An Evidential Tool Bus

John Rushby

Computer Science Laboratory
SRI International
Menlo Park, California, USA

Formal Methods Integration

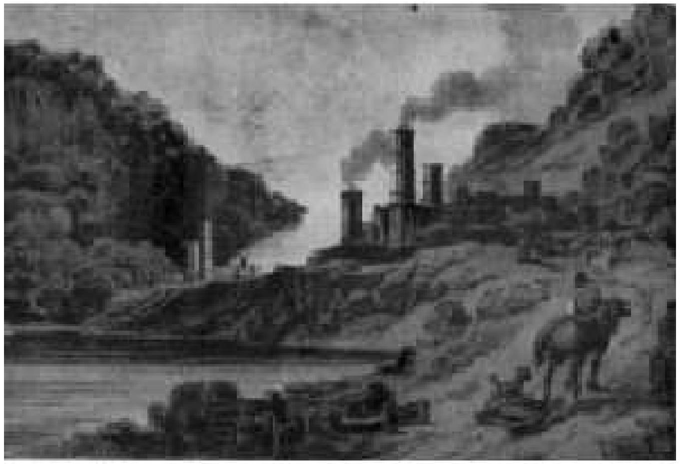
- No one notation, method, tool, or technology solves all problems
- Sometimes we need to make a selection
- And sometimes we need to use several in combination
- How to make multiple state of the art tools work together?
  - Want an architecture for tool integration that can be deployed today
  - But that will work with tools of 10, 15 years hence
- Note, I'm focused on tools and analysis
- I'll describe past and present, and a proposal for the future

Prehistory: No mechanized Tools

- Can use ad-hoc combinations of notations and methods
- Integration is informal, and best kept that way
- Little point in worrying about the fine details of semantic compatibility

Industrialization: Independent Tools

- Prehistorical notations and methods become supported by tools
- Typecheckers, static analyzers, theorem provers, model checkers
- Can continue to use ad-hoc combinations: e.g., model check before you prove
- And more integrated ones: e.g., prove an abstraction that can be model checked

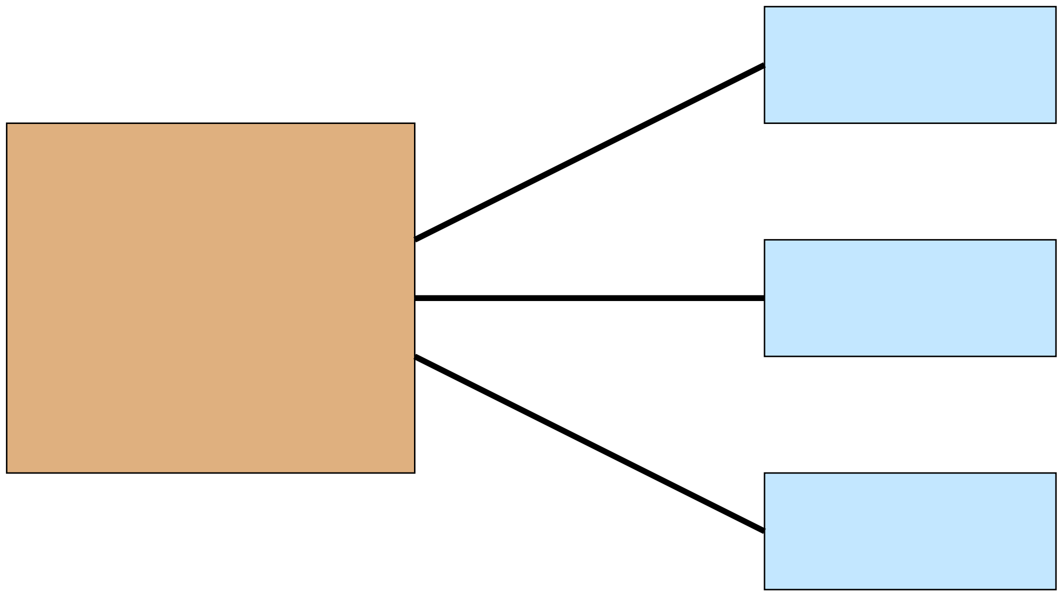


- Integration is still informal
- Managed by the human user

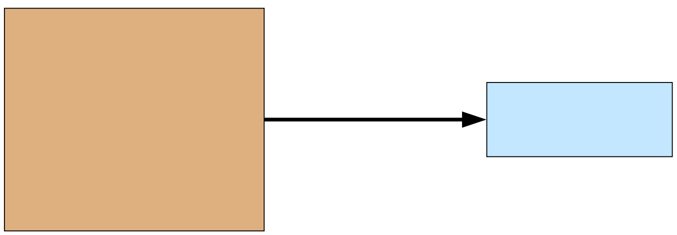
20th Century: Tools That Integrate Many Components

