A tutorial on SPIN

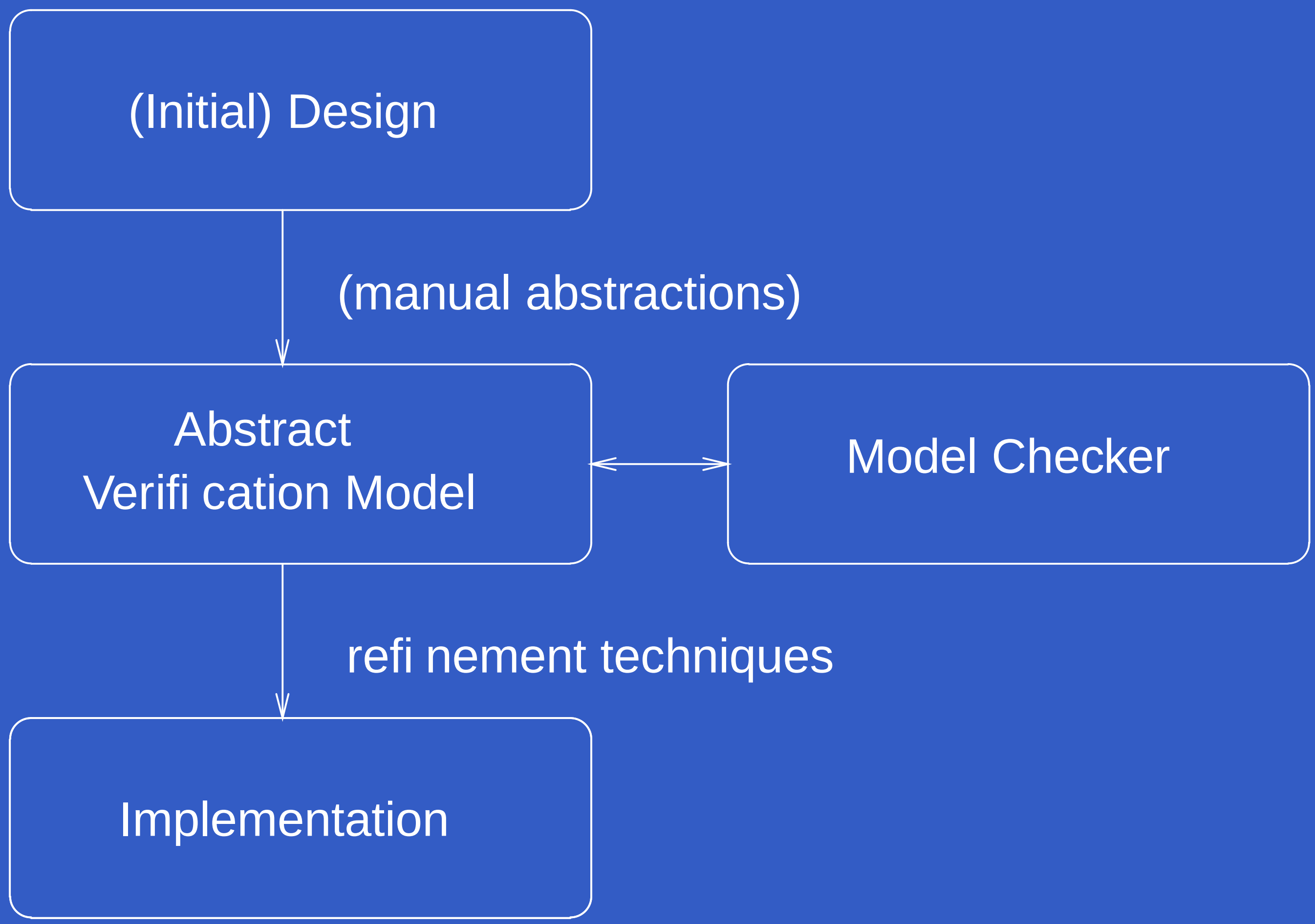
Meenakshi. B.

Honeywell Technology Solutions Lab
Bangalore.

