PURDUE
UNIVERSITY

Principles of Concurrency

Lecture 21
Model-Checking

Verification vs. Testing

- Testing provides assurance by exploring (sampling) points in an application's input space
  - Premised on the assumption that this sample is representative of the entire space
  - Challenge is determining how best to define the sampler

- Verification provides assurance by demonstrating that a property holds for *all* elements in the input space
  - Proofs provide stronger guarantees than tests
  - But, the challenge is identifying a proof method that scales (e.g., even in the presence of an unbounded input space)

Properties

- Application is free from deadlock
- Application if free from livelock, starvation
- Application satisfies performance (realtime) guarantees
- There is no path through the application's control-flow that results in an assertion violation
- Application adheres to a protocol specification

Hard to realize in a sequential setting

Exponentially harder in a concurrent setting because
of the additional (non-deterministic) interleavings that manifest among threads

Model Checking

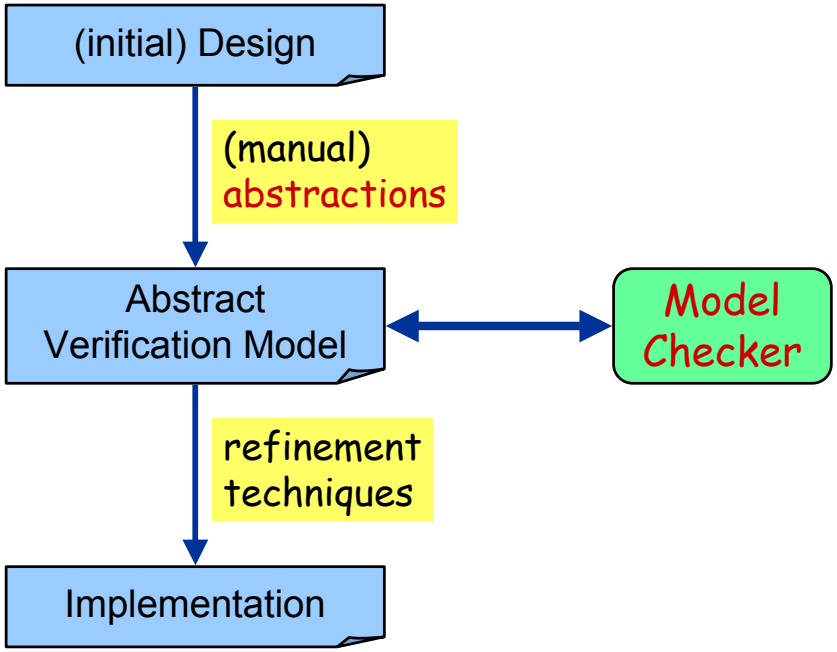
Model checking is an automated technique that, given a finite-state model of a system and a logical property, systematically checks whether this property holds for (a given initial state in) that model. (Clarke and Emerson, 1981)

Goal: verify $M \models \phi$

where M is a finite-state model and ϕ is a property stated in some formal logic

Problem: exponential state-space explosion

Verification



- Prove the correctness of the model with respect to an implementation
- Alternatively, find errors in the model prior to implementation

SPIN

IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 23, NO. 5, MAY 1997

The Model Checker SPIN

Gerard J. Holzmann

Abstract—SPIN is an efficient verification system for models of distributed software systems. It has been used to detect design errors in applications ranging from high-level descriptions of distributed algorithms to detailed code for controlling telephone exchanges. This paper gives an overview of the design and structure of the verifier, reviews its theoretical foundation, and gives an overview of significant practical applications.

Index Terms—Formal methods, program verification, design verification, model checking, distributed systems, concurrency.

1 INTRODUCTION

SPIN is a generic verification system that supports the design and verification of asynchronous process systems [36], [38]. SPIN verification models are focused on proving the correctness of *process interactions*, and they attempt to abstract as much as possible from internal sequential computations. Process interactions can be specified in SPIN with rendezvous primitives, with asynchronous message passing through buffered channels, through access to shared variables, or with any combination of these. In focusing on asynchronous control in software systems, rather than synchronous control in hardware systems, SPIN distinguishes itself from other well-known approaches to model checking, e.g., [12], [49], [53].

As a formal methods tool, SPIN aims to provide:

1) an intuitive, program-like notation for specifying design choices unambiguously, without implementation detail,
2) a powerful, concise notation for expressing general correctness requirements, and
3) a methodology for establishing the logical consistency of the design choices from 1) and the matching correctness requirements from 2).

