SPIN 99: Trento, Italy, 9:10 to 10:30 on 5 July 1999

Integrated Formal Verification:

Using Model Checking With Automated Abstraction, Invariant Generation, and Theorem Proving

John Rushby

Computer Science Laboratory
SRI International
Menlo Park, California, USA

Model Checking Arrives on the Scene

- Model checking rescued the reputation of formal methods—Daniel Jackson
- Model checking not only introduced useful tools, such as SPIN
- It brought a different focus to formal methods
  - Finding bugs, rather than proving correctness
  - Tools and pragmatics, rather than mathematical exercises
- This was greatly liberating for those who had been working on traditional tools, based on theorem proving
- And stimulated them to provide greater automation

Aside: Mechanized Formal Methods Using Theorem Proving

- Specify
  - The system (at some suitable level of abstraction)
  - A required property
  - The assumptions
  - Necessary background theories

  as formulas in a single logic
- And prove that

$$background + assumptions + system \vdash requirement$$

- Variation:
  - Prove that implementation is refinement of specification (i.e., commutative diagram, or theory interpretation)

Cf. Mechanized Formal Methods Using Model Checking

- Specify
  - The system as a finite transition system
  - A required property as a formula in a temporal logic
- Establish that the system is a model of the formula

$$system \models requirement$$

  Accomplished by a search over the state space
- Usually no background theories
  (sometimes limited arithmetic)
- Usually, the only assumptions concern interpretation of liveness properties (branching vs. linear time, fairness)
  - Built in to choice of logic and selection of operators

Model Checking as a Back-End to Theorem Proving

- One response was greater use of decision procedures
  - Often resisted before because of fundamentalist approaches to soundness
- Another was to use model checking as a back-end decision procedure
  - E.g., in PVS, a symbolic model checker (due to Geert Janssen) is used as a decision procedure for Park's μ-calculus over heriditarily finite types
  - And CTL model checking is supported on top of that

Theorem proving as a Front-End to Model Checking

- Conversely, as users pushed against the limits of model checking
    - State explosion
    - Non-finite data
- They started using lightweight theorem proving as a front-end to model checking

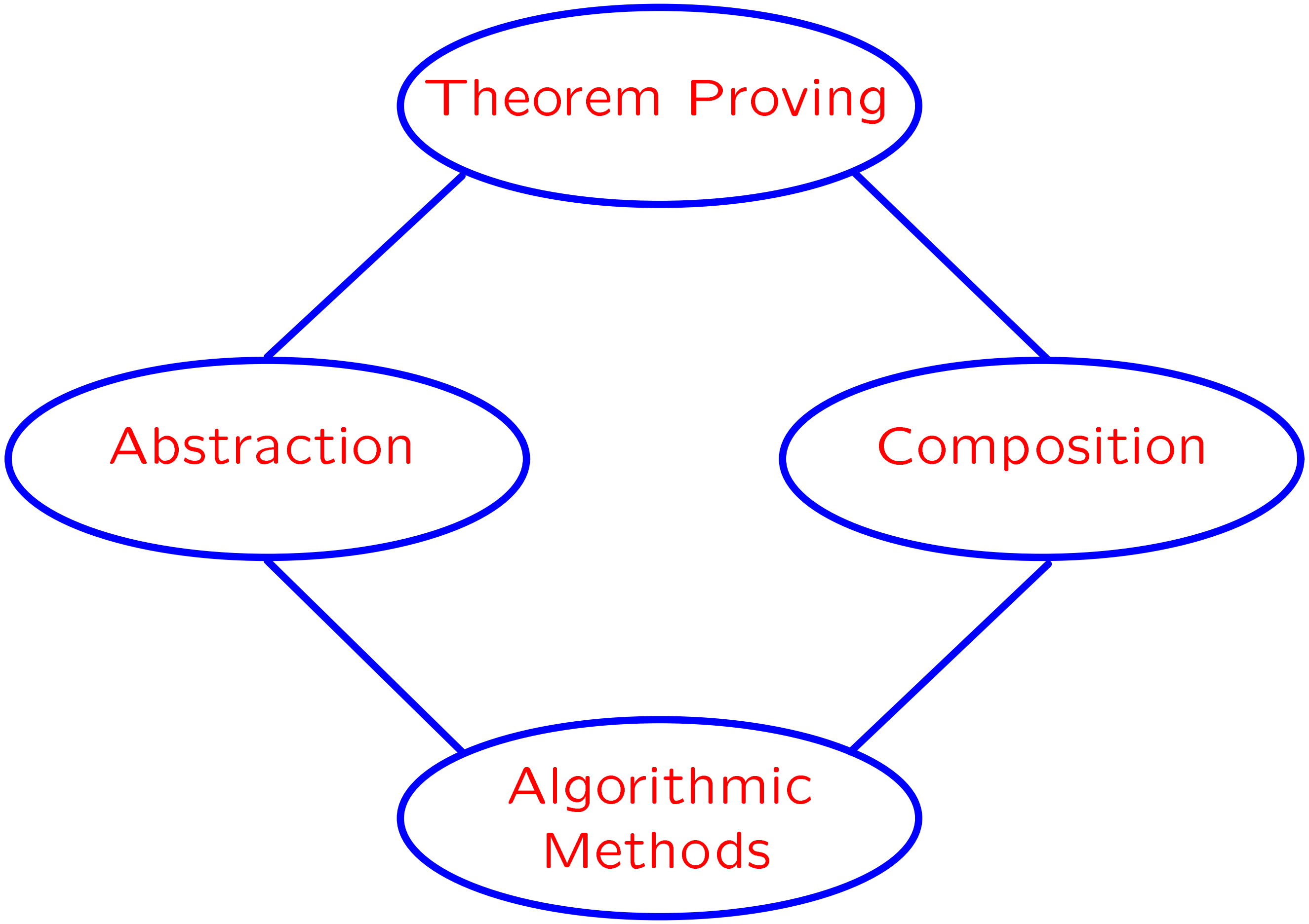
Ken McMillan's Slide

- Goes here

However...

