Combining Complementary Formal Verification Strategies to Improve Performance and Accuracy

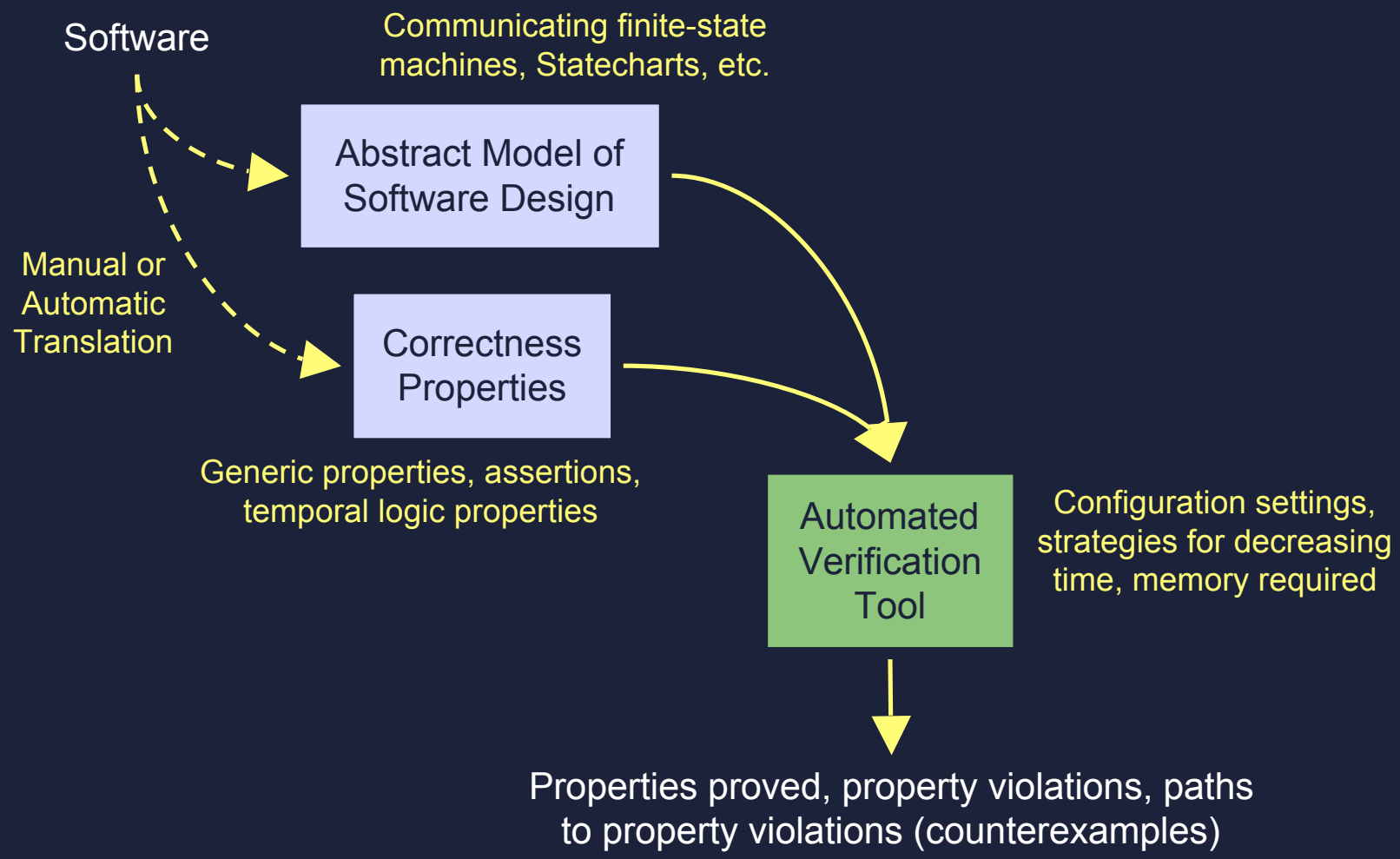
David Owen

June 15, 2007

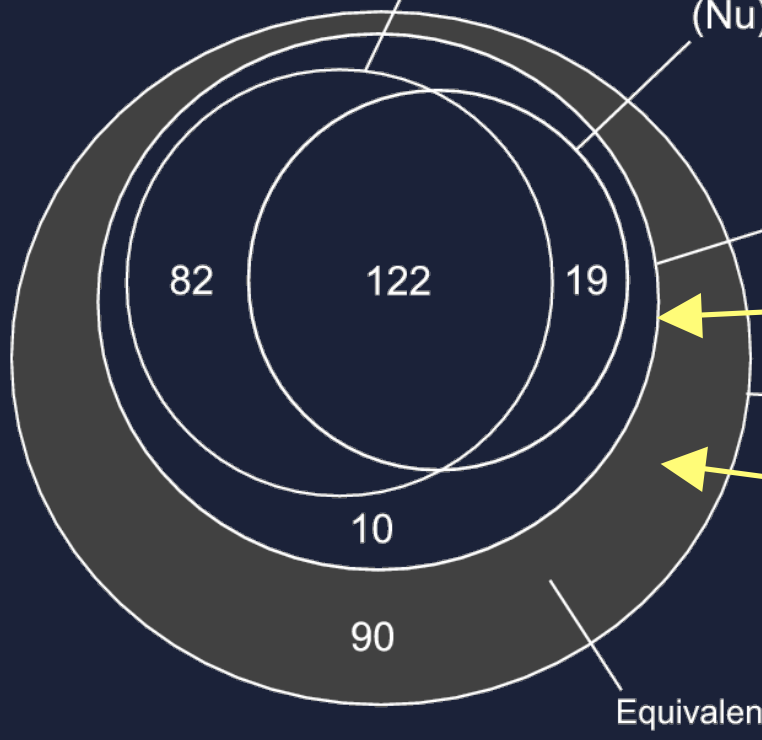
Overview

- Four Key Ideas
  - A Typical Formal Verification Strategy
  - Complementary Verification Capability
  - Complementary Performance
  - Performance of Combined Strategy
- Introduction
- Related Work
- Motivating Examples
- Case Study
- Conclusion