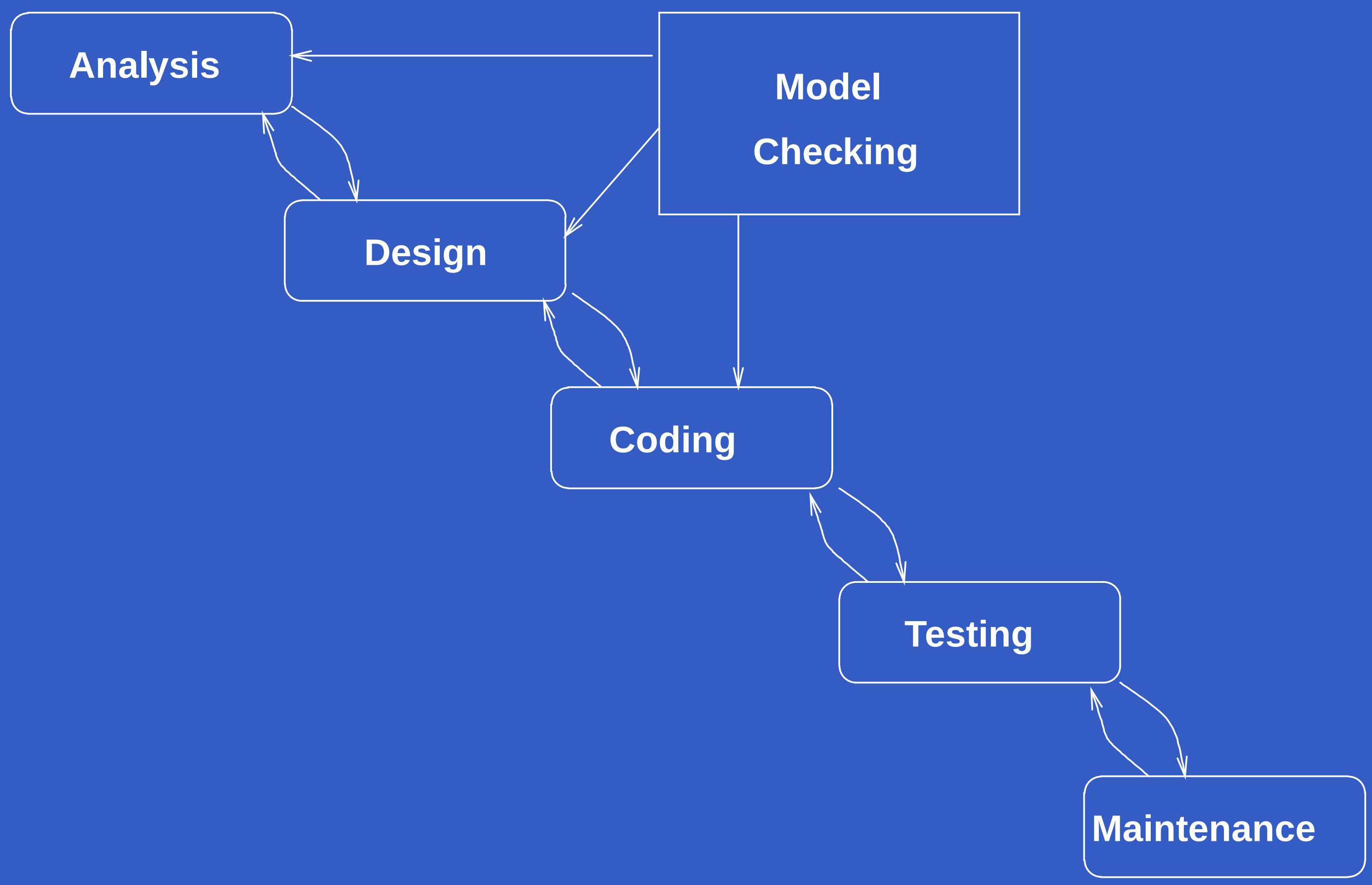
What is Model Checking?

- **Clarke & Emerson 1981**: 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.
- Model checkers are tools that perform model checking.
- Inputs: M, a finite state model of the system and ϕ, a requirement.
  Output: Yes or No + a system run violating the requirement (*Counter example*).

Model Checking



Model of System Development



Some popular model checkers

- SPIN: Verification of distributed software systems
  `http://www.spinroot.com`
- SMV: Verification of hardware circuits
  `http://www-cad.eecs.berkeley.edu/~kenmcmil/smv/`
- UPPAAL: Verification of real-time systems.
  `http://www.docs.uu.se/docs/rtmv/uppaal/index.shtml`

Distributed systems

- *Distributed systems*: Systems with many components (processes) that communicate by exchanging messages, synchronously or by using shared variables.
- Examples include network applications, data communication protocols, multi-threaded code, client-server applications.