- These are asymmetric approaches
  - Each sees the “other” technology as just a supporting player
- I believe that significant further advances require more symmetric approaches
  - Theorem proving and model checking cooperate as partners
  - Supported by additional new capabilities
    - ⋆ Such as abstraction and invariant generators
  - In a compositional framework

The Symmetric View



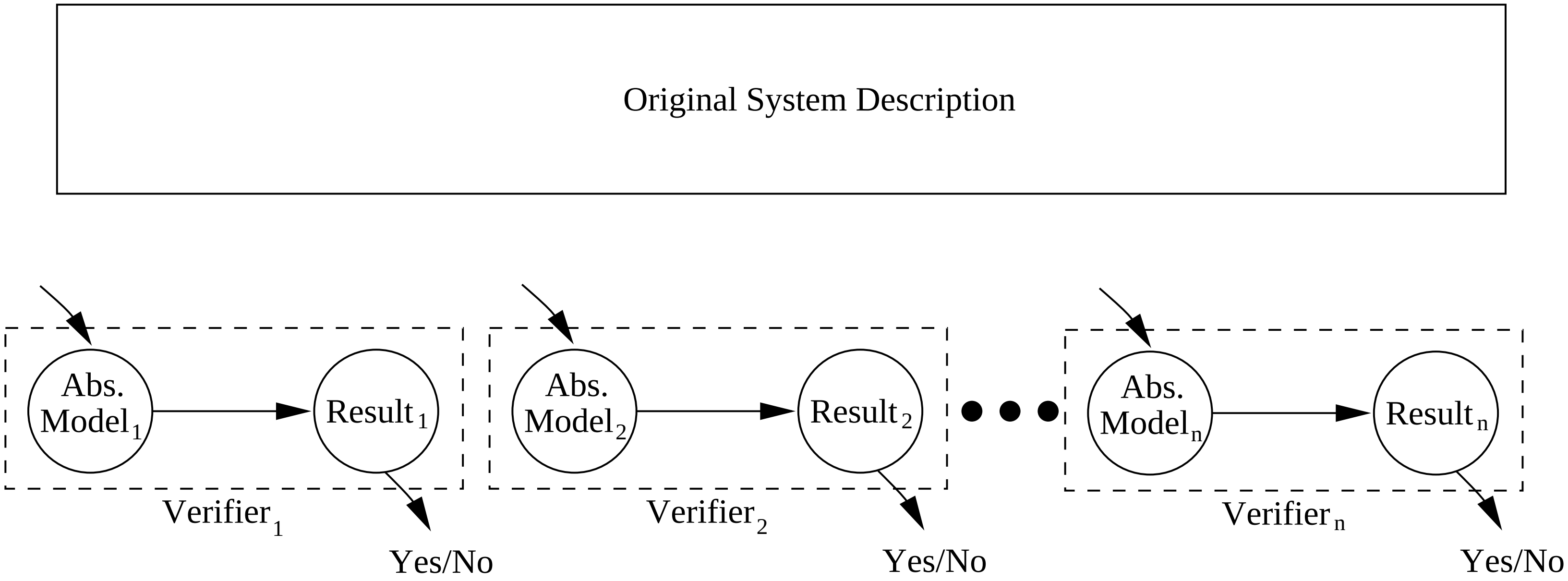
Abstraction and composition are the bridges between deductive and algorithmic methods of verification

Current Formal Analysis Lifecycle

- Start with a specification or design in some generally-used language or notation
  - UML, Java, C, Verilog, VHDL
- Extract the part relevant to the property of interest
- Then reduce it to a form and notation where algorithmic techniques can be applied
  - The reduction is often Draconian (e.g., downscaling)
  - May produce false positives and negatives
- Perform the analysis
- Iterate until satisfied

Formal Analysis Today

- The reduced models are built by hand
- Only results are yes/no verifications (and, possibly, counterexamples at the level of the abstracted description)
- No connection between different tools, nor between different methods of analysis

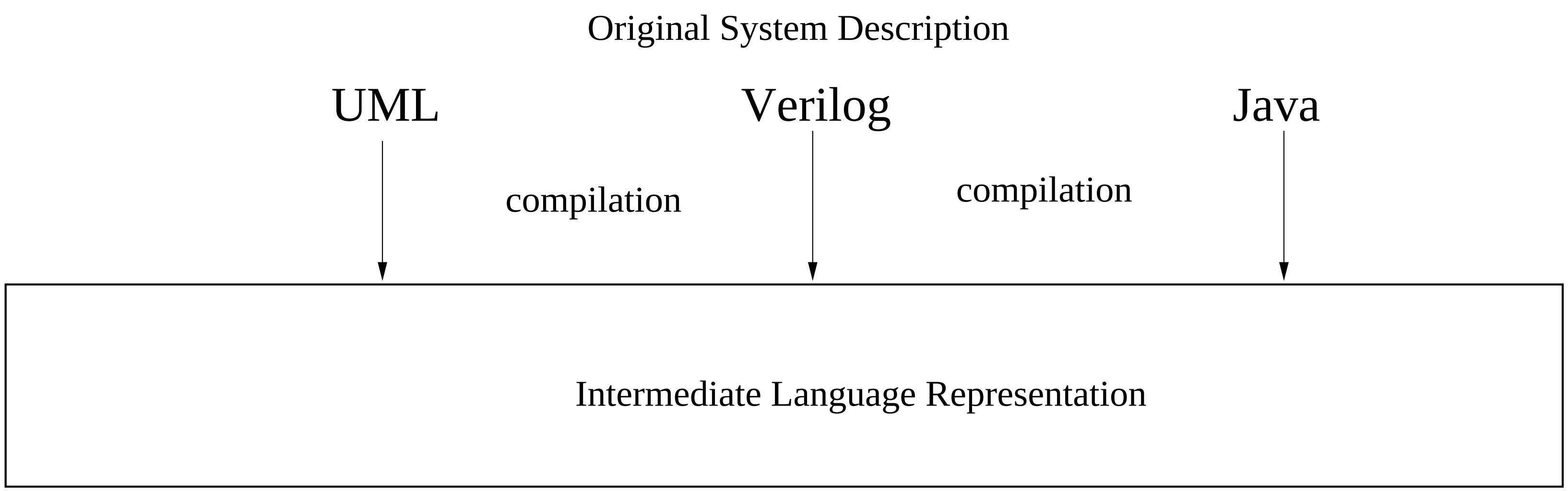


Integrated Approaches—I

- Currently, only the analysis is automated
- So automate the extraction
  - Slicing, compilation
- And the reduction
  - Abstraction, invariant generation