A Typical Formal Verification Strategy



Complementary Verification Capability



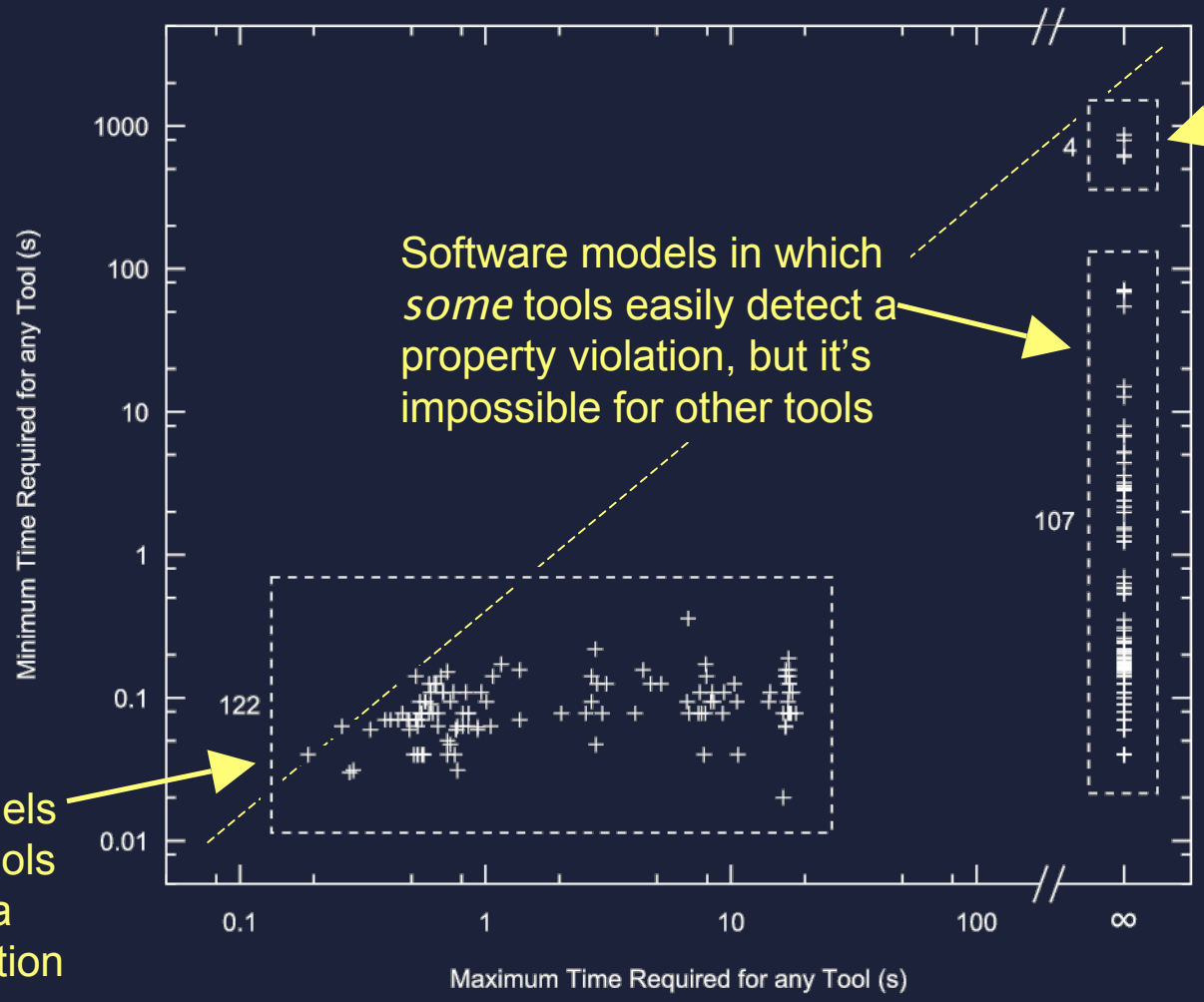
Three verification strategies:
1. Random search
2. Symbolic model checking
3. Explicit-state model checking

Sets of fault-seeded software models for which each strategy could detect a property violation