- A front end
  - Typically an interactive theorem prover
- Manages several backends
  - Decision procedures
  - Model checkers

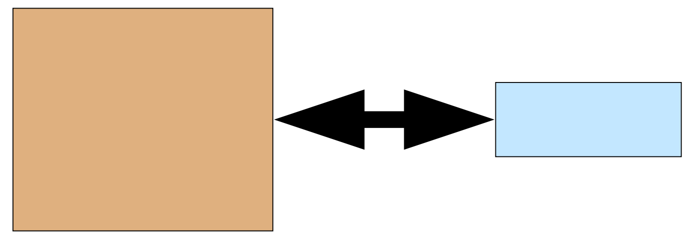
Simple Backend Architecture



Two Kinds of Backends



Endgame



Integrated

The Simple Case: Endgame Verifiers

- A higher level proof manager calls components (typically, decision procedures) to discharge subgoals
- Components return only `verified` or `unverified`
  - Embellishments: proof objects and counterexamples
- But the information returned on failure does not guide the higher-level proof search
  - Other than to cause it to try something else
  - Hence endgame verifiers

Endgame Verifier Examples

1979: Stanford Pascal Verifier and STP used decision procedures for combinations of theories including arithmetic (STP gave rise to Ehdm, then PVS)

1995: PVS used a BDD-based symbolic model checker

1999: PVS used a predicate abstractor

2000: PVS used Mona for WS1S

Not only did theorem provers use model checkers as backends, some model checkers grew a front-end theorem prover

1998: Cadence SMV had a proof assistant that generated model checking subproblems by abstraction and composition

And some other systems used an entire interactive theorem prover for the endgame

1999: VSDITLU: used PVS backend to check side conditions on Symbolic Definite Integral Table Look-Up in Maple

Integrating Endgame Verifiers

It's pretty simple

- Provide higher level proof strategies that decompose proof goals into subgoals that can be steered towards the competence of the endgame verifier(s)
- Provide a recognizer for proof goals within the competence of an endgame verifier
- Provide glue code to translate suitable proof goals into the input of an endgame verifier and to interpret its output

Many classes of endgame verifiers are being honed through competition

- Improves performance (be careful)
- Standardizes interfaces
- FO provers, BDD packages, SAT solvers, SMT solvers

When Endgame Is Not Enough

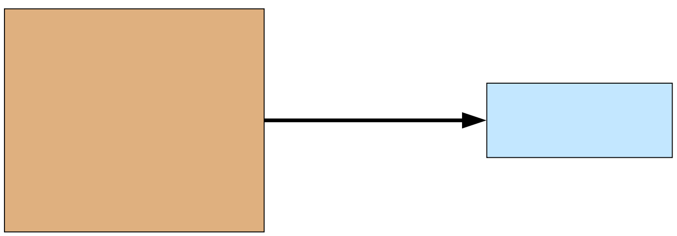
- When you need interaction across multiple backends
- Or when you need massive interaction between front and back ends
- Example: fault-tolerant real-time systems

  Fault tolerance: case explosion; Needs BDDs or SAT

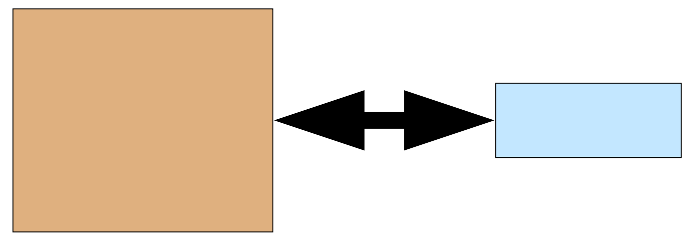
  Real time: continuous time; needs real arithmetic or timed automata

  Need tight interaction between these: loose combination will not do it

Recap: Two Kinds of Backends



Endgame



Integrated

A Difficult Case: Tightly Integrated Backends

- Endgame verifiers are easy to integrate because they do not interact with higher level proof search (nor with each other)
- In fact, they are barely integrated
- Tight integration is much harder
- Classic Boyer-Moore 1986 paper describes tight integration of linear arithmetic decision procedure with Nqthm
    - Two pages of code for endgame decision procedure
    - Became 60 for integrated version
- PVS takes an intermediate path
    - Decision procedures are integrated with the rewriter
    - And used in simplification
- A tractable case is the integration of decision procedures with each other . . .

