processes
- Assertions
  ```c
  #assert P deadlockfree;
  #assert Q reaches allpositive;
  #assert Q |= []<> allpositive;
  ```
Process Expressions

<table>
<thead>
<tr>
<th>Process Expression</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stop</td>
<td>To do nothing</td>
</tr>
<tr>
<td>Skip</td>
<td>To terminate</td>
</tr>
<tr>
<td>P</td>
<td></td>
</tr>
<tr>
<td>[b]P</td>
<td>To do P if b is true; otherwise, wait</td>
</tr>
<tr>
<td>P</td>
<td></td>
</tr>
<tr>
<td>P</td>
<td></td>
</tr>
<tr>
<td>a{x=x+1,y=x;} -&gt; P</td>
<td>To do event a first and then do P</td>
</tr>
<tr>
<td>P interrupt Q</td>
<td>To do P until the first event of Q occurs and then behave as Q</td>
</tr>
</tbody>
</table>

Demonstration

- Sliding game problem
- Multi-lifts System
Table of Content

- Background
  - Model Checking ABC
  - Why PAT?
- System Modeling in Easy in PAT
  - LTS
  - CSP#
- Verification Support in PAT
  - Reachability Analysis
  - Temporal Logic
  - Refinement Checking

System Verification

```
#include "PAT.Lib.Example";
#define NoOfFloors 2;
#define NoOfLifts 2;
#define NoOfUsers 2;
var extrequestsUP[NoOfFloors];
var extrequestsDOWN[NoOfFloors];
var intrequests[NoOfLifts][NoOfFloors];
door = [-1(NoOfLifts)]; //initiate an array of -1 with length NoOfLifts

if (System()) = (||| {NoOfUsers} @ User()) ||| (||| x : {0..NoOfLifts-1} @ Lift(x, 0, 1));
User() = []pos:{0..NoOfFloors-1}@ (ExternalPush(pos); UserWaiting(pos));
ExternalPush(pos) = case {
  pos == 0 : pushup.pos{extrequestsUP[pos] = 1;} -> Skip
  pos == NoOfFloors-1 : pushdown.pos{extrequestsDOWN[pos] = 1;} -> Skip
  default : pushup.pos{extrequestsUP[pos] = 1;} -> Skip
  [] pushdown.pos{extrequestsDOWN[pos] = 1;} -> Skip
};
UserWaiting(pos) = [] i:{0..NoOfLifts-1} @
  ([]door[i] == pos]enter.i -> (
    [] y:{0..NoOfFloors-1}@ (push.y{intrequests[i][y] = 1;} ->
    ([]door[i] == y]exit.i -> User()));
Lift(i, level, direction) =
  if (intrequests[i][level] != 0 | | (direction == 1 && extrequestsUP[level] == 1) | | (direction == -1 && extrequestsDOWN[level] == 1)) {
    opendoor.i.level{
      door[i] = level; intrequests[i][level] = 0;
      if (direction > 0) {
        extrequestsUP[level] = 0;
      } else {
        extrequestsDOWN[level] = 0;
      }
    } -> close.i.level{door[i] = -1;} -> Lift(i, level, direction)
  } else {
    checkIfToMove.i.level ->
      if (call(CheckIfToMove, level, direction, i, NoOfFloors, intrequests, extrequestsUP, extrequestsDOWN)) {
        moving.i.level.direction ->
          if (level+direction == 0 || level+direction == NoOfFloors-1) {
            Lift(i, level+direction, -1*direction)
          } else {
            Lift(i, level+direction, direction)
          }
      } else {
        if ((level == 0 && direction == 1) || (level == NoOfFloors-1 && direction == -1)) {
          Lift(i, level, direction)
        } else {
          changedir.i.level -> Lift(i, level, -1*direction)
        }
      }
  }
#define liveness extrequestsUP[0] == 0;
#define assert LiftSystem() deadlockfree;
#define assert LiftSystem() | |= []<> liveness;
```
Property

- Reachability
  - #assert P reaches goal;
- Temporal logic
  - #assert P |= [] goal;
  - #assert P |= []<> goal;
  - #assert P |= [] (cond U goal);
- Refinement relationship
  - #assert P refines Q;
  - #assert P refines <F> Q;