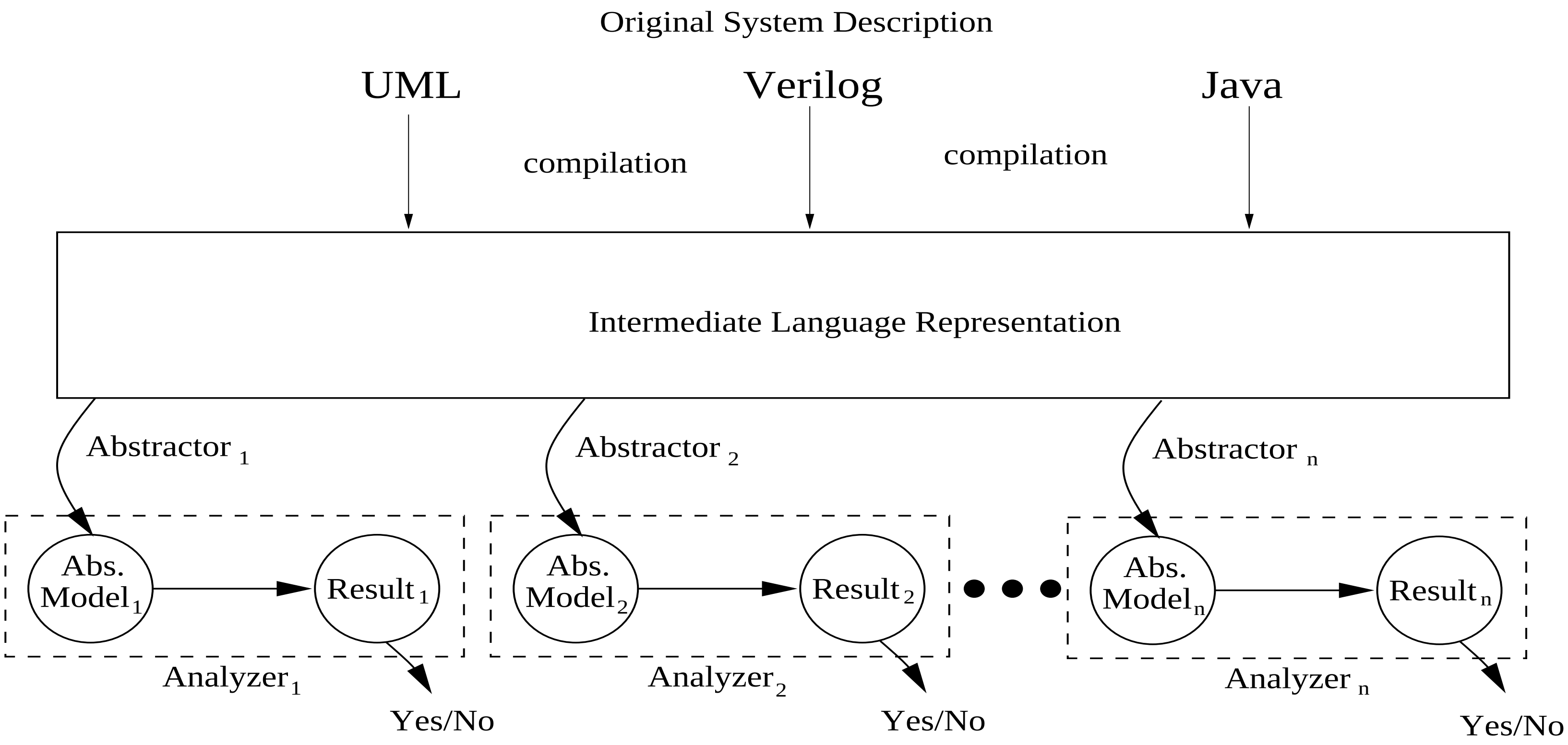
The Vision of Integration—Part 1a

System descriptions are compiled into a common intermediate language representation



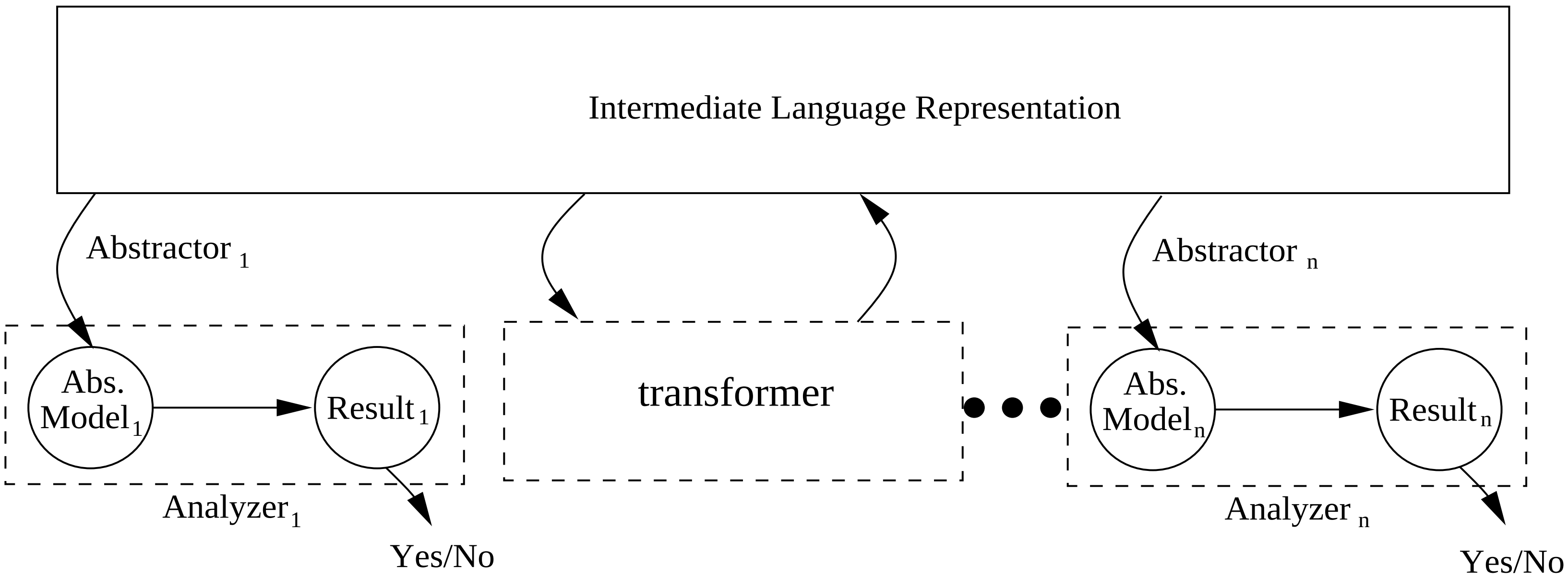
The Vision of Integration—Part 1b

Models for analysis are constructed by automated abstractors from the common intermediate representation



The Vision of Integration—Part 1c

- Can also imagine transformations on the intermediate representation
- For example, partial order reduction
  - Why should this be limited to SPIN?