Remaining software models, which contain no property violations

Overlap = opportunity to combine strategies

Complementary Performance



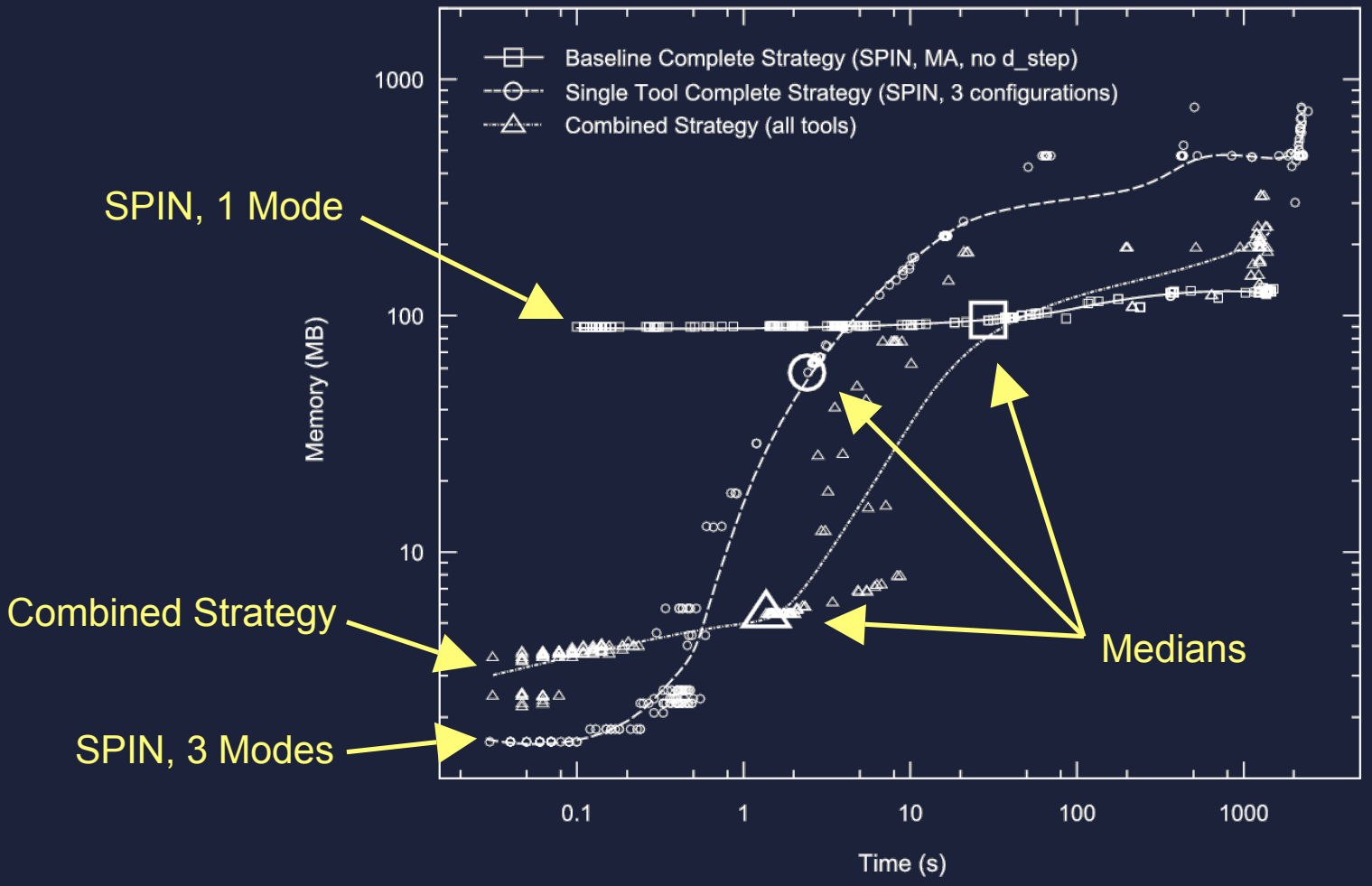
Software models for which it's hard or impossible for any tool to detect a property violation

Software models in which *some* tools easily detect a property violation, but it's impossible for other tools

Software models in which *all* tools easily detect a property violation

Nearly all models not easily checked by all tools are easy for at least one tool

Performance of Combined Strategy



Strategy combining multiple tools is faster and often requires less memory than single-tool alternative strategies

Overview (2)

- Three Key Results
- Introduction
  - Background
  - Combining Complementary Verification Strategies
  - Previous Work
  - Contributions
- Related Work
- Motivating Examples
- Case Study
- Conclusion

Background

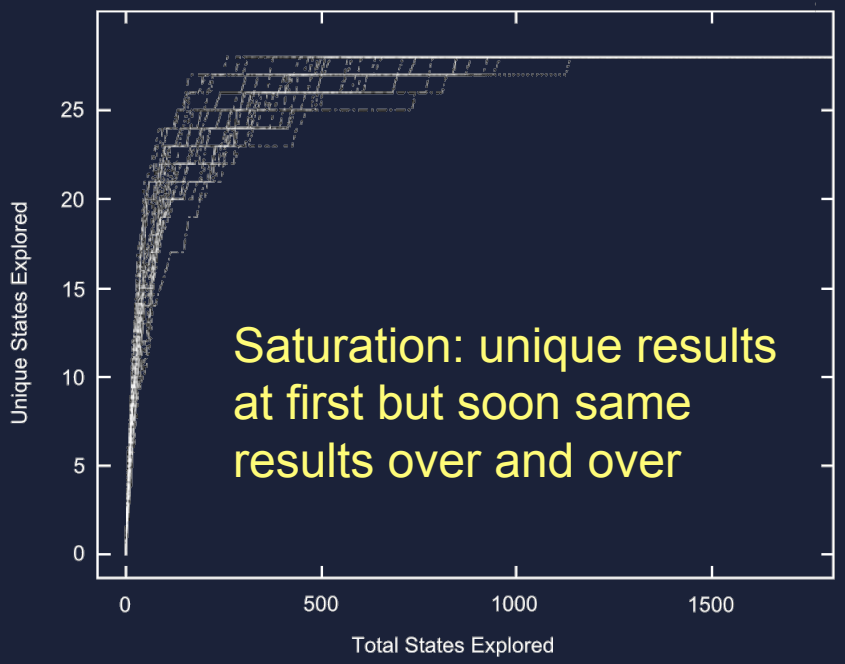
- Increasingly complex software, increasingly critical applications
- Powerful but costly verification methods
  - Cost in user effort and verification expertise, domain knowledge
  - Cost in time and memory required to run automated tools
- Many different strategies for decreasing these costs
  - Strategies for decreasing modeling effort
  - Strategies for more efficient verification