Many formalisms have been suggested to address the first two items, but rarely are the language choices directly related to a basic feasibility requirement for the third item. In SPIN the notations are chosen in such a way that the logical consistency of a design can be demonstrated mechanically by the tool. SPIN accepts design specifications written in the verification language PROMELA (a Process Meta Language) [36], and it accepts correctness claims specified in the syntax of standard Linear Temporal Logic (LTL) [60].

There are no general decision procedures for unbounded systems, and one could well question the soundness of a design that would assume unbounded growth. Models that can be specified in PROMELA are, therefore, always required to be bounded, and have only countably many distinct behaviors. This means that all correctness properties automatically become formally decidable, within the constraints that are set by problem size and the computational resources that are available to the model checker to render the proofs. All verification systems, of course, do have physical limitations that are set by problem size, machine memory size, and the maximum runtime that the user is willing, or able, to endure. These constraints are an often neglected issue in formal verification. We study the limitations of the model checker explicitly and offer relief strategies for problems that are outside the normal domain of exhaustive proof. Such strategies are discussed in Sections 3.3 and 3.4 of this paper

1.1 Structure

The basic structure of the SPIN model checker is illustrated in Fig. 1. The typical mode of working is to start with the specification of a high level model of a concurrent system, or distributed algorithm, typically using SPIN's graphical front-end XSPIN. After fixing syntax errors, interactive simulation is performed until basic confidence is gained that the design behaves as intended. Then, in a third step, SPIN is used to generate an optimized on-the-fly verification program from the high level specification. This verifier is compiled, with possible compile-time choices for the types of reduction algorithms to be used, and executed. If any counterexamples to the correctness claims are detected, these can be fed back into the interactive simulator and inspected in detail to establish, and remove, their cause.

The remainder of this paper consists of three main parts. Section 2 gives an overview of the basic verification method that SPIN employs. Section 3 summarizes the basic algorithms and complexity management techniques that have been implemented. Section 4 gives three examples of typical applications of the SPIN model checker to design and verification problems. The first example is the problem of devising a correct process scheduler for a distributed operating system; the second problem is the verification of a leader election protocol for a distributed ring; the third problem is the proof of correctness of a standard sliding window flow control protocol. Section 4 concludes with a summary of a range of other significant verification problems to which SPIN has been applied. Section 5 concludes the paper.

• *G.J. Holzmann is with the Computing Sciences Research Center, Bell Laboratories, Murray Hill, NJ 07974. Email: gerard@research.bell-labs.com.*

Manuscript received Sept. 30, 1996.
Recommended for acceptance by L.K. Dillon and S. Sankar.
For information on obtaining reprints of this article, please send e-mail to: transse@computer.org, and reference IEEECS Log Number 104928.0.

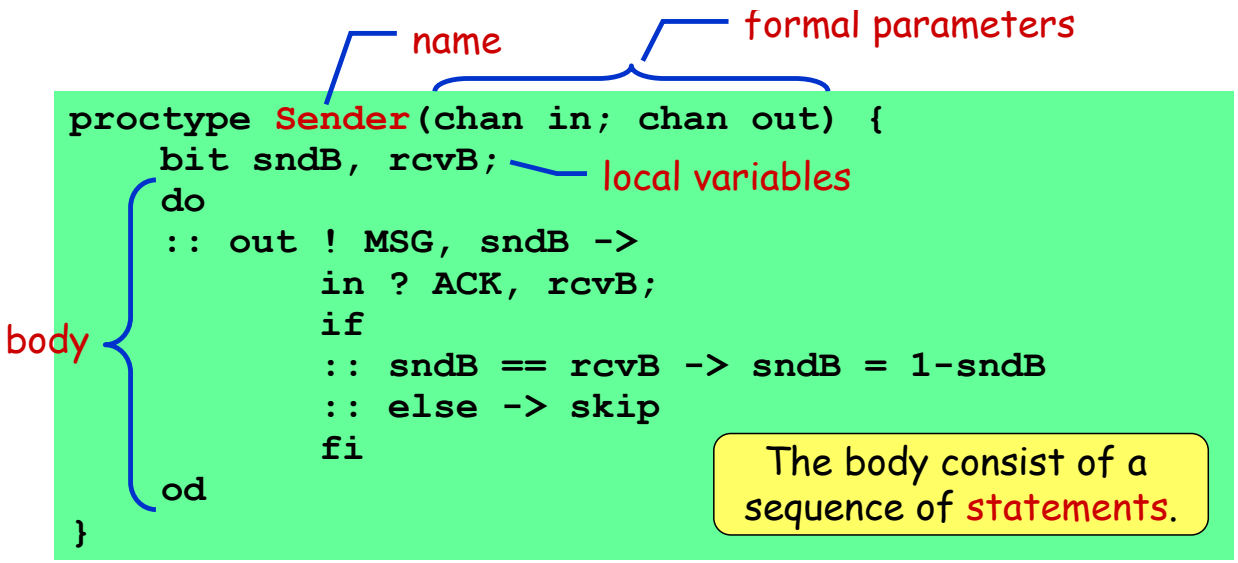
- An automated tool for checking the logical consistency of asynchronous systems
- Uses a specification/modeling language called Promela (Protocol/Process MetaLanguage)
  - communication via messaging
    - synchronous
    - asynchronous