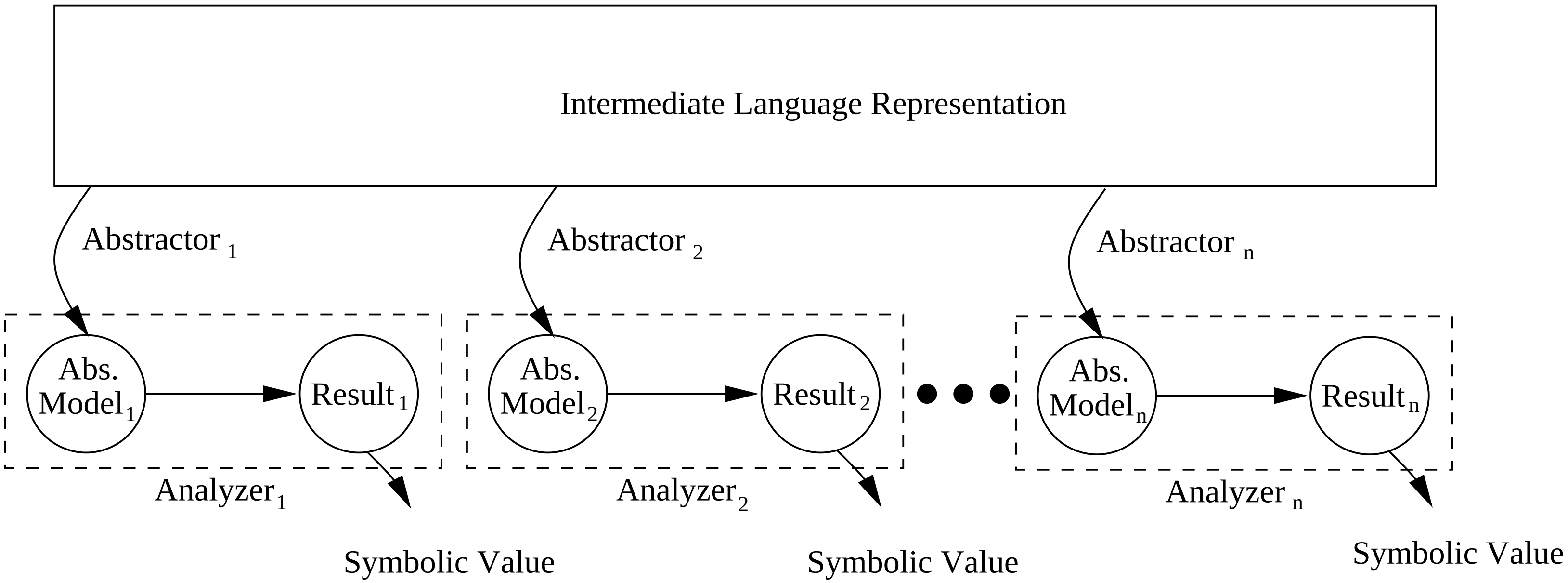


Integrated Approaches—II

- Currently, only one analysis is performed at a time
  - Information is not fed back that could help further analyses, or reductions
- The big idea
  - Think of calculating and accumulating properties rather than just verifying a specific property
  - Properties (e.g., invariants) can help produce more refined abstractions
  - Requires that tools return symbolic values and properties rather than just yes/no results of verifications