Design flaws in distributed systems

Common design flaws that occur in design of distributed systems are

- Deadlock — all the processes/components are blocked.
- Livelock, starvation — all the processes are doing "useless" computation.
- Underspecification — unexpected reception of messages.
- Overspecification — Dead code

The model checker SPIN

- SPIN (Simple ProMeLa INterpreter) is a verification tool for models of distributed software systems.
- SPIN takes a model of the system design and a requirement as input and the model checking algorithm specifies whether the system design meets the requirement or not. If the requirement is not met, SPIN pulls out a system run which violates the requirement (*counter example*).

Focus of SPIN

- SPIN verification is focussed on proving the correctness of *process interactions*; not much importance is given to internal computations of the processes.
- *Processes* refer to system components that communicate with each other.
- Communication is through rendezvous primitives (synchronous), with asynchronous message passing through buffered channels, through access to shared variables or with any combination of these.

What does SPIN provide?

As a formal verification tool, SPIN provides

1. An intuitive, C-like notation for specifying system design or its finite-state abstraction unambiguously (*ProMeLa — Process Meta Language*).
2. A notation for expressing general correctness requirements as *LTL formulae*.
3. A methodology for establishing the logical consistency of system design specified in ProMeLa and the matching correctness requirements written as LTL formulae.

SPIN ad!

- SPIN won the ACM software system award for 2001 (Other winners include UNIX (1983), TeX (1986), TCP/IP (1991), WWW (1995) and Java (2002)).
- Holzmann (author of SPIN) won the Thomas Alva Edison patent award in the Information Technology Category, for the patent on software verification with SPIN in 2003.
- SPIN is an open source tool.

ProMeLa model

- ProMeLa is a C-like language to describe models of distributed systems.
- ProMeLa also borrows notation from Dijkstra’s guarded command language and Hoare’s CSP language to talk about process interactions.
- A model specified in ProMeLa is non-deterministic and *finite state*.

ProMeLa model

ProMeLa model consists of

- *variable* declarations with their types
- *channel* declarations
- *type* declarations
- *process* declarations
- *init* process (optional)

ProMeLa model — example

```
bool flag;
chan PtoQ;
mtype = \{msg, ack\};

proctype P() \{   proctype Q() \{
             ...                ...
            \}                 \}

init \{
     ...
    \}
```

Processes in ProMeLa

- A process is defined by a `proctype` definition.
- A `proctype` definition consists of
  - name of the process
  - list of formal parameters
  - declaration of local variables
  - sequence of statements local to the process

Process definition—Example

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

Processes in ProMeLa

- There can be more than one process inside a ProMeLa model.
- A process executes *concurrently* with other processes.
- A process also *communicates* with other processes by sending/receiving messages across channels by using shared (global) variables with other processes.
- *Local state* of a process is defi ned by *process counter* (defi nes the location of the process) and the values of the local variables of the process.

Invoking a process

- Processes can be created at any point inside the model (even within another process).
- Creation of a process is done by using a `run` statement inside the `init` process.
- Processes can also be created by adding the keyword `active` in front of the `proctype` declaration.

Invoking a process—Example

```
 proctype P(byte x) {
    ...
}
init {
    run P(19);
   ...
}
...
active Q(int y) {
    ...
}
```

Variables in ProMeLa

- Variables should be declared. A declaration consists of the *type* of the variable followed by its name.
- There are five different types— `bit` ([0..1]), `bool` ([0..1]), `byte` ([0..255]), `short` ($[-2^{1}6 - 1..2^{1}6 - 1]$), `int` ($[-2^{3}2 - 1..2^{3}2 - 1]$).

Variables in ProMeLa

- ProMeLa models can also have *arrays* and *records*.
- Arrays are declared with their name followed by their range (array indexing starts from 0) and records are declared by a `typedef` declaration folllowed by the record name.

Variables in ProMeLa

- Variables can be *local* or *global*.
- Default initial value of both local and global variables is 0.
- Variables can be assigned a value by an assignment, argument passing or message passing.
- Type conflicts are found at run-time.
- Variables can be used in *expressions* which includes most arithmetic, relational and logical operators of C.

Statements in ProMeLa

- Statements are separated by a semi-colon.
- *Assignments* and *expressions* are statements.
- `skip` statement: does nothing, only changes the process counter.
- `printf` statement: not evaluated during verifi cation.
- `assert(expr)`: Assert statement is used to check if the property specifi ed by the expression `expr` is valid within a *state*. If `expr` evaluates to 0, it implies that it is not valid and SPIN will exit with an error.