Promela model

A model consists of:

- Declarations (types, channel, processes)
- Definition of a finite-state transition system
  - no notion of "unboundedness"

Process
- local-state
- communication via channels and global variables
- can be created arbitrarily



Process Statements

- The body of a process consists of a sequence of statements
- A statement is either executable or blocked
- Assignment and assert statements are always executable

```
int x;
proctype Aap()
{
  int y=1;
  skip;
  run Noot();
  x=2;
  x>2 && y==1;
  skip;
}
```

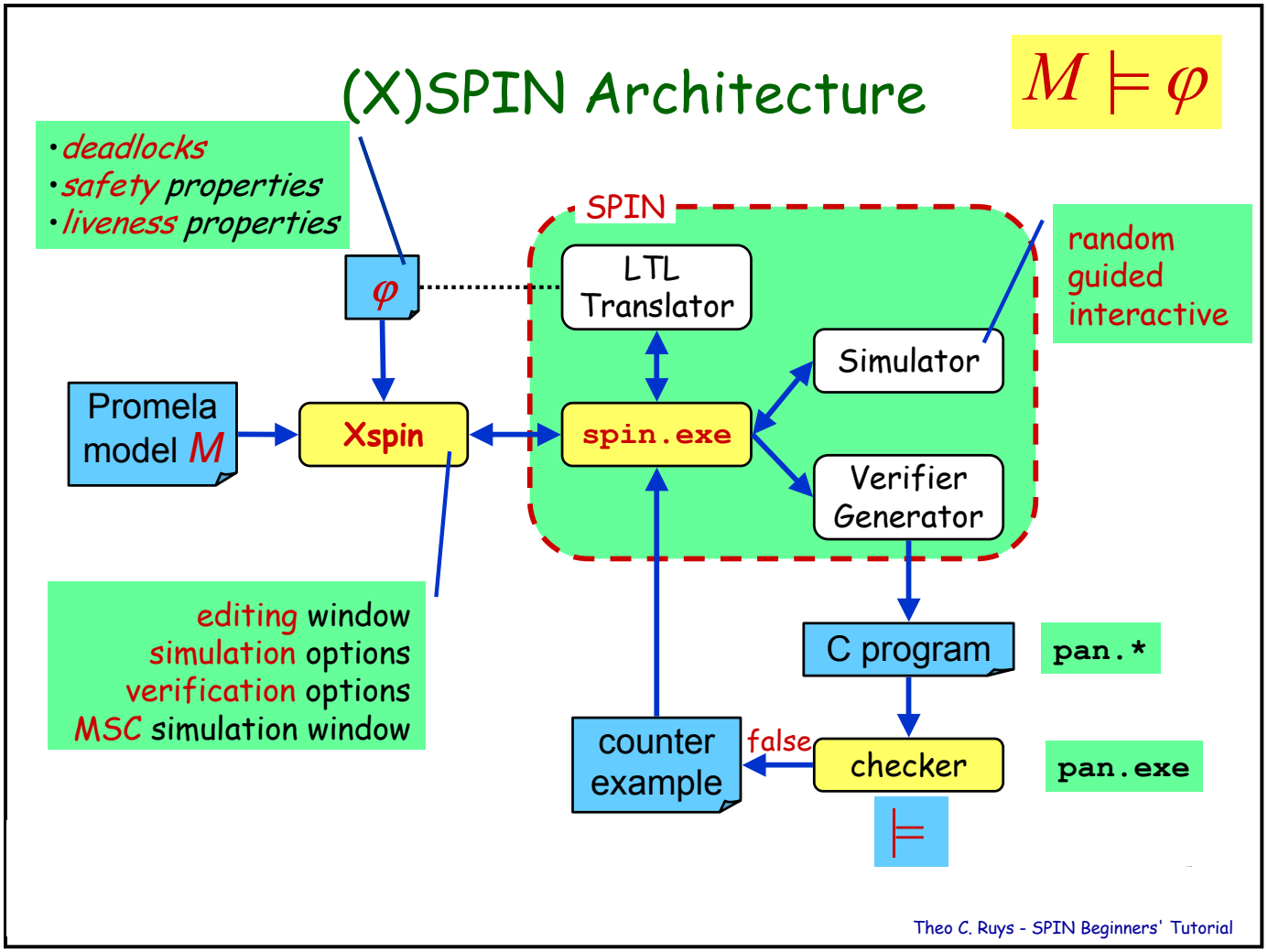
Executable if `Noot` can be created…

Can only become executable if a some other process makes `x` greater than 2.

Statements

skip	always executable
assert(*<expr>*)	always executable
expression	executable if not zero
assignment	always executable
if	executable if at least one guard is executable
do	executable if at least one guard is executable
break	always executable (exits **do**-statement)
send **(ch!)**	executable if channel **ch** is not full
receive **(ch?)**	executable if channel **ch** is not empty

Architecture



Mutual Exclusion


```
bit  flag;    /* signal entering/leaving the section */
byte mutex;   /* # procs in the critical section.    */

proctype P(bit i) {
  flag != 1;
  flag  = 1;
  mutex++;
  printf("MSC: P(%d) has entered section.\n", i);
  mutex--;
  flag  = 0;
}

proctype monitor() {
  assert(mutex != 2);
}

init {
  atomic { run P(0); run P(1); run monitor(); }
}
```

models:
`while (flag == 1) /* wait */;`

Problem: assertion violation!
Both processes can pass the `flag != 1` "at the same time", i.e. before `flag` is set to 1.

starts two instances of process P

Theo C. Ruys - SPIN Beginners' Tutorial