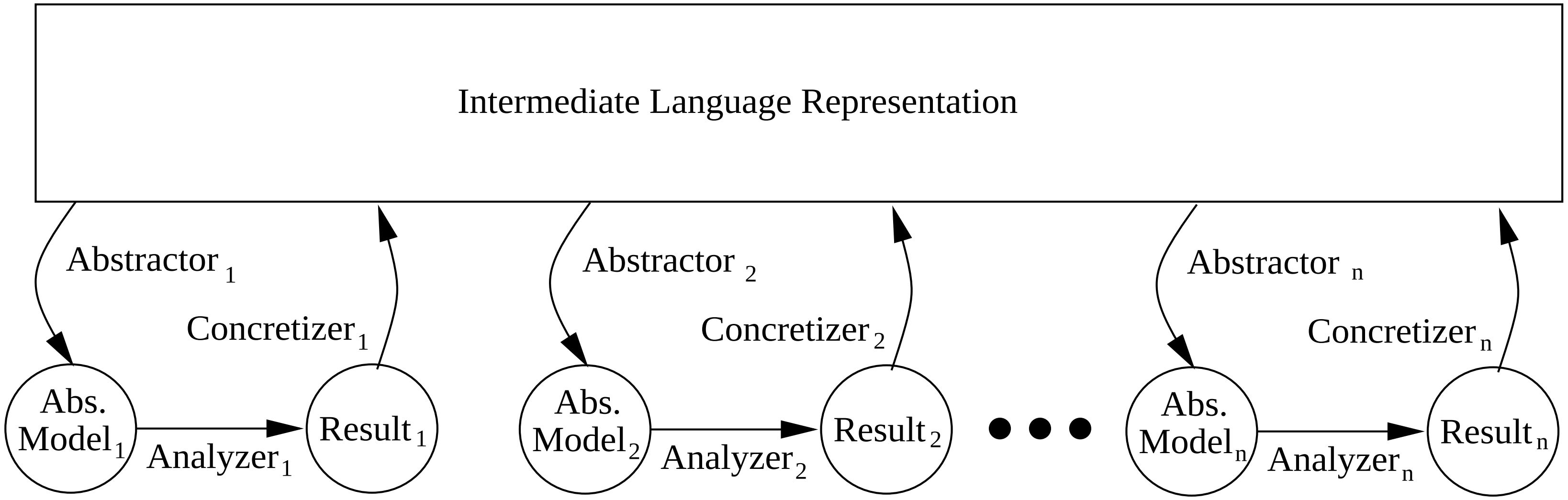
The Vision of Integration—Part IIa

The tools produce symbolic values to assist further analyses



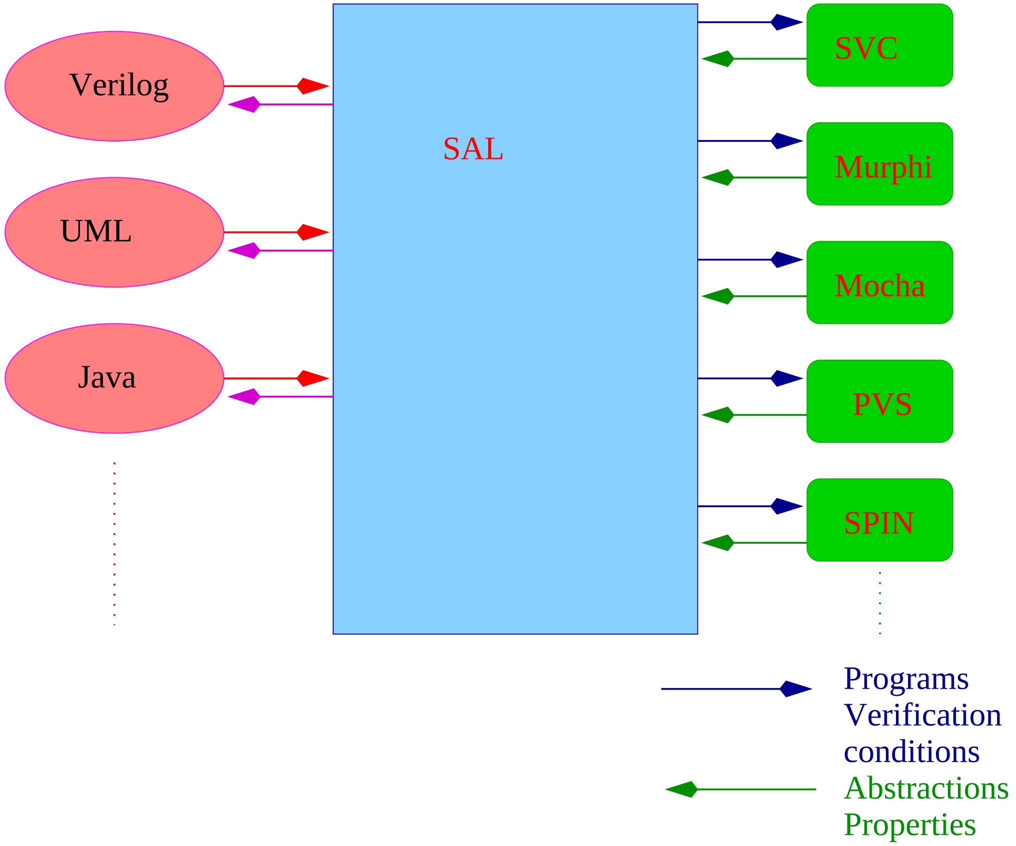
The Vision of Integration—Part IIb

- Symbolic values are converted back into the terms of the original description by automated concretizers
- And the whole process can iterate through several tools, each using results computed by others



Architectures Proposed

SAL (Symbolic Analysis Laboratory) at SRI and Stanford, and VeriTech at The Technion both use (different) intermediate language representations in blackboard architectures



SAL Language (Just for Illustration)

- Intended as an intermediate representation
  - Not as something users write
- Based on transition relations
- Must be able to represent special cases effectively

  Hardware: Deterministic, defined by assignments to variables, synchronous composition

  Protocols: Nondeterministic, defined by guarded commands that may update several variable simultaneously, asynchronous composition

  As well as general and mixed systems
- To allow optimized/dedicated analyses

SAL: L0 Language

- Assignments to variables

```
onebitadder: MODULE =
 INPUT cin, a, b: boolean
 OUTPUT cout, sum
 DEFINE sum = (a XOR b) XOR cin
        cout = (a AND b) OR (a AND cin) OR (b AND cin)
 END
```

- Causal loops in synchronous systems must be eliminated without undue restrictions

```
X = IF A THEN NOT Y ELSE C ENDIF
Y = IF A THEN B ELSE X ENDIF
```

- SAL generates proof obligations to do this
  - Opportunity created by having a theorem prover

SAL: Synchronous Composition

```
Nbitadder [N : natural] : MODULE =
  INPUT A, B : ARRAY [0..N] OF boolean, carryin: boolean
  OUTPUT S, C : ARRAY [0..N] OF boolean
   (RENAME a TO A[0], b TO B[0], cin TO carryin,
            sum TO S[0], cout TO C[0]
      onebitadder)
    ||
   (|| (i : [1..N]):
      (RENAME a TO A[i], b TO B[i], cin TO C[i-1],
               sum TO S[i], cout TO C[i]
          onebitadder))
```

SAL: L1 Language

```
PC : TYPE = {sleeping, trying, critical}
 mutex[tval: boolean]  : MODULE =
  INPUT pc2 : PC, x2 : bool
  OUTPUT pc1 : PC, x1 : bool
  INIT
    TRUE --> pc1 = sleeping
  NEXT
    pc1 = sleeping -->
        {pc1' = trying;  x1' = (x2 = tval)}
    pc1=trying AND (pc2=sleeping OR x1 = (x2/=tval)) -->
        {pc1' = critical}
    pc1 = critical -->
        {pc1' = sleeping;  x1' = (x2 = tval)}
 END
```

SAL: Asynchronous Composition

Two-process mutual exclusion

```
system : MODULE  =
 LOCAL x1, x2
  (mutex[FALSE]
   []
  (RENAME pc2 TO pc1, x2 TO x1,
          pc1 TO pc2, x1 TO x2
   mutex[TRUE]))
```