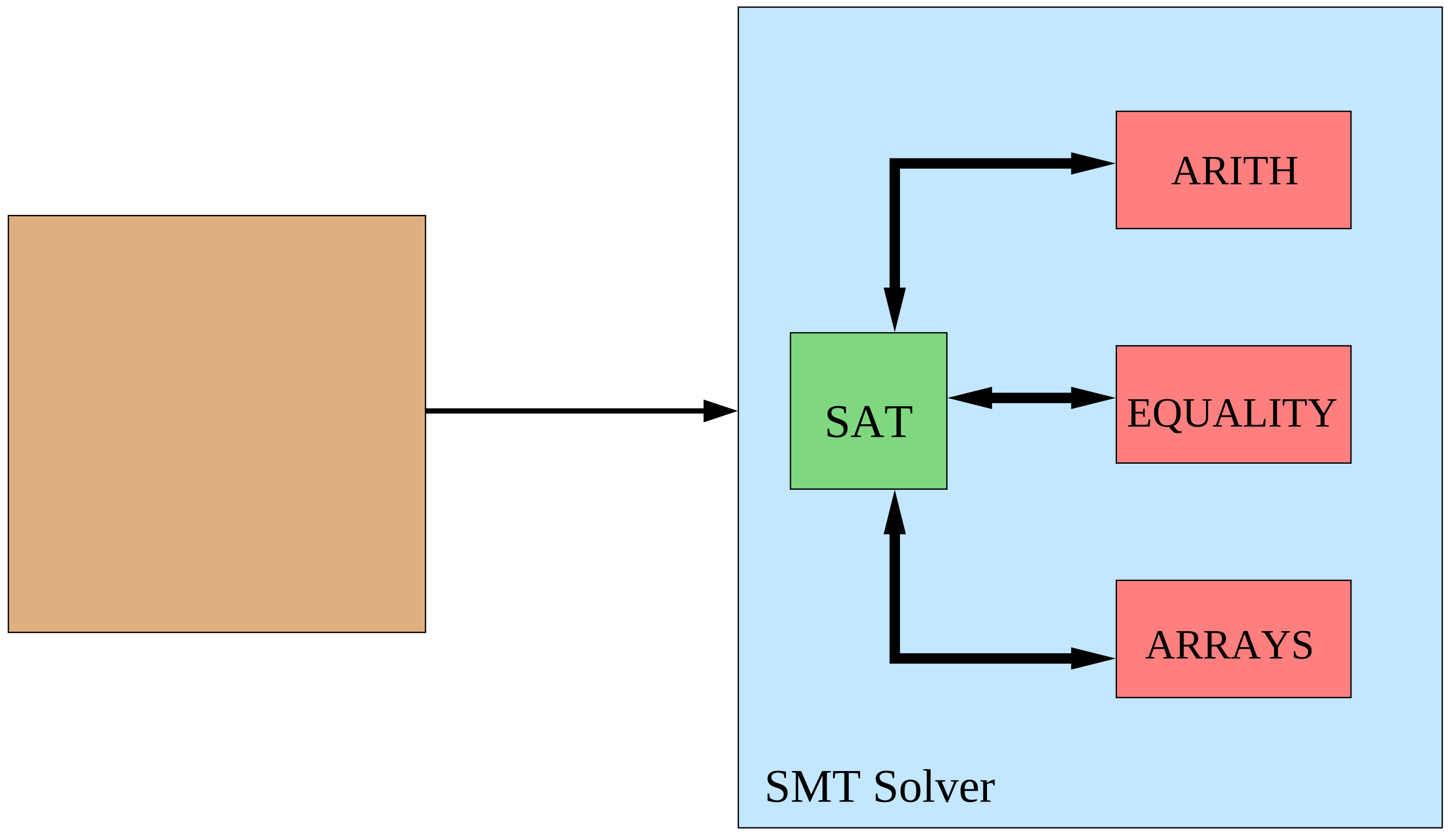
The Present Day

Three trends:

- Evolved (internally integrated) backends
- Scriptable components
- Customized integrations