Reachability Analysis

- Goal: to determine whether there is a reachable state such that certain condition is satisfied.
  - e.g., searching for a state such that the white ones are at (2,2), (2,3), (3,2), (3,3)
Reachability Analysis: Methods

- Depth-First-Search
  - Default in PAT
  - Potential memory saving
  - Potentially longer witness trace
- Breadth-First-Search
  - Enabled by ticking “Shortest Witness Trace”
  - Produces shortest witness trace

Temporal Logic

- A property is captured in the form of temporal logic formulae.
  - Linear Temporal Logic – There is only one future!
  - Computation Tree Logic – There are multiple futures!
  - CTL*, TCTL, PCTL, etc.
- A LTL formula is a proposition extended with the following operators.
  - [] which reads “always”
  - <> which reads “eventually”
  - U which reads “until”
Temporal Logic: Examples

- \( \Box \) not (the white ones are at (2,2), (2,3),(3,2), (3,3))
  - If this property is false, a counterexample is a solution of the shunting game.
- \(<>\Box\) (one and only one leader)
  - This is a desired property for leader election protocols.
- \(\Box<>\) (a request for an elevator is served)
  - This is a desired property for a multi-lifts system

Temporal Logic: Methods

- A lot of dedicated research is evolved in developing efficient algorithms for verifying temporal logic formulae.
- For LTL model checking, it is reduced to the problem of searching for a loop in the system graph.
  - For \(\Box<>\) (a request for an elevator is served), a counterexample is a loop during which the request is never served.
Temporal Logic: Methods

- Property: no body starves to death $== []<>$ every philosopher eats.
- Fairness needed (see lift system example as well)!

Refinement Checking

- A model (in the same language) is used to capture the property and then the property is verified by showing a refinement relationship from the system model to the property model.
  - The system model describes an algorithm for mutual exclusion.
  - The property model describes all good behaviors – no violation of mutual exclusion.
  - The property is verified by showing that the set of all behaviors of the system model is a subset of those of the property model.
System Model
System = (Proc(0) ||| Proc(1)) || Semaphore;
Proc(i) = req -> cs.i -> rel -> Proc(o);
Semaphore = req -> rel -> Semaphore;

Property Model
Property = req -> cs.0 -> rel -> Property
[] req -> cs.1 -> rel -> Property

If traces(System) is a subset of traces(Property), then System satisfies mutual exclusion.

Refinement More Examples

How to verify a pacemaker system?
- The system is the composition of the pacemaker and a troubled heart.
- The specification/property is a normal heart.

How to verify a concurrent stack?
- The system is a concurrent stack.
- The specification/property is a standard (sequential) stack.
Table of Content

- Background
  - Model Checking ABC
  - Why PAT?
- System Modeling in Easy in PAT
  - LTS
  - CSP#
- Verification Support in PAT
  - Reachability Analysis
  - Temporal Logic
  - Refinement Checking

Exercise
SPIN was developed at Bell lab.
SPIN is the most popular model checker!
SPIN is initially designed for formal verification of communicating protocols.
   ◦ The input language is Promela.
SPIN works by generating model-specific C programs which contains all possible system scheduling.
PAT vs SPIN: Modeling

- A SPIN model is the parallel composition of multiple finite-state machines.
- PAT supports more than one modeling language.
  - LTS – closest to Promela
  - CSP# – fully hierarchical
  - RTS – CSP# + real-time
  - PCSP# – CSP# + probabilistic choices
  - PRTS – CSP# + real-time + probabilistic choices

PAT vs SPIN: Efficiency

- For systems which are modeled in Promela and CSP#, SPIN is probably faster in same cases – not always! Why?
  - SPIN generates model specific C programs – optimization may be applied during the process.
  - A configuration in SPIN is of the form (V, Sp1, Sp2, …) where V is the variable valuation; Sp1 is the state of the process 1; Sp2 is the state of the process 2.
  - A configuration in PAT with CSP# is of the form (V, P) such that P is a process.
PAT vs SPIN: Efficiency (cont’d)

- SPIN supports
  - Partial order reduction
  - Parallel model checking
- PAT supports
  - Partial order reduction – not any more!
    - Manual partial order reduction through keyword `atomic`
  - Symmetry reduction
  - BDD – finished for LTS, in progress for CSP#

PAT vs SPIN: Others

- SPIN supports LTL whereas PAT supports SE-LTL.
- SPIN supports arrays of channels, and channels inquiry functions, which PAT does not.
- SPIN supports embedded C program (in an odd way), whereas PAT supports arbitrary C# data types or static methods.
- PAT supports refinement checking whereas SPIN does not.
- PAT supports a variety of fairness whereas SPIN does not.
**PAT vs SPIN: GUI**

- Once you tried both PAT and xSPIN, you will know.