Mutual Exclusion


```
bit  x, y;     /* signal entering/leaving the section  */
byte mutex;    /* # of procs in the critical section.  */

active proctype A() {
  x = 1;
  y == 0;
  mutex++;
  mutex--;
  x = 0;
}

active proctype B() {
  y = 1;
  x == 0;
  mutex++;
  mutex--;
  y = 0;
}

active proctype monitor() {
  assert(mutex != 2);
}
```

Process A waits for process B to end.

Problem: invalid-end-state!
Both processes can pass execute `x = 1` and `y = 1` "at the same time", and will then be waiting for each other.

Theo C. Ruys - SPIN Beginners' Tutorial

Mutual Exclusion


```
bit  x, y;     /* signal entering/leaving the section  */
byte mutex;    /* # of procs in the critical section.  */
byte turn;     /* who's turn is it?                    */

active proctype A() {
  x = 1;
  turn = B_TURN;
  y == 0 ||
    (turn == A_TURN);
  mutex++;
  mutex--;
  x = 0;
}

active proctype B() {
  y = 1;
  turn = A_TURN;
  x == 0 ||
    (turn == B_TURN);
  mutex++;
  mutex--;
  y = 0;
}

active proctype monitor() {
  assert(mutex != 2);
}
```

Can be generalised to a single process.

First "software-only" solution to the mutex problem (for two processes).

Theo C. Ruys - SPIN Beginners' Tutorial

Mutual Exclusion


```
byte turn[2];  /* who's turn is it?              */
byte mutex;    /* # procs in critical section */

proctype P(bit i) {
  do
  :: turn[i] = 1;
     turn[i] = turn[1-i] + 1;
     (turn[1-i] == 0) || (turn[i] < turn[1-i]);
     mutex++;
     mutex--;
     turn[i] = 0;
  od
}

proctype monitor() { assert(mutex != 2); }
init { atomic {run P(0); run P(1); run monitor()}}
```

Problem (in Promela/SPIN): `turn[i]` will overrun after `255`.

More mutual exclusion algorithms in (good-old) [Ben-Ari 1990].