Evolution of Endgame Verifiers

Inside an Evolved Endgame Verifier



Inside an Evolved Endgame Verifier

- Individual decision procedures decide conjunctions of formulas in their decided theories
- Combinations of decision procedures (using, e.g., Nelson-Oppen or Shostak methods) decide conjunctions over the combined theories
- What if we have richer propositional structure
  - E.g., $(x \leq y \vee y = 5) \wedge (x < 0 \vee y \leq x) \wedge x \neq y$

    . . . possibly continued for 1000s of terms
- Should exploit search strategies of modern SAT solvers
- So replace the terms by propositional variables
  - $(A \vee B) \wedge (C \vee D) \wedge E$
- Get a solution from a SAT solver (if none, we are done)
  - E.g., A, D, E

Inside an Evolved Endgame Verifier (ctd)

- Restore the interpretation of variables and send the conjunction to the core decision procedure
  - E.g., $x \leq y \wedge y \leq x \wedge x \neq y$
- If satisfiable, we are done
- If not, ask SAT solver for a new assignment—but isn't it expensive to keep doing this?
- Yes, so first, do a little bit of work to find fragments that explain the unsatisfiability, and send these back to the SAT solver as additional constraints (i.e., lemmas)
  - $A \wedge D \supset \neg E$
- Iterate to termination
- We call this "lemmas on demand" or "lazy theorem proving"
- it works really well
- Yields satisfiability modulo theories (SMT) solvers—e.g., ICS

Evolution of Endgame Verifiers (ctd.)

- One path grows the endgame verifier and specializes and shrinks the higher-level proof manager
- Example:
  - SAL language has a type system similar to PVS, but is specialized for specification of state machines
    (as transition relations)
  - The SAL infinite-state bounded model checker uses an SMT solver (ICS), so handles specifications over reals and integers, uninterpreted functions
  - Often used as a model checker (i.e., for refutation)
  - But can perform verification with a single higher level proof rule: k-induction (with lemmas)
  - Note that counterexamples help debug invariant

Bounded Model Checking

- Given a system specified by initiality predicate I and transition relation T on states S, finding a counterexample of length k for P (or test case for $\neg P$) requires a sequence of states $s_0, \ldots, s_k$ such that

$$I(s_0) \wedge T(s_0, s_1) \wedge T(s_1, s_2) \wedge \cdots \wedge T(s_{k-1}, s_k) \wedge \neg P(s_k)$$

- Given a Boolean encoding of I and P (i.e., a circuit), this is a propositional satisfiability (SAT) problem
- If T and P are defined over infinite types such as reals, integers, datatypes, or use symbolic functions or constants, then need to solve the BMC SAT problem over these theories
- That's what an SMT solver does

k-Induction

- Ordinary inductive invariance (for P):

  Basis: $I(s_0) \supset P(s_0)$

  Step: $P(r_0) \wedge T(r_0, r_1) \supset P(r_1)$

- Extend to induction of depth k:

  Basis: No counterexample of length k or less

  Step: $P(r_0) \wedge T(r_0, r_1) \wedge P(r_1) \wedge \cdots \wedge P(r_{k-1}) \wedge T(r_{k-1}, r_k) \supset P(r_k)$