Combining Complementary Verification Strategies

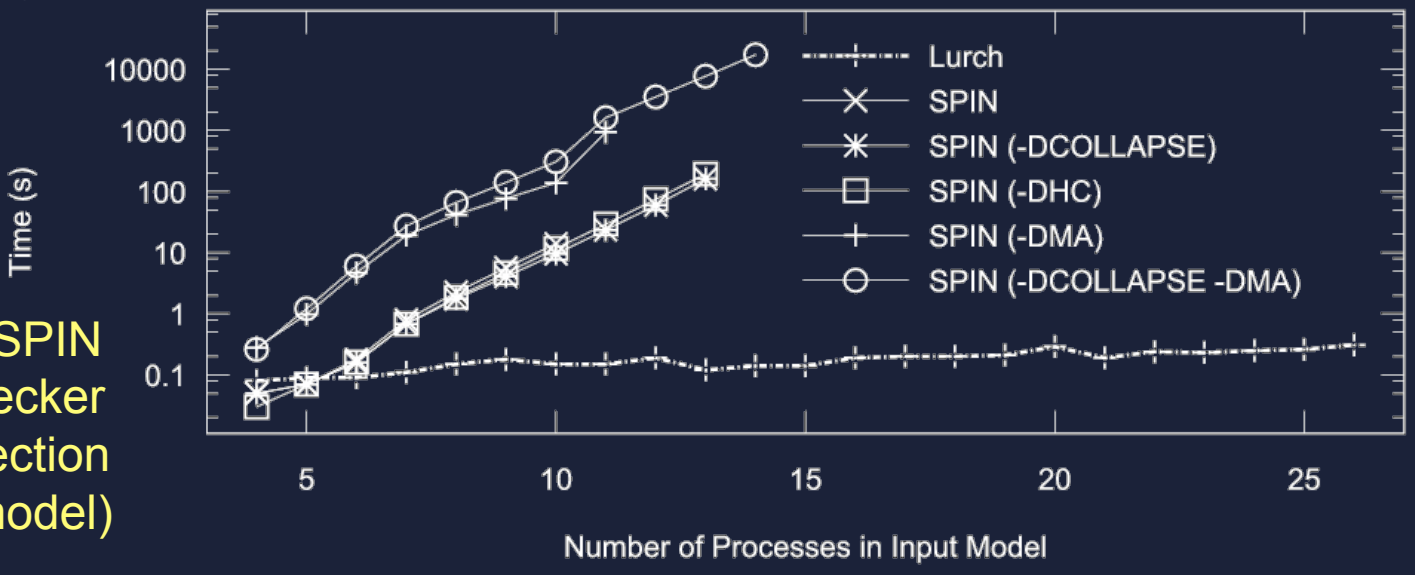
- Alternative strategies have different strengths
  - How to choose the right strategy? (Cobleigh et.al.)
  - Here we propose to combine complementary strategies
- Benefits of combining strategies
  - Performance
    - Strategies can be *cascaded*—try a quick, efficient method first; try resource-hungry methods later
  - Accuracy
    - Different strategies have complementary verification capabilities
    - Expose hidden assumptions, improve understanding of individual strategies
    - Increase confidence in verification results
  - Ease of use
    - Less burden on user to choose between strategies
    - Less burden on user to understand idiosyncrasies of individual strategies

Previous Work

- Random search for debugging formal models
  - Consistent results when run to *saturation*
  - Efficient detection of property violations



Saturation: unique results at first but soon same results over and over



Lurch vs. SPIN Model Checker (leader election protocol model)

Contributions

- Specific
  - Effective verification strategy for SCR software specifications
  - Exploiting complementary capability and performance of verification tools integrated with SCR Toolset
- General
  - Justification for future work to develop multiple-tool verification strategies for other types of formal models
- Additional contributions
  - Lurch random search tool for debugging formal models
  - Automatic translation from SCR to Lurch
  - Lessons learned in use of individual verification tools
  - How to get accurate results using SPIN and NuSMV model checkers on SCR specifications

Overview (3)

- Three Key Results
- Introduction
- Related Work
  - Testability
  - Verification
  - Random Search
- Motivating Examples
- Case Study
- Conclusion