Theo C. Ruys - SPIN Beginners' Tutorial

Safety and Liveness

Safety:

- "Nothing bad ever happens"
- invariants
- deadlock-freedom
- Model-checker: try to find a violating trace

Liveness:

- "Something good eventually happens"
- termination
- reactivity
- Model-checker: search for an infinite loop in which something good does not happen

Implementation

- SPIN uses a depth first search algorithm (DFS) to generate and explore the complete state space.

```
procedure dfs(s: state) {
  if error(s)
      reportError();
  foreach (successor t of s) {
      if (t not in Statespace)
          dfs(t);
  }
```

states are stored in a hash table

requires state matching

Only works for state properties.

the old states s are stored on a stack, which corresponds with a complete execution path

- Note that the construction and error checking happens at the same time: SPIN is an on-the-fly model checker.

Theo C. Ruys - SPIN Beginners' Tutorial

Reduction

SPIN combats exponential search space using a number of reduction techniques:

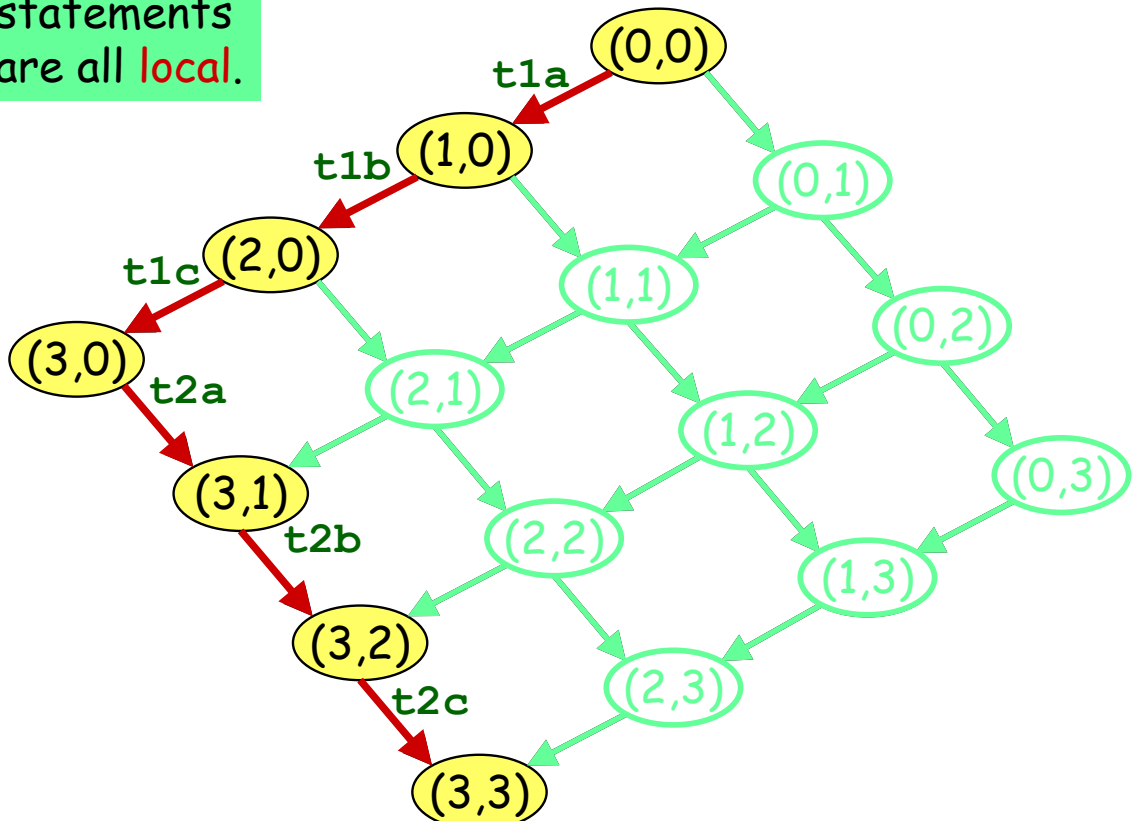
- partial order reduction
- bitstate hashing
- state vector compression
- dataflow analysis
- slicing

Partial order reduction:

- intuition: exploit conditions in which interleaved operations do not affect final outcome (e.g., commutativity)
- Example: if in state, process P executes only "local" statements, then the actions of other processes can be deferred until P finishes
- local: access only local variables
- receive or send data from/to an exclusive queue

Example

Suppose the statements of P1 and P2 are all local.



