# Overview of SRI's Symbolic Analysis Laboratory (SAL)

Lee Pike

June 3, 2005

lee.s.pike@nasa.gov

Introduction to Automated Verification

SAL: A Verification Framework

The Language

Examples

The Future...

# Model-Checking 101

Model-checking is a way *automatically* to verify hardware or software. For a property $P$,

- A *Model-checking program* checks to ensure that every state on each execution path satisfies $P$.
- Returns a counter-example otherwise.

# No Free Lunch

- Model-checking is expensive (both in space and time).
- Most model-checkers can handle only finite models.
- The specification must be encoded as a state machine, and properties must be stated in a restricted language (temporal logic).

# Benefits of Model-Checking

- Dramatic improvements over the years (in theory and practice) have scaled-up automated verification of real-world systems.
- Relatively less user expertise & user interaction required than for theorem-proving.
- Many industrial problems fit the "model-checking paradigm."

# Some Well-Known Model-Checkers

- Action Language Verifier (discrete-time specification)
  <http://www.cs.ucsb.edu/~bultan/composite/>
- MOCHA (symbolic)
  <http://www-cad.eecs.berkeley.edu/~mocha/>
- NuSMV (symbolic, bounded)
  <http://nusmv.irst.itc.it/>
- SMART (symbolic—MDD's)
  <http://www.cs.ucr.edu/~ciardo/SMART/index.html>
- SPIN (explicit-state)
  <http://spinroot.com/spin/whatispin.html>
- Uppaal (timed automata)
  http://www.uppaal.com/

N.B. This list is not exhaustive (nor representative)!

# About SAL

The Symbolic Analysis Laboratory (SAL) is an integrated formal verification environment.

- Developed by SRI, International (the makers of PVS).
- Publicly available at <http://sal.csl.sri.com/> (for noncommercial use).
- Available for:
  - Linux
  - Solaris
  - MacOS X
  - Cygwin (for Windows)

# The SAL Philosophy



- One language, many tools.
- Designed for extension: model-checkers are Scheme scripts.
- Plug 'n play:
  - Can be used with multiple decision procedures (e.g., CVC Lite, CVC, SVC, UCLID, etc.).
  - Can be used with multiple SAT solvers (e.g., ICS, Siege, zChaff, Berkmin, etc.).

# (Finite-State) Model-checkers

- Symbolic model-checker (BDDs) (MDDs in the future)
- Witness symbolic model-checker
- Bounded model-checker
- (Explicit-state model-checker in the future)

All of which are "state-of-the-art"

# Other Tools

- Simulator
- Parser
- Infinite-state bounded model-checker!

# Overview

- Building block: the module
- Typed
- Synchronous and asynchronous composition of modules
- XML abstract syntax exists for the language

# Types



The language is typed, following PVS typing conventions

- Finite Types (e.g., booleans, finite arrays, records, finite ranges of $\mathbb{Z}$, tuples)
- Infinite types (e.g., $\mathbb{R}$, $\mathbb{N}$)
- Subtyping possible

# Variables

- With respect to a module, variables can be
  - Local
  - Global
  - Input
  - Output
- Modules can update global, local, and output variables
- Communication between modules via shared variables

# Other Considerations

- Uninterpreted constants & functions
- Interpreted constants & functions
- Quantification over finite types
- Synchronous and asynchronous composition operators

# A Module (Bakery Example)

```
PC: TYPE = {sleeping, trying, critical};

job: MODULE =
BEGIN
  INPUT  y2 : NATURAL
  OUTPUT y1 : NATURAL
  LOCAL  pc : PC
  INITIALIZATION
    pc = sleeping;
    y1 = 0
  TRANSITION
  [
      pc = sleeping --> y1' = y2 + 1;
                        pc' = trying
   []
      pc = trying AND (y2 = 0 OR y1 < y2) --> pc' = critical
   []
      pc = critical --> y1' = 0;
                        pc' = sleeping
  ]
END;
```

# Module Composition

- Asynchronous composition:

```
system: MODULE = reader [] writer;
```

- Synchronous composition:

```
system: MODULE = reader || writer;
```

- Parameterized composition with renaming:

```
IDENTITY: TYPE = [1 .. 5];
                      ⋮
system: MODULE =
  WITH OUTPUT time_out: TIMEOUT_ARRAY
  ([] (i: IDENTITY): (RENAME timeout TO time_out[i]
                      IN process[i]));
```

# Property Specification Language

- CTL or LTL, depending on the model checker
- Examples:
  - ```
    reachable: THEOREM
      system |- (FORALL (i : Process_Id): EF(pc[i] = cs));
    ```
  - ```
    mutex: THEOREM
      system |- G(NOT(pc.1 = critical AND pc.2 = critical));
    ```

# Invariants

- Finding inductive invariants that hold in every state for transition systems is hard (especially in infinite-state systems).
- Sometimes finding an invariant that holds after $k$ steps is easier.
- Intuition:
  - A subroutine is guaranteed to complete in $k$ steps and guarantees some invariant property.
  - Reduces the number of unreachable states considered in the inductive hypothesis.

# $k$-Induction



- $k$-Induction is a generalization of induction (for transition systems):
- *$k$-Induction Principle*: to show that $I(s)$ holds for all reachable states $s$, show

  **Base Case** For all trajectories of length $k$ that begin with an initial state, show each state of the trajectory satisfies $I$.

  **Induction Step** For all trajectories of length $k$ such that $I(s_i)$ for $0 \leq i \leq k-1$, show that for each state $s_k$, $I(s_k)$.
- Induction is the special case where $k = 1$

# Recent Successes

- The verification of a real-time model of the TTP/C startup protocol using `sal-inf-bmc`

  Bruno Dutertre & Maria Sorea (SRI)
- The efficient generation of test-cases to meet a coverage criterion

  Grgoire Hamon (Chalmers),
  Leonardo de Moura & John Rushby (SRI)
- The verification of a real-time model of a reintegration protocol using `sal-inf-bmc`

  Lee Pike (NASA)
- Many other nontrivial examples
  <http://sal.csl.sri.com/examples.shtml>

# PVS & SAL: When to Use What

- PVS may be preferable if . . .
  - You are doing "real math" (calculus, number theory, algebra, etc.).
  - You want to write abstract specifications & requirements.
  - You want to reason at the "requirements level."
- SAL may be preferable if . . .
  - Your specification is a state machine.
  - you want to prove invariants over infinite-state systems, relative to a decidable theory (`sal-inf-bmc`).
  - You can write specifications in a temporal logic.
  - You want to reason at the "implementation level."

In practice, these tools will cohabit a formal verification endeavor. . .

# Future Work

- Tighter integration with PVS
- Type-checking
- Additional optimizations & improvements

SAL 2.4 to be released soon!