`if` statement

- ```
   if
  ::  choice1 -> stat1.1; stat1.2; ...
  ::  choice2 -> stat2.1; stat2.2; ...
  ::  ...
  ::  choicen -> statn.1; statn.2; ...
  fi;
  ```
- `if` statement is *executable* if there is at least one choice which is executable and is *blocked* if none of the choices are executable.
- If more than one choice is executable, SPIN *non-deterministically* chooses one of the executable choices.
  ```

`if` statement—Example

```
if
::   (n >= 0) -> n = n - 2
::   (n%3 == 0) -> n = 3
::   else -> skip
fi;
```

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

Smart use of `if` statement

Give the variable n a random value between 1 and 3.

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

do statement

- ```
  do
  ::  choice1 -> stat1.1; stat1.2; ...
  ::  choice2 -> stat2.1; stat2.2; ...
  ::  ...
  ::  choicen -> statn.1; statn.2; ...
  od;
  ```
- do statement behaves in the same way as `if` statement in terms of choice selection but, executes the choice selection repeatedly.
- `break` statement can be used to come out of a do loop. It transfers control to the statement just outside the loop.
  ```

Modelling communications with chann

- Communication between processes is through *channels*.
- There can be two types of communications:
  - Message-passing or asynchronous
  - Rendezvous or synchronous

Channels in ProMeLa

Channels are FIFO in nature and are declared as arrays:

```
chan <name> = [<dim>] of
<type1>,<type2>,  <typen>;
```

`name` is the name of the channel, `dim` is the number of elements that can occupy the channel (synchronous communication is through a channel of dimension 0) and `type1` etc. are the type of elements that can be passed in the channel.

Example: `chan ptoq = [2] of {mtype, bit}`

Sending and receiving messages in Pro

- The notation for sending a message in a channel is !.
```
chan-name !  <expr1>, <expr2>, ...,
<exprn>;
```
- The notation for receiving a message from a channel is ?.
```
chan-name ?  <expr1>, <expr2>, ...,
<exprn>;
```
- In both the cases, the type of the expression should match the channel declaration.

Modelling rendezvous communication

- Rendezvous communication is modelled using a channel of dimension zero.
- If sending through a channel is enabled and if there is a *corresponding* receive that can be executed simulteneously, then both the statements are enabled. Both the statements will *handshake* together and it will be a *common transition* between the sending and the receiving process.

Example

Example:
chan ch = [0] of bit, byte;

- P wants to do ch ! 1, 3+7
- Q wants to do ch ? 1, x
- After the communication, x will have the value 10.

Interleaving Semantics

- Statements belonging to different processes are interleaved.
- Interleaving: If two statements of two different processes can be executed independent of each other, then the order of their execution is arbitrary.
- Example: Statements changing values of two local variables by two different processes.

Statements—executable or blocked

ProMeLa statements are either executable or blocked.

- Assignment statements, `skip`, `break`, `printf` statements are always executable.
- An expression is executable if it does not evaluate to zero.
- `if` and `do` statements are executable if at least one guard evaluates to true.
- Send is executable if the channel is not full (by default) and receive is executable if the channel is not empty.

Atomic statement

```
atomic { statement1; ...; statementn }
```

- Can be used to group statements of a particular process into one *atomic sequence*. That is, the statements are executed in a single step and are not interleaved with statements of other processes.
- The statement is *executable* of the fi rst statement `statement1` is executable.
- The atomicity is broken if any of the statements is blocking. That is, statements of other processes can be interleaved in between.

Atomic statement: Example

```
proctype P { byte x, y;
atomic {
x++;
y--;
}
}
```

d-step statement

```
 d-step { statement1; ...;
statementn }
```

- Again executed in one step.
- No intermediate states are generated or stored.
- If one of the statements `statementi` blocks, it is a run-time error.

`atomic` and `d-step` can be used to reduce the *number of states* in the ProMeLa model.

Timeout statement

```
timeout
```

- `timeout` statement becomes executable if no other statement in any process is executable.
- It is like a system timeout that SPIN uses to excape from hanging or deadlock and is global.
- It is not a real-time feature and is cannot be used to model time-outs involved in the system design.

SPIN references

- SPIN page: http://spinroot.com
- G. Holzmann, *The Model Checker Spin* ,IEEE Trans. on Software Engineering, Vol. 23, No. 5, May 1997, pp. 279-295.
- G. Holzmann, *The Spin Model Checker: Primer and Reference Manual*, Addison-Wesley, ISBN 0-321-22862-6, 608 pgs, cloth-bound.