Liveness Specifications

LTL formulae: propositional formula with temporal operators:

- $\Box$ P - formula P is true now and forever into the future
- $\Diamond$ P - formula P is satisfied at some point in the future

Linear temporal logic refers to the underlying nature of time and the choices possible in the future:

- linear: each point in time has a well-defined successor
- branching: each point in time has multiple possible futures

- think of time in terms of ordering of events

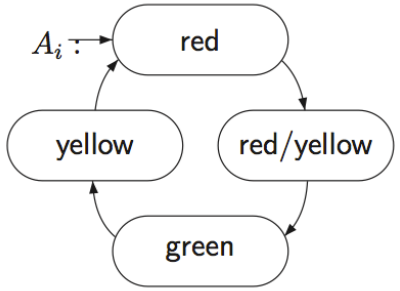
- $p \rightarrow \Diamond q$ p implies eventually q (response)
- $p \rightarrow q \; U \; r$ p implies q until r (precedence)
- $\Box\Diamond p$ always eventually p (progress)
- $\Diamond\Box p$ eventually always p (stability)
- $\Diamond p \rightarrow \Diamond q$ eventually p implies eventually q (correlation)

- Eventually $\Diamond\phi := true \; U \; \phi$
- Always $\Box\phi := \neg\Diamond\neg\phi$

Example

System description

- Focus on lights in on particular direction
- Light can be any of three colors: green, yellow, read
- Atomic propositions = light color



Ordering specifications

- Liveness: “traffic light is green infinitely often”

  $\Box\Diamond$green

- Chronological ordering: “once red, the light cannot become green immediately”

  $\Box$ (red $\rightarrow \neg \bigcirc$ green)

- More detailed: “once red, the light always becomes green eventually after being yellow for some time”

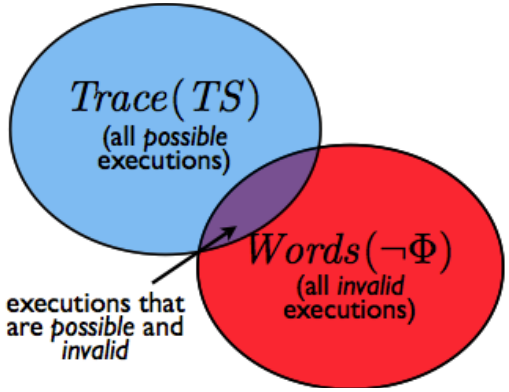
  $\Box$(red $\rightarrow$ ($\Diamond$ green $\wedge$ ($\neg$ green U yellow)))

  $\Box$(red $\rightarrow \bigcirc$ (red U (yellow $\wedge \bigcirc$ (yellow U green))))

Progress property

- Every request will eventually lead to a response

  $\Box$ (request $\rightarrow \Diamond$response)



Temporal Logic of Actions

- Formulas (aka specifications) are built using:
  - values, variables, states, functions, and actions
  - an action is a Boolean-valued expression that relates states
- Temporal logic is a formalism that reasons about behavior in terms of sequences of states (time progresses in a linear order)
- Used to describe the dynamic behavior of concurrent/reactive systems
- Focussed on catching design (rather than implementation) errors
- Used in industry

How Amazon Web Services Uses Formal Methods

Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff

Communications of the ACM

April 2015, Vol. 58, No. 4, pages 66–73

Specifications

- A mathematical description language, compiler, and model-checker for describing and proving properties of concurrent and distributed systems

Example: CHOOSE x \in S : \A y \in S : y <= x.

Built on top of propositional logic

$\wedge$ conjunction (and)	$\Rightarrow$ implication (implies)
$\vee$ disjunction (or)	$\equiv$ equivalence (is equivalent to)
$\neg$ negation (not)	

Sets, and the following two primitive predicates:

$\forall$ universal quantification (for all)
$\exists$ existential quantification (there exists)

Abstraction

- An execution is represented as a sequence of discrete steps
  - simulate a concurrent program as a sequential interleaving of its threads
- A step denotes a change in state
- A sequence of states is a behavior
  - Goal: describe all possible behaviors of a design/system
  - Abstract behaviors as state machines:
    - All possible initial states
    - Actions: next states
    - States:
      - variables
      - initial values
      - relations among them in the current state
      - relations between values in the current state and next state

Example


```
int i ;
void main()
   { i = someNumber() ;
     i = i + 1 ;
   }
```