Abstractors

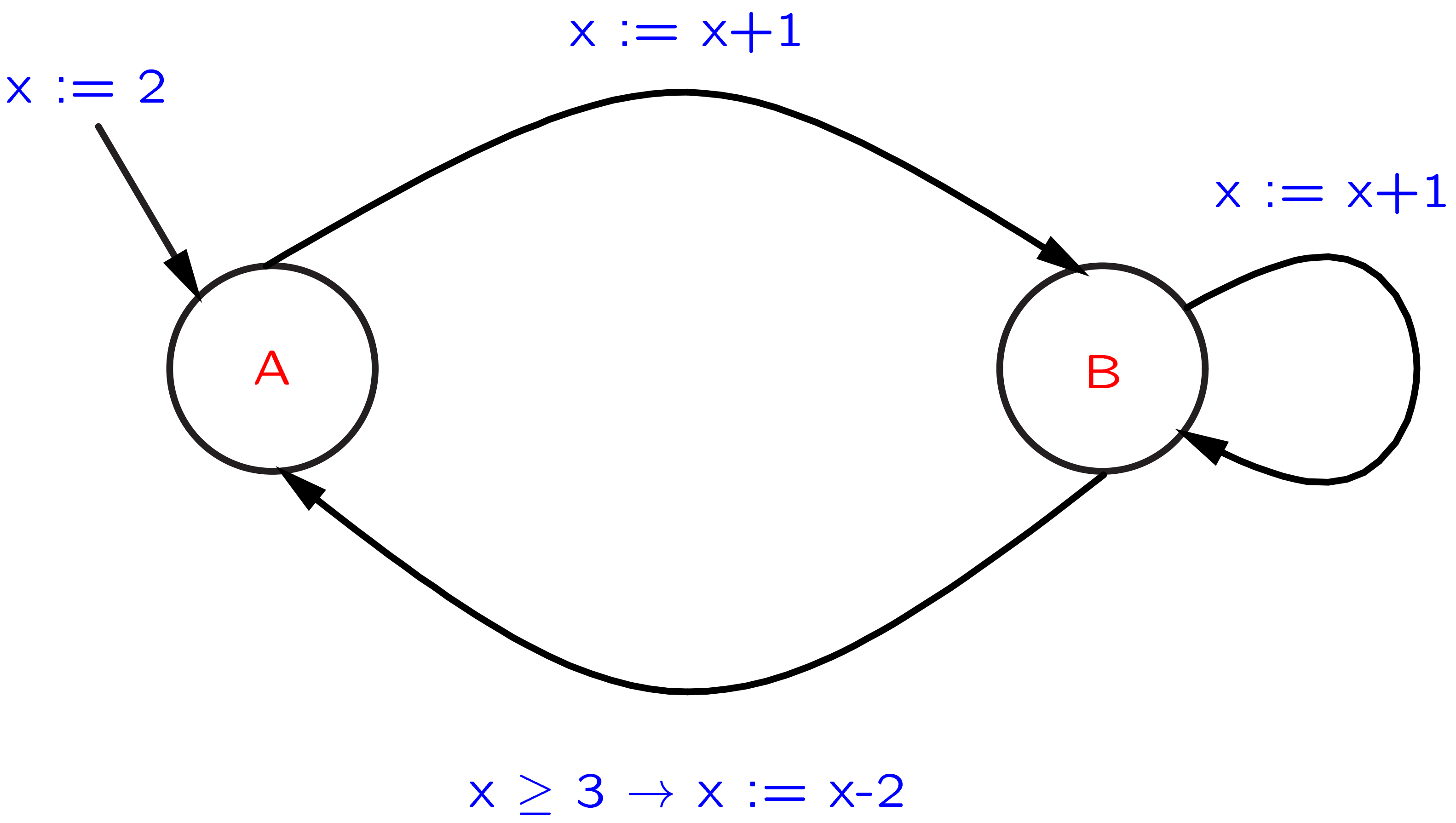
- Given a concrete system
- The idea is to develop an abstracted system that is faithful to the original
- Two broad classes
  (that can be seen as variants of each other)
  - Syntax-directed
    - ⋆ Abstract interpretation, slicing, downscaling
  - Semantic
    - ⋆ Their construction or justification requires deduction

Verification Via Property-Preserving Abstraction

- In general, we need a (finite) abstract state space with transition relation $\mathtt{tr_a}$
- And an abstraction function abs from the concrete state space to the abstract one
- And a predicate $\mathtt{p_a}$ on the abstract states
- Such that
  1. $\mathtt{init_c(cs)} \supset \mathtt{init_a(abs(cs))}$
  2. $\mathtt{tr_c(pre_c, post_c)} \supset \mathtt{tr_a(abs(pre_c), abs(post_c))}$
  3. $\mathtt{p_a(abs(cs))} \supset \mathtt{p_c(cs)}$
- Then
  - $\mathtt{invariant(p_a)(init_a, tr_a)} \supset \mathtt{invariant(p_c)(init_c, tr_c)}$
- And the antecedent can be proved by model checking

A Trivial Example

Show that when control is at B, then $\mathtt{x} \geq 2$

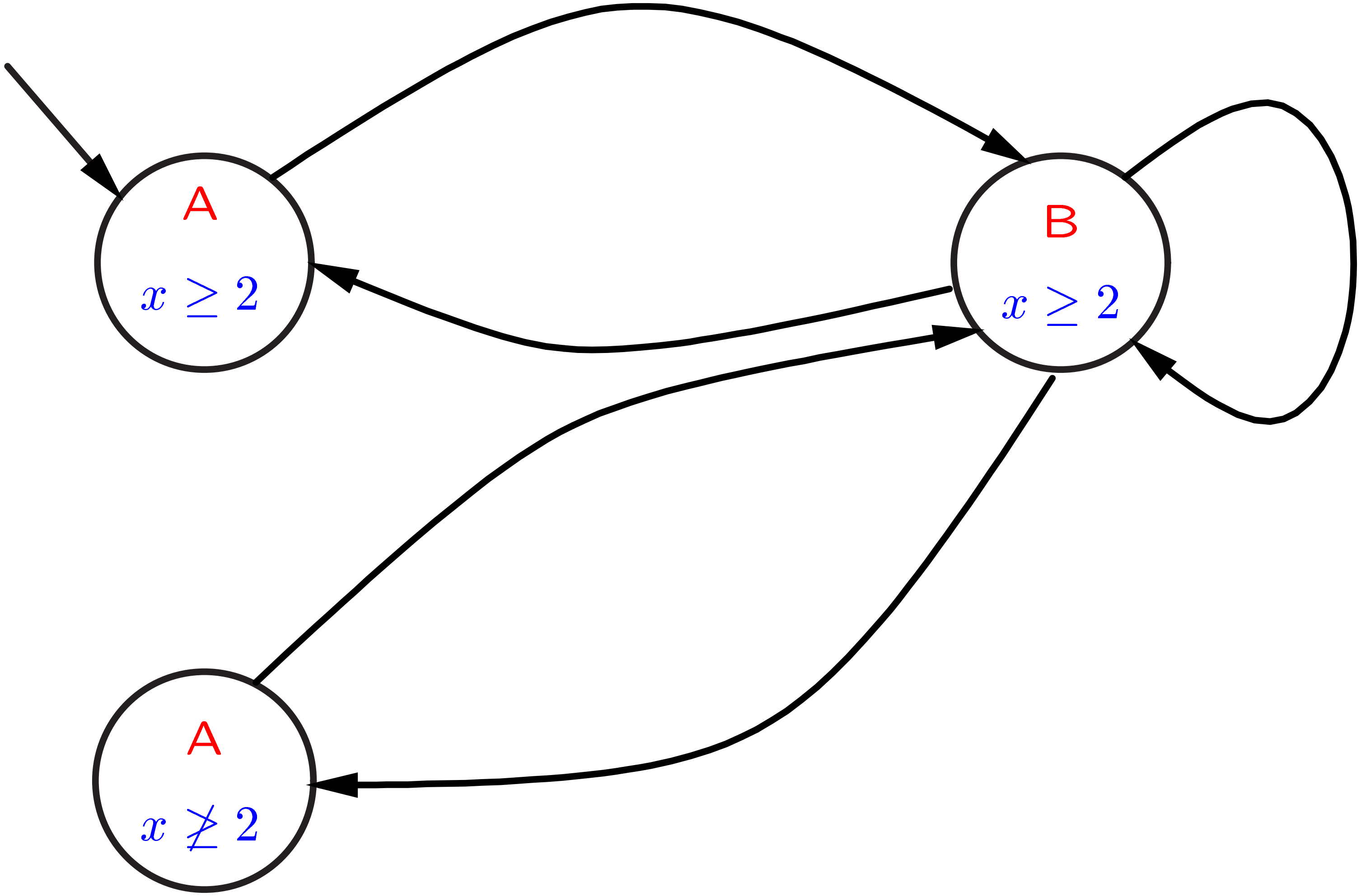


Boolean Abstraction

- Often convenient to choose an abstract state space consisting of
  - The control locations of the concrete system, plus
  - Some boolean state variables that correspond to predicates in the concrete system
- This is Boolean abstraction
- For the example, we'll have one abstract Boolean state variable corresponding to the concrete state predicate $\mathtt{x} \geq 2$