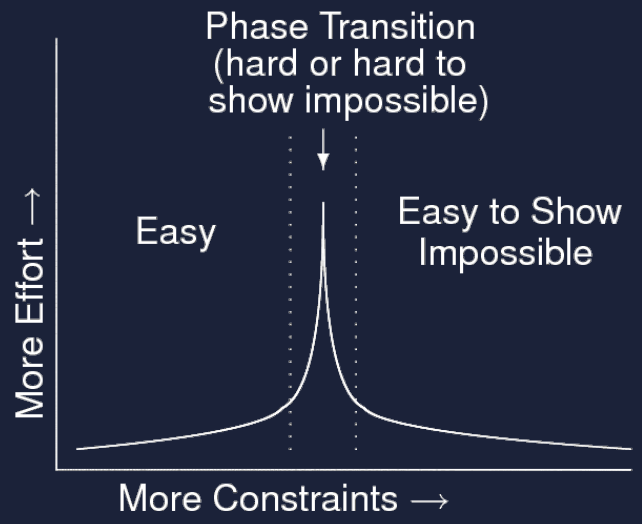
Testability

- Definitions
  - The degree to which a program facilitates the establishment of test criteria and performance of tests (IEEE Glossary of Software Engineering Terminology)
  - The probability a program will fail under test, if it contains at least one fault (Voas and Miller, Bertolino and Stringini)
  - Reachability: for a more testable program, fewer tests exercise more behavior (Menzies and Cukic)
- Testability in the context of random search
  - Saturation – quick rise to a high plateau
- Testability in the context of alternative testing strategies
  - Testability is relative to testing strategy
  - A given program will be most testable by a combination of complementary strategies

Verification

- Formal methods
  - Powerful but costly in user effort, expertise
- Model Checking
  - Easier to use (although still difficult), but costly in time and memory requirements
- Complete Strategies for Improving Scalability
  - BDDs (SMV), partial order reduction (SPIN), state compression
- Incomplete Strategies
  - Bounded model checking (SMV), lossy state compression, random search
  - Use of model checking terminated early
- More powerful testing tools (which scale to, e.g., source code) inspired by ideas from Model Checking
- Many strategies for improving scalability—many different strengths and weaknesses

Random Search



- Randomized algorithms
  - Simplicity, robustness, efficiency, effectiveness
  - But not repeatable, not guaranteed to find the best solution
- Problem structures (theoretically) favorable to random search
  - Because of a *phase transition*
    - Worst-case problem instances a small subset of all instances
  - Because of *funnels*
    - A small subset of key variables largely determine the behavior of everything else
- Random search used to debug protocol models (West)
  - Surprisingly successful
  - Faults (much) less complex than the systems they reside in?

Overview (4)

- Three Key Results
- Introduction
- Related Work
- Motivating Examples
  - Inconsistent Results (3 Examples)
  - Performance Variations (4 Examples)
- Case Study
- Conclusion

Inconsistent Results
(from alternative verification tools)

- Cadence SMV and NuSMV
  - NuSMV missed error detected by Cadence SMV
  - Automatic translator output fine for Cadence SMV, but not right (although syntactically correct) for NuSMV
- SPIN and Lurch
  - SPIN (complete tool) missed error detected by Lurch (incomplete tool)
  - Translator to SPIN used invalid *d_step*
- SPIN and Salsa
  - SPIN reported violation of property proved true by Salsa
  - *NATURE* constraint in SCR model ignored by translator to SPIN

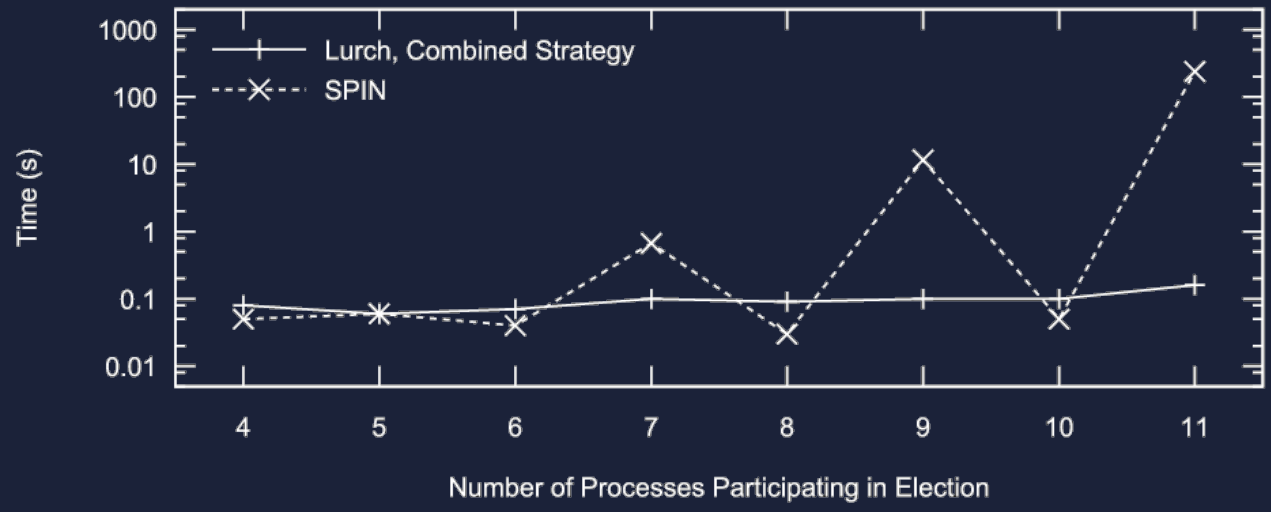
No indication NuSMV or SPIN had been used incorrectly on these models