In English

if current value of pc equals "*start*"
  then next value of i in $\{0, 1, \ldots, 1000\}$
    next value of pc equals "*middle*"
  else if current value of pc equals "*middle*"
    then next value of i equals current value of $i + 1$
      next value of pc equals "*done*"
    else no next values

In TLA+

MODULE *SimpleProgram*

EXTENDS *Integers*

VARIABLES i, pc

$$Init \triangleq (pc = \text{“start”}) \land (i = 0)$$

$$\begin{aligned} Next \triangleq\ & \lor \land pc = \text{“start”} \\ & \quad \land i' \in 0 \,..\, 1000 \\ & \quad \land pc' = \text{“middle”} \\ & \lor \land pc = \text{“middle”} \\ & \quad \land i' = i + 1 \\ & \quad \land pc' = \text{“done”} \end{aligned}$$

Example: Specifying an Hour Clock

$$HCini \triangleq hr \in \{1, \ldots, 12\}$$

$$HCnxt \triangleq hr' = \text{IF } hr \neq 12 \text{ THEN } hr + 1 \text{ ELSE } 1$$

$$HC \triangleq HCini \wedge \Box[HCnxt]_{hr}$$

action

temporal action
why?

stuttering
why?

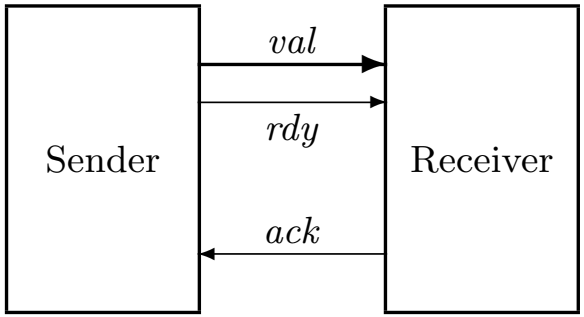
admissible behavior (a possibly infinite collection of states):

$$[hr = 10] \rightarrow [hr = 11] \rightarrow [hr = 11] \rightarrow [hr = 11] \rightarrow \cdots$$

Example


```
---------------------- MODULE HourClock ---------------------
EXTENDS Naturals
VARIABLE hr
HCini  ==  hr \in (1 .. 12)
HCnxt  ==  hr' = IF hr # 12 THEN hr + 1 ELSE 1
HC  ==  HCini /\ [][HCnxt]_hr
-------------------------------------------------------------
THEOREM  HC => []HCini
=============================================================
```

Specifying Asynchrony



$$\begin{bmatrix} val = 26 \\ rdy = 0 \\ ack = 0 \end{bmatrix} \xrightarrow{Send\ 37} \begin{bmatrix} val = 37 \\ rdy = 1 \\ ack = 0 \end{bmatrix} \xrightarrow{Ack} \begin{bmatrix} val = 37 \\ rdy = 1 \\ ack = 1 \end{bmatrix} \xrightarrow{Send\ 4}$$

$$\begin{bmatrix} val = 4 \\ rdy = 0 \\ ack = 1 \end{bmatrix} \xrightarrow{Ack} \begin{bmatrix} val = 4 \\ rdy = 0 \\ ack = 0 \end{bmatrix} \xrightarrow{Send\ 19} \begin{bmatrix} val = 19 \\ rdy = 1 \\ ack = 0 \end{bmatrix} \xrightarrow{Ack} \cdots$$

MODULE *AsynchInterface*

EXTENDS *Naturals*
CONSTANT *Data*
VARIABLES val, rdy, ack

$$\begin{aligned} TypeInvariant \triangleq\ & \land\ val \in Data \\ & \land\ rdy \in \{0,1\} \\ & \land\ ack \in \{0,1\} \end{aligned}$$

$$\begin{aligned} Init \triangleq\ & \land\ val \in Data \\ & \land\ rdy \in \{0,1\} \\ & \land\ ack = rdy \end{aligned}$$

$$\begin{aligned} Send \triangleq\ & \land\ rdy = ack \\ & \land\ val' \in Data \\ & \land\ rdy' = 1 - rdy \\ & \land\ \text{UNCHANGED}\ ack \end{aligned}$$