  These are close relatives of the BMC formulas

- Induction for $k = 2, 3, 4 \ldots$ may succeed where $k = 1$ does not

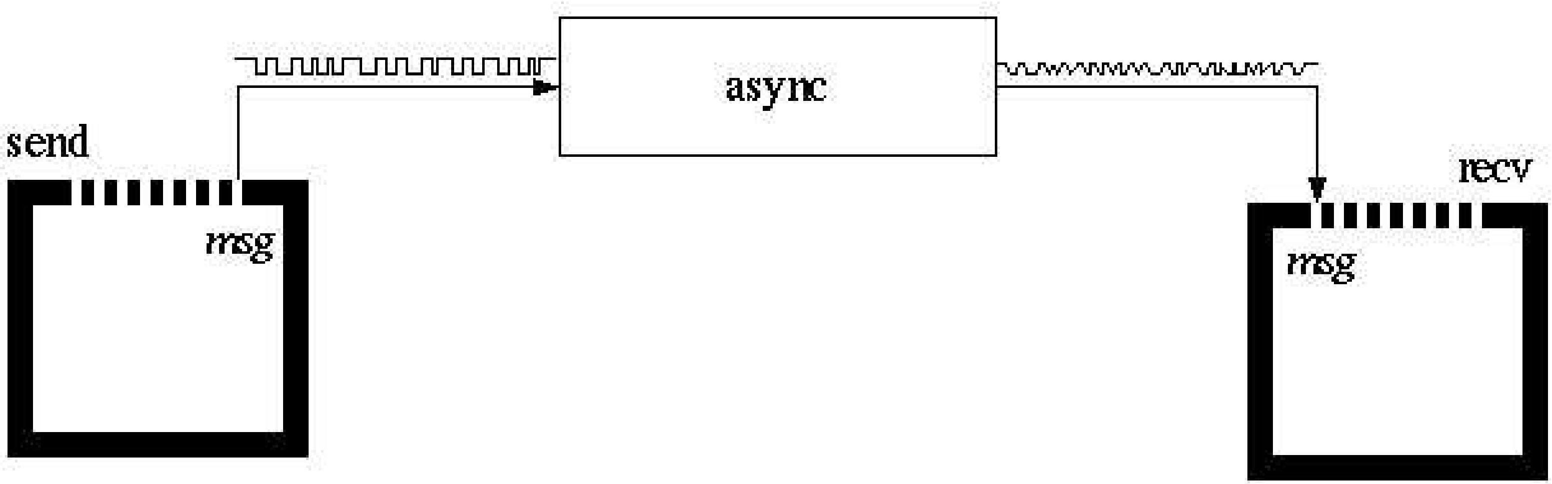
- Is complete for some problems (e.g., timed automata)

- Fast, too, e.g., Fischer with 43 processes

Performance of Evolved Endgame Verifiers

- Biphase Mark Protocol is an algorithm for asynchronous communication
  - Clocks at either end may be skewed and have different rates, jitter
  - So have to encode a clock in the data stream
  - Used in CDs, Ethernet
  - Verification identifies parameter values for which data is reliably transmitted
- Verified in ACL2 by J Moore (1994)
- Three different verifications used PVS
  - One by Groote and Vaandrager used PVS + UPPAAL
  - Required 37 invariants, 4,000 proof steps, hours of prover time to check

Biphase Mark Protocol



Performance of Evolved Endgame Verifiers (ctd.)

- Brown and Pike recently did it with `sal-inf-bmc`
  - Used timeout automata to model timed aspects
  - Statement of theorem discovered systematically using disjunctive invariants (7 disjuncts)
  - Three lemmas proved automatically with 1-induction,
  - Theorem proved automatically using 5-induction
  - Verification takes seconds to check
- Adapted verification to 8-N-1 protocol (used in UARTs)
  - Additional lemma proved with 13-induction
  - Theorem proved with 3-induction (7 disjuncts)

Integrated Decision Procedures and SMT Solvers

- Long line of research on integrating decision procedures for separate theories so they decide the combined theory
  - Starts with Nelson-Oppen and Shostak methods
  - Activity continues today: theory, presentation, verification, and pragmatics
- Recently extended through integration with SAT solving to yield SMT solvers
  - Interactions are intense (millions per verification)
  - Information from decision procedures must be used efficiently to prune SAT search
  - Impacts design of individual decision procedures
  - Engineering choices explored through benchmarking and competition
- Homogeneous integration: not quite solved, but on the way

Scriptable Components

- Many backend components have standardized APIs
- Can often create special instantiations with a wrapper
- Several modern model checkers are scriptable
- Have a core API implemented efficiently (e.g., in C)
- E.g., `Bogor`, `SAL`
  - SAL model checkers are scripts in Scheme over this API
  - Easy to create new capabilities by writing new scripts
  - E.g., `sal-atg` automated test generator