Performance Variations

- SPIN and Lurch on fault-seeded SCR specifications
  - Lurch more quickly detected errors in 34 of 38 fault-seeded versions
  - Lurch: 3.74 s; SPIN: 43.3 s (avg for 34)
  - Combined strategy: 46.5 s; SPIN: 82.4 s (avg for 38)
- NuSMV and Lurch on fault-seeded versions of (large) RSML model
  - Lurch more quickly detected errors in 42 of 44 fault-seeded versions
  - Lurch: 251 s; NuSMV: 7920 s (avg for 42)
  - Combined strategy: 1100 s; NuSMV: 8200 s (avg for 44)

Performance Variations (2)

- Two (slightly) different versions of the dining philosophers problem
  - SPIN finds deadlock much faster in the normal version
  - NuSMV finds deadlock much faster in the *no-loop* version
- Scalable multi-process leader election protocol
  - Model seeded with two faults
  - SPIN very fast at detecting property violation on instances with an even number of processes, but very slow on instances with an odd number
  - Lurch fast (but incomplete) on all instances



Overview (5)

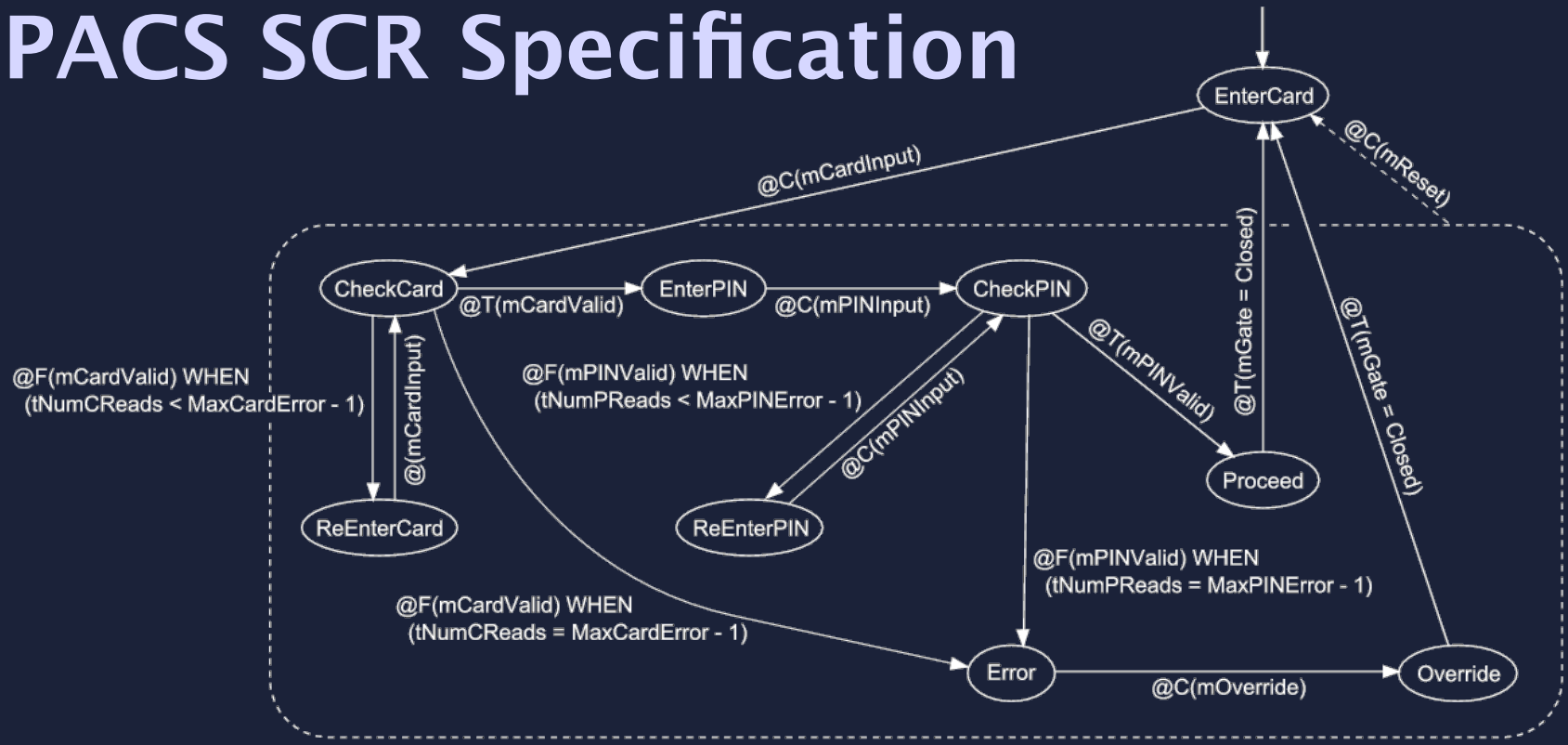
- Three Key Results
- Introduction
- Related Work
- Motivating Examples
- Case Study
  - Verification Tools
  - PACS SCR Specification
  - Generating Fault-Seeded Specifications
  - Experimental Results
  - Comparing Subsets of Specifications
  - Proposed Combination Strategy
- Conclusion