**SPIN is probably more efficient for networks of concurrent processes and LTL properties without fairness; If your system is hierarchical, timed, or probabilistic, or your property is liveness requiring fairness, or you like better simulation, go for PAT.**

---

**Alloy Analyzer May 21st 2006**

- Alloy analyzer was developed at MIT.
- Alloy analyzer essentially solves a different problem.
  - Model checking: given a model, check whether the model satisfies certain property.
  - Alloy analyzer: given a set of constraints (in the form of a declarative model), check whether there exists a model which satisfies the constraints.
- Alloy analyzer’s problem is harder whereas model checking more practical.

**If you have a model checking problem, go for PAT.**
FDR 2.83 July 23 2007

- FDR was developed at Oxford university.
- FDR is a dedicated model checker for Hoare’s CSP.
  - FDR models fully hierarchical systems.
  - It’s semantics is mostly compositional.
  - All inter-process communication is through barrier synchronization!
  - FDR verifies properties by establishing refinement relationship.

PAT vs FDR: Modeling

- PAT supports inter-process communication through
  - Barrier synchronization
  - Shared variables
  - Synchronous/asynchronous channel communication
- FDR doesn’t support shared variables!
- PAT supports all assertions which are supported by FDR.
PAT vs FDR: Efficiency

- FDR incrementally compile and compress processes.
  - It could check $10^{20}$ dining philosophers if you model it correctly!
  - It supports other kinds of optimization including bi-simulation reduction and a partial order reduction like tau-transition reduction.
  - If the above optimizations do not work, PAT probably is fast.

Unless you are a real expert of CSP, be careful should you choose FDR.

LTSA V3.0 June 2006

- LTSA was developed at Imperial college.
- LTSA is very much related to FDR or process algebra, yet it has its own features.
  - LTSA has no supports for variables.
  - LTSA supports a simple form of timing.
  - ...

LTSA comes from a less formal background.
**NuSMV 2.5**

- NuSMV is jointly developed by multiple universities.
- Its input language was designed for hardware circuits.
  - It supports hierarchical systems in a different way.
- It is based on symbolic model checking techniques.
  - BDD
  - SAT solving

NuSMV is designed for symbolic model checking.

**UPPAAL 4.0**

- UPPAAL was developed jointly at Uppsala University of Aalborg University.
- The input language of UPPAAL is Timed Safety Automata
  - Networks of finite-state automata with clocks
- The property language of UPPAAL is a restrictive subset of Timed CTL.

UPPAAL is designed and optimized for simple real-time systems.
PRISM 3.3.1 Nov 22 2009

- PRISM was developed at University of Birmingham and now at Oxford University.
- The input language of PRISM is a simple state-based language with probabilistic distributions
  - Markov chain, Markov Decision Process, CTMC

PRISM is designed and optimized for simple probabilistic systems.

PAT VS

<table>
<thead>
<tr>
<th>Tool</th>
<th>Fully Hierarchical Systems</th>
<th>Complex Data Operations</th>
<th>Real-Time</th>
<th>Probability</th>
<th>Concurrency</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPIN</td>
<td>o</td>
<td>*</td>
<td>o</td>
<td>o</td>
<td>x</td>
</tr>
<tr>
<td>UPPAAL</td>
<td>o</td>
<td>*</td>
<td>x</td>
<td>o</td>
<td>x</td>
</tr>
<tr>
<td>PRISM</td>
<td>o</td>
<td>x</td>
<td>*</td>
<td>*</td>
<td>x</td>
</tr>
<tr>
<td>NuSMV 2</td>
<td>x</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>x</td>
</tr>
<tr>
<td>FDR</td>
<td>x</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>x</td>
</tr>
</tbody>
</table>

PAT is more than one model checker, rather it is a framework for realizing system verification techniques.