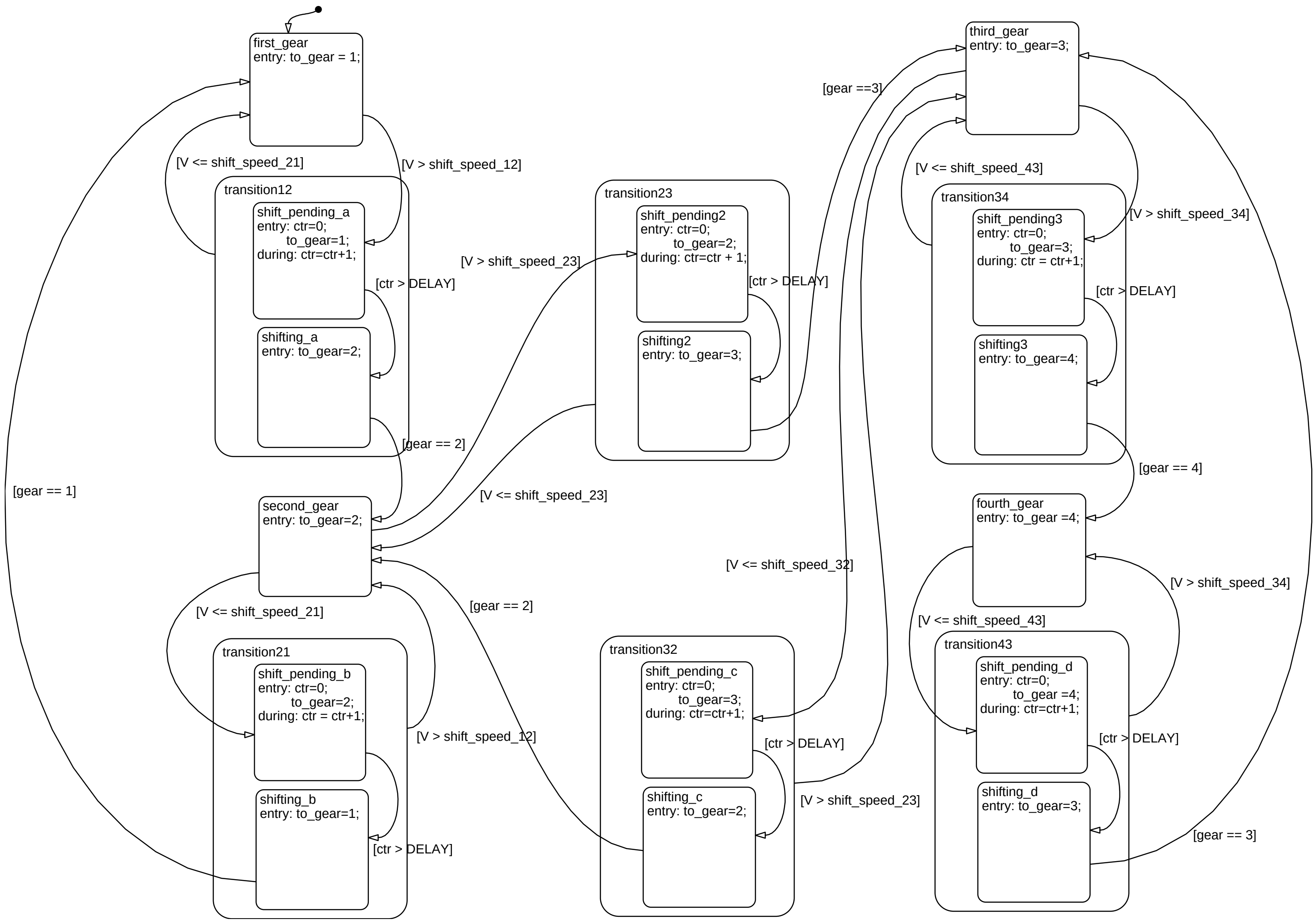
Core Of The SAL-ATG Test Generation Script

```
(define (extend-search module goal-list
         path scan prune innerslice start step stop)
   (let ((new-goal-list (if prune (goal-reduce scan goal-list path)
                              (minimal-goal-reduce scan goal-list path))))
    (cond ((null? new-goal-list) (cons '() path))
          ((> start stop) (cons new-goal-list path))
          (else
           (let* ((goal (list->goal new-goal-list module))
                  (mod (if innerslice
                         (sal-module/slice-for module goal) module))
                  (new-path
                   (let loop ((depth start))
                        (cond ((> depth stop) '())
                              ((sal-bmc/extend-path
                                  path mod goal depth 'ics))
                              (else (loop (+ depth step)))))))
             (if (pair? new-path)
                 (extend-search mod new-goal-list new-path scan
                                prune innerslice start step stop)
               (cons new-goal-list path)))))))
```

Outer Loop Of The SAL-ATG Test Generation Script

```
(define (iterative-search module goal-list
          scan prune slice innerslice bmcinit start step stop)
  (let* ((goal (list->goal goal-list module))
         (mod (if slice (sal-module/slice-for module goal) module))
         (path (if bmcinit
                   (sal-bmc/find-path-from-initial-state
                       mod goal bmcinit 'ics)
                 (sal-smc/find-path-from-initial-state mod goal))))
     (if path
         (extend-search mod goal-list path scan prune
             innerslice start step stop)
         #f)))
```