Verification Tools

- SCR Toolset Consistency Checker
  - Can check syntax, generic properties
- Salsa Invariant Checker
  - Can prove user-specified and generic properties, but unproven properties may or may not be true
- Cadence SMV and NuSMV Symbolic Model Checkers
  - Can detect property violations, but only for single-state assertions
- SPIN Explicit-State Model Checker
  - Can detect violations of single and two-state assertions, but requires most time and memory
- Lurch Random Search Tool
  - Can detect violations of single and two-state assertions, but not complete
  - Translator from SPIN (Promela), produced by SCR Toolset, to Lurch

Automatic translators in SCR Toolset

PACS SCR Specification



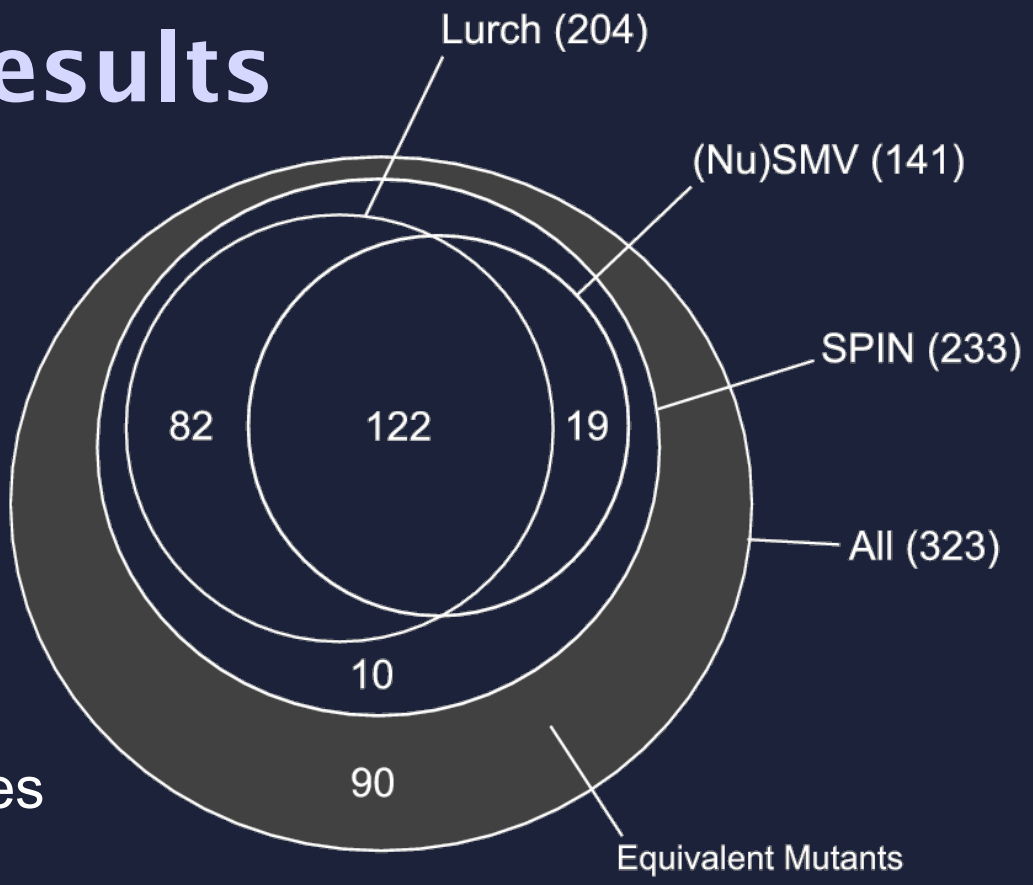
- Based on prose requirements from others' work comparing high process maturity vs. formal methods for effectiveness in producing reliable software
- SCR specification derived from requirements document as an example of a high-quality formal requirements specification
- Checks user card and PIN number, grants access to a restricted area

Generating Fault-Seeded Specifications

- 10 mutation operators chosen based on Offutt et.al. and Andrews et.al.
  - 3 taken directly from Offutt's set of 5 (judged sufficient for Fortran programs)
  - 3 more based on remaining 2 from Offutt
  - 4 operators designed to be more SCR-specific
- 323 fault-seeded specifications used in experiments
  - 229 with one mutation, 94 with two mutations
  - Included 45 generated manually for preliminary experiments (tools' performance on these very similar to performance on those generated automatically)
  - 90 found to be equivalent mutants