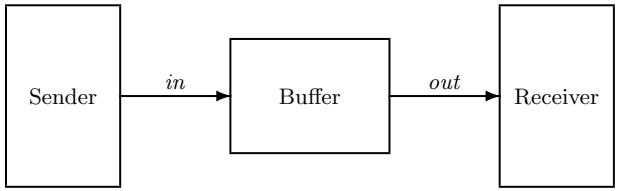
$$\begin{aligned} Rcv \triangleq\ & \land\ rdy \neq ack \\ & \land\ ack' = 1 - ack \\ & \land\ \text{UNCHANGED}\ \langle val, rdy \rangle \end{aligned}$$

$$Next \triangleq Send \lor Rcv$$

$$Spec \triangleq Init \land \Box[Next]_{\langle val, rdy, ack \rangle}$$

THEOREM $Spec \Rightarrow \Box TypeInvariant$

A FIFO Queue



MODULE *Channel*

EXTENDS *Naturals*
CONSTANT *Data*
VARIABLE *chan*

$TypeInvariant \triangleq chan \in [val : Data,\ rdy : \{0,1\},\ ack : \{0,1\}]$

$Init \triangleq \land TypeInvariant$
$\land chan.ack = chan.rdy$

$Send(d) \triangleq \land chan.rdy = chan.ack$
$\land chan' = [chan \text{ EXCEPT } !.val = d,\ !.rdy = 1 - @]$

$Rcv \triangleq \land chan.rdy \neq chan.ack$
$\land chan' = [chan \text{ EXCEPT } !.ack = 1 - @]$

$Next \triangleq (\exists\, d \in Data : Send(d)) \lor Rcv$

$Spec \triangleq Init \land \Box[Next]_{chan}$

THEOREM $Spec \Rightarrow \Box TypeInvariant$

MODULE *FIFO*

CONSTANT *Message*
VARIABLES *in*, *out*

$Inner(q) \triangleq$ INSTANCE *InnerFIFO*

$Spec \triangleq \exists\, q : Inner(q)!Spec$

MODULE *InnerFIFO*

EXTENDS *Naturals*, *Sequences*
CONSTANT *Message*
VARIABLES *in*, *out*, *q*

$InChan \triangleq$ INSTANCE *Channel* WITH *Data* $\leftarrow$ *Message*, *chan* $\leftarrow$ *in*
$OutChan \triangleq$ INSTANCE *Channel* WITH *Data* $\leftarrow$ *Message*, *chan* $\leftarrow$ *out*

$Init \triangleq \land InChan!Init$
$\land OutChan!Init$
$\land q = \langle\,\rangle$

$TypeInvariant \triangleq \land InChan!TypeInvariant$
$\land OutChan!TypeInvariant$
$\land q \in Seq(Message)$

$SSend(msg) \triangleq \land InChan!Send(msg)$
$\land$ UNCHANGED $\langle out, q \rangle$

Send *msg* on channel *in*.

$BufRcv \triangleq \land InChan!Rcv$
$\land q' = Append(q, in.val)$
$\land$ UNCHANGED *out*

Receive message from channel *in* and append it to tail of *q*.

$BufSend \triangleq \land q \neq \langle\,\rangle$
$\land OutChan!Send(Head(q))$
$\land q' = Tail(q)$
$\land$ UNCHANGED *in*

Enabled only if *q* is nonempty. Send $Head(q)$ on channel *out* and remove it from *q*.

$RRcv \triangleq \land OutChan!Rcv$
$\land$ UNCHANGED $\langle in, q \rangle$

Receive message from channel *out*.

$Next \triangleq \lor \exists\, msg \in Message : SSend(msg)$
$\lor BufRcv$
$\lor BufSend$
$\lor RRcv$

$Spec \triangleq Init \land \Box[Next]_{\langle in, out, q \rangle}$

THEOREM $Spec \Rightarrow \Box TypeInvariant$

TLC: A Model-Checker for TLA+

- Handles specifications of the form:

$$Init \wedge \Box[Next]_{vars} \wedge Temporal$$

- Without specifications, TLC checks for:
  - type errors
  - deadlock, i.e., violation of:

$$\Box(\text{ENABLED } Next).$$

- TLC generates behaviors that satisfy a specification based on a model (i.e, an assignment of values to variables in the specification)
  - check all reachable states i.e., find all states that can occur in behaviors satisfying:

$$Init \wedge \Box[Next]_{vars}.$$

- Checking all states is usually impossible (e.g., arbitrarily many messages) - infinitely many states/paths.
  - In practic, "finitize" the model