Example Shift Scheduler



Customized Integrations

Static analyzers (e.g., SDV) often integrate many components

For example, software model checkers generally have:

- C front end with CFG analyzer
- Predicate abstractor
  - Which uses decision procedures
  - And possibly a model checker
- Model checker and counterexample generator
- Counterexample concretizer and refinement generator
  - Which uses Craig interpolation

And a control loop around the whole lot

Customized Integrations (ctd.)

- The components exchange more than just proof judgments
  - Traces
  - Counterexamples
  - Abstractions
  - Sets of predicates
- And interact in more complex ways than front end/backend

The Future

- Expect many different high-performance components
- And many scripting options
- And lots of cool ideas for using them
- We need an architecture!
- That will remain good for 15 years or more

Integration of Heterogeneous Components

- Modern formal methods tools do more than verification
- They also do refutation (bug finding)
- And test-case generation
- And controller synthesis
- And construction of abstractions and abstract interpretation
- And generation of invariants
- And . . .
- Observe that these tools can return objects other than verification outcomes
  - Counterexamples, test cases, abstractions, invariants

  Hence, heterogeneous

Integration of Heterogeneous Components

- LAST (Xia, DiVito, Muñoz) generates MC/DC tests for avionics code involving nonlinear arithmetic (with floating point numbers, trigonometric functions etc.)
- Applied it to Boeing autopilot simulator
  - Modules with upto 1,000 lines of C
  - 220 decisions
- Generated tests to (almost) full MC/DC coverage in minutes

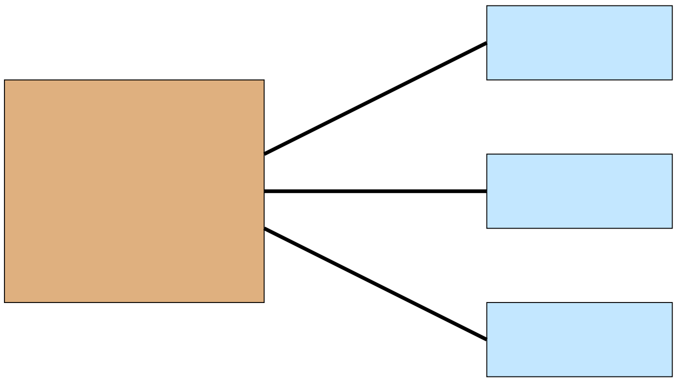
Structure of LAST

- It's built on Blast (Henzinger et al)
  - A software model checker, itself built of components
  - Including CIL and CVC-Lite
- But extends it to handle nonlinear arithmetic using RealPaver (a numerical nonlinear constraint unsatisfiability checker)
  - Added 1,000 lines to CIL front end for MC/DC
  - Added 2,000 lines to RealPaver to integrate with CVC-Lite (Nelson-Oppen style)
  - Changed 2,000 lines in Blast to tie it all together

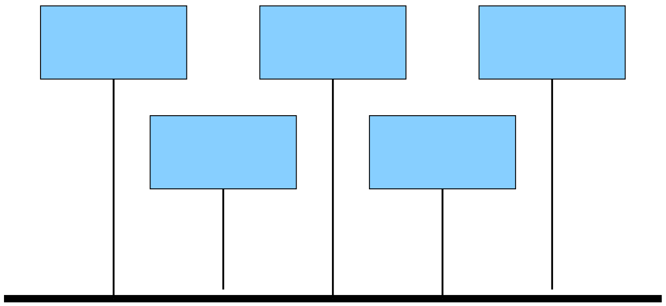
A Tool Bus

- How can we construct these customized combinations and integrations easily and rapidly?
- The integrations are coarse-grained (hundreds, not millions of interactions per analysis), so they do not need to share state
- So we could take the outputs of one tool, massage it suitably and pass it to another and so on
- A combination of XML descriptions, translations, and a scripting language could probably do it
- Suitably engineered, we could call it a tool bus

From Backends to Bus



Backends



Bus

- Bus is a federation of equals
- Theorem prover is just another component

But . . .