An Abstract Transition Relation For The Example

Constructed by hand



Clearly, the abstract invariant is satisfied

Verification Conditions for the Example Abstraction

- All trivial except number 2: default proof strategy yields

```
[-1]     pc(post_c!1) = B
[-2]     x(pre_c!1) = 0
  |-------
[1]      x(pre_c!1) + 1 ≥ 2
```

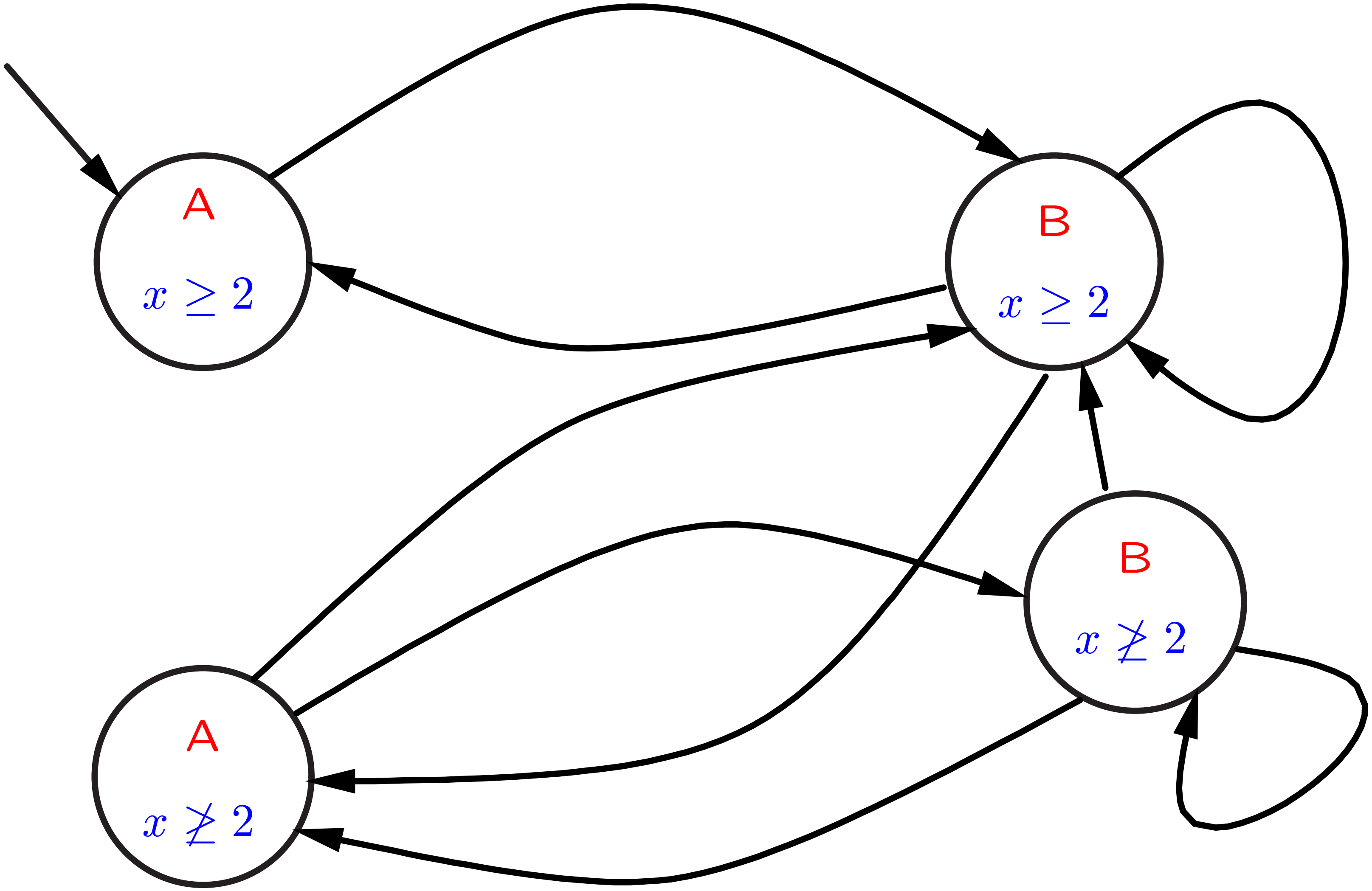
- I.e., Consider some transition into B from a state with $x = 0$
- Requires an invariant!
- Larger example: verification of Bounded Retransmission Protocol (BRP) by abstraction
  - Required 45 invariants to justify the abstraction
  - (But better than 57 required for direct invariant proof)

So What's To Be Done?

Calculate the abstract system (given the abstraction function) rather than "invent and verify"

- Saves manual effort of construction
- Abstract system is an abstraction (by construction)
- But may be too coarse to satisfy desired abstract invariant

Calculated Abstract Transition Relation For The Example



Abstract invariant is not satisfied

Diagnosing The Problem

- Model checking produces this counterexample trace
  - $\{\mathtt{A},\ x \geq 2\} \rightarrow \{\mathtt{B},\ x \geq 2\} \rightarrow \{\mathtt{A},\ x \not\geq 2\} \rightarrow \{\mathtt{B},\ x \not\geq 2\}$
- If we "concretize" this we see that the last transition is impossible in the concrete system
  - $\underset{2}{\{\mathtt{A},\ x \geq 2\}} \rightarrow \underset{3}{\{\mathtt{B},\ x \geq 2\}} \rightarrow \underset{1}{\{\mathtt{A},\ x \not\geq 2\}} \rightarrow \underset{2}{\{\mathtt{B},\ x \not\geq 2\}}$
- We see that it is important to know $\mathtt{x} \geq 1$ at $\mathtt{A}$
- So add another abstract state variable corresponding to $\mathtt{x} \geq 1$ and repeat
- This does it!

Making It Practical

(At least) two ways of calculating the abstracted system

- Start with universal transition relation; then for each arc
  - Generate the verification condition (VC) that allows it to be removed
  - Leave it in if cannot prove the VC
  - CAV paper by Abdulla et al.