- But we'd need to know the names and capabilities of the tools out there and explicitly to script the desired interactions
  - And we'd be vulnerable to change
- Whereas I would like to exploit whatever is out there
  - And in 15 years time there may be lots of things out there
- That is, I want the bus to operate declaratively
  - By implicit invocation
- And I want evidence that supports the overall analysis
  (i.e., the ingredients for a safety or assurance case)
- That is, I want a semantic integration

A Formal Tool Bus

- The data manipulated by tools on bus are formulas in logic
- In fact, they can be seen as formulas in a logic
  - The Formal Tool Bus Logic
  - Each tool operates on a sublogic
  - Syntactic differences masked with XML wrappers
- No point in limiting the expressiveness of the tool bus logic
  - Should be at least as expressive as PVS
    - ⋆ Higher order, with predicate, structural, and dependent subtypes, abstract data types, recursive and inductive definitions, parameterized theories, interpretations
  - With structured representations for important cases
    - ⋆ State machines (as in SAL), counterexamples, process algebras, temporal logics . . .
    - ⋆ Handled directly by some tools, can be expanded to underlying semantics for others

Tool Bus Judgments

The tools on the bus evaluate and construct predicates over expressions in the logic—we call these judgments

Parser: A is the AST for string S

Prettyprinter: S is the concrete syntax for A

Typechecker: A is a well-typed formula

Finiteness checker: A is a formula over finite types

Abstractor to PL: A is a propositional abstraction for B

Predicate abstractor: A is an abstraction for formula B wrt. predicates ϕ

GDP: A is satisfiable

GDP: C is a context (state) representing input G

SMT: ρ is a satisfying assignment for A

Tool Bus Queries

- Tools publish their capabilities and the bus uses these to organize answers to queries

  Query: `well-typed?(A)`

  Response: `PVS-typechecker(...)  |- well-typed?(A)`

  The response includes the exact invocation of the tool concerned

- Queries can include variables

  Query: `predicate-abstraction?(a, B,` ϕ`)`

  Response:

  `SAL-abstractor(...) |-predicate-abstraction?(A, B,` ϕ`)`

  The tool invocation constructs the witness, and returns its handle A

Tool Bus Operation

- The tool bus operates like a distributed datalog framework, chaining on queries and responses
- Similar to SRI AIC's Open Agent Architecture
  - And maybe similar to MyGrid, Linda, . . . ?
- Can have hints, preferences etc.
- Tools can be local or remote
- Tools can run in parallel, in competition
- The bus needs to integrate with version management

Scripting

Three levels of scripting

Tools:

- Tools should be scriptable
- Better functionality, performance than wrappers
- E.g., SAL model checkers are Scheme scripts over an API
- Test generator is another script over the same API

Wrappers:

- Some functionality can be achieved by a little programming and maybe some tool invocation

Tool Bus:

- Scripts are chains of judgments

Tool Bus Scripts

- Example
  - If A is a finite state machine and P a safety property, then a model checker can verify P for A
  - If B is a conservative abstraction of B, then verification of B verifies A
  - If A is a state machine, and B is predicate abstraction for A, then B is conservative for A
- How do we know this is sound?
- And that we can trust the computations performed by the components?

An Evidential Tool Bus

- Each tool should deliver evidence for its judgments
  - Could be proof objects (independently checkable trail of basic deductions)
  - Could be reputation (“Proved by PVS”)
  - Could be diversity (“using both ICS and CVC-Lite”)
  - Could be declaration by user
    - “Because I say so”
    - “By operational experience”
    - “By testing”
- And the tool bus assembles these (on demand)
- And the inferences of its own scripts and operations
- To deliver evidence for overall analysis that can be considered in a safety or assurance case—hence evidential tool bus

The Evidential Tool Bus

- There should be only one evidential tool bus
- Just like only one WWW
- How to do it?
  - Standards committee?
  - Competition and cooperation!
- Probably not difficult to integrate multiple buses
  - Need agreement on ontologies
  - Fairly minimal glue code to link them together
- We'll be building one
  - Initially to integrate PVS and SAL
  - And to reconstruct Hybrid-SAL
- Will appreciate your input, and hope you'll like to use it, and to attach your tools

Thank you!

- And thanks to Bruno Dutertre, Grégoire Hamon, Leonardo de Moura, Sam Owre, Harald Rueß, Hassen Saïdi, N. Shankar, Maria Sorea, and Ashish Tiwari
- You can get our tools and papers from `http://fm.csl.sri.com`