  This approach preserves structure

- Develop the relation by a forward reachability analysis
  - At each point generate the VCs that lead to successor states with given predicate true resp. false
  - CAV paper by Saïdi and Shankar

  This approach usually has fewer states

It Is Practical!

- The trivial example in PVS

```
initial(s): bool = pc(s) = A and x(s) = 2
prop(s): bool = pc(s)=B => x(s)>=2
triv: lemma initial(s) implies AG(transition, prop)(s)
```

- Into the prover (uses the second method)

```
  |-------
{1}   FORALL (s: state): initial(s) IMPLIES AG(transition, prop)(s)

 Rule? (ABSTRACT-AND-MC ("LAMBDA s: x(s)>=2" "LAMBDA s:  x(s)>=1"))

 Starting computation of abstraction...abstraction computed in 4 seconds

 calls to the decision procedure: possible: 38   performed: 34

 MU simplification took 0.08 real, 0.05 cpu seconds
Q.E.D.
```

- Does BRP in a couple of minutes

Making It Really Practical

- Generate as many invariants as possible by static analysis and throw those into the proofs/calculations
  - Can easily deduce $\mathtt{x} \geq 1$ in the example
- Use heuristics to generate plausible initial abstractions
  - Boolean abstraction on (atomic) guard predicates
- Build tools for concretizing counterexample traces and checking them against the concrete system
  - To help distinguish between
    - ⋆ An excessively coarse abstraction
    - ⋆ A bug in the concrete system
- CAV paper by Das, Dill, and Park

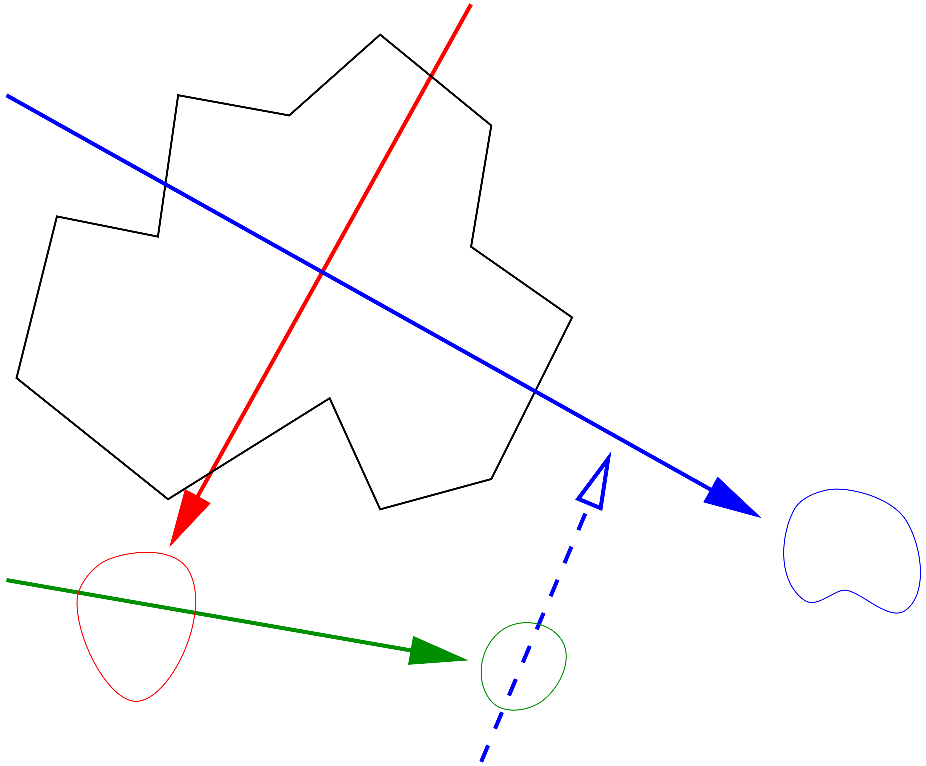
Iterated Analysis

- Model checkers often calculate the reachable stateset
  - Which is the strongest invariant

  And then throw it away
- The concretization of the reachable states of an abstraction is an invariant of the concrete system
  - And often a strong one
- Modify a model checker to return the reachable states as a formula that the theorem prover can manipulate
- Use simple abstractions to develop invariants that enable construction of finer ones
  - E.g., Boolean abstraction on $\mathtt{x} \geq 1$ in the example provides the invariant that enables construction of the fine abstraction on $\mathtt{x} \geq 2$

The Vision of Integration—IIb (reprise)

- Tools return symbolic values, not just true/false
  - E.g., reachable stateset discovered by a model checker
- Results from one analysis can then assist further ones



Next Steps

- Exploit the work and tools of the program analysis community
  - Slicers, abstract interpreters, etc.
- Find new ways to calculate invariants
  (the more you know, the more effective your abstractors)
  - Use decision procedures and quantifier elimination
  - Mix abstraction and fixpoint methods
- Extend the repertoire of analyzers:
  - SAT procedures (cf. Stålmarck), Herbrand automata, WS1S and other powerful decision procedures
- Put a proof manager on top
  - To keep track of what's been accomplished
  - To generate subgoals (abstract on those)

Summary

- Instead of trying to build ever more powerful tools
- Try to make the problems easier
  - Cut them down to a size that existing tools can handle
- By making ubiquitous use of automated abstraction
  - That is, construction of simpler descriptions that ignore/approximate aspects of the original
- Within a framework that allows multiple tools to cooperate
  - Generate models appropriate to different analyses and different tools from a single description
- Cooperation requires tools to exchange symbolic values, not just true/false verification outcomes

Credits

None of this work is mine; it is due to my colleagues

- Klaus Havelund: BRP example
- Hassen Saïdi: The Invariant Checker, PVS abstractor
- Saddek Bensalem, Yassine Lakhnech, Sam Owre: InVeSt
- Vlad Rusu and Eli Singerman: Mini-SAL experiments
- Shankar: SAL

    Being developed with David Dill (Stanford)

    And Tom Henzinger (Berkeley)

  Current implementation by César Muñoz and Eli Singerman

To Learn More

- Check out the CAV papers mentioned
- There is not much published on SAL, but precursor ideas and systems are described in papers and technical reports at `http://www.csl.sri.com/fm.html`
  - `~owre/cav98.html` and `~owre/cav98-tool.html` for InVeSt
  - `~rusu/tacas99.html` for mini-SAL experiments
  - `~saidi/Invariant-Checker/index.html` for the Invariant Checker
- Information about our verification system, PVS, and the system itself are available from `http://pvs.csl.sri.com`
  - Freely available under license to SRI
  - Allegro Lisp for Solaris, or Linux
  - Version 2.3 includes the abstractor