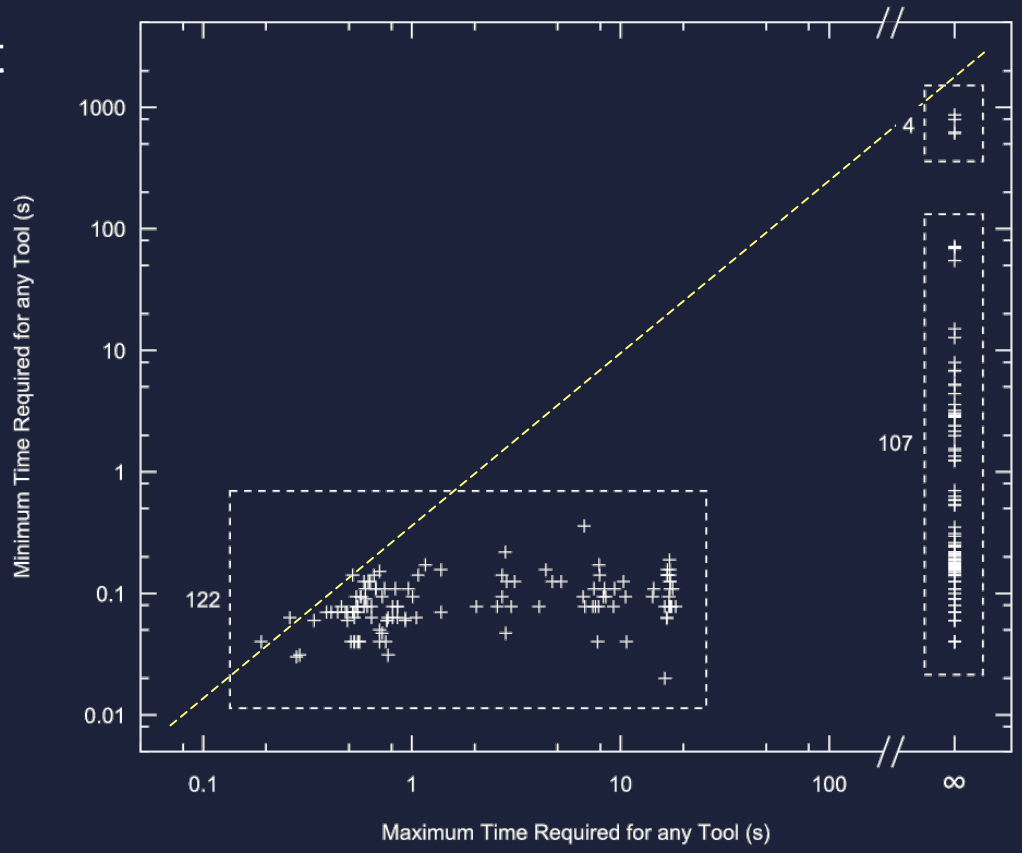
Experimental Results

- For 122 specifications in which Lurch, SMV and SPIN detected property violations, running SMV saves 6 minutes.
- For 82 specifications in which Lurch and SPIN detected property violations, running Lurch saves 41 minutes.
- For 19 specifications in which SMV and SPIN detected property violations, running SMV saves 201 minutes (3.5 hours).



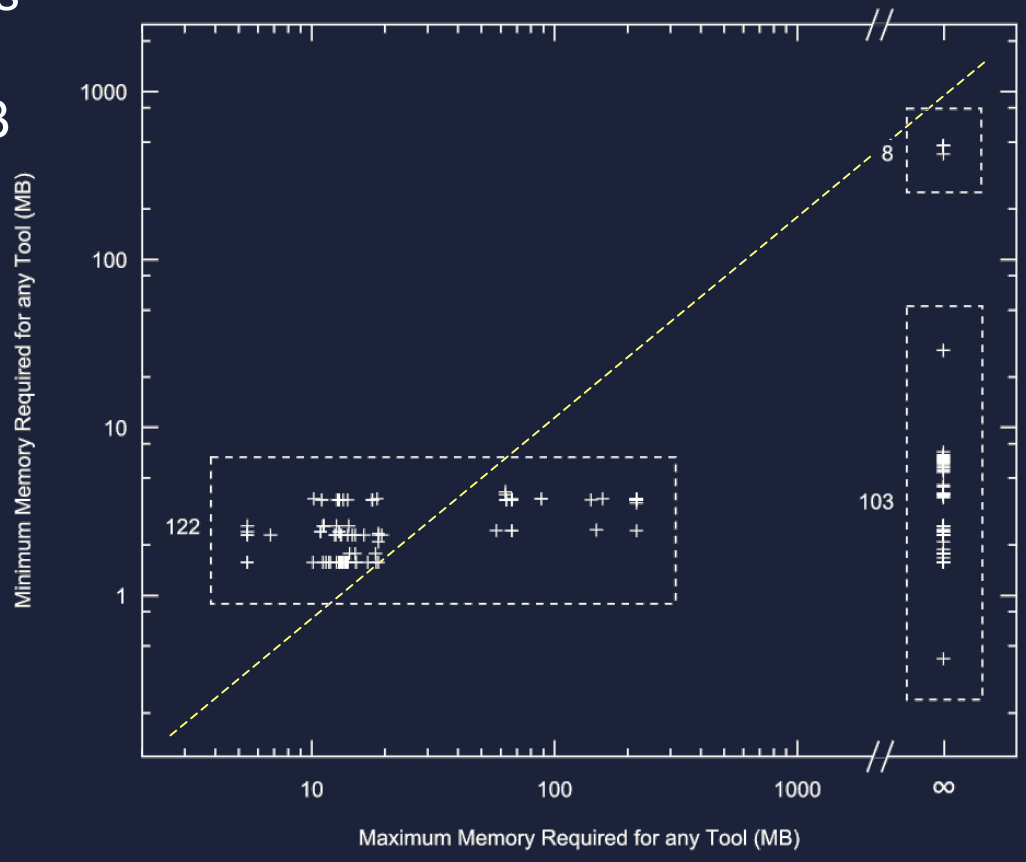
Experimental Results (2)

- Complementary performance (time requirements)
  - For 122 specifications with property violations detected by all tools, fastest required less than 1 s and slowest required less than 20 s
  - For 107 specifications impossible for 1 or more tools, fastest tool required less than 100 s
  - Only 4 specifications required over 500 s for best tool



Experimental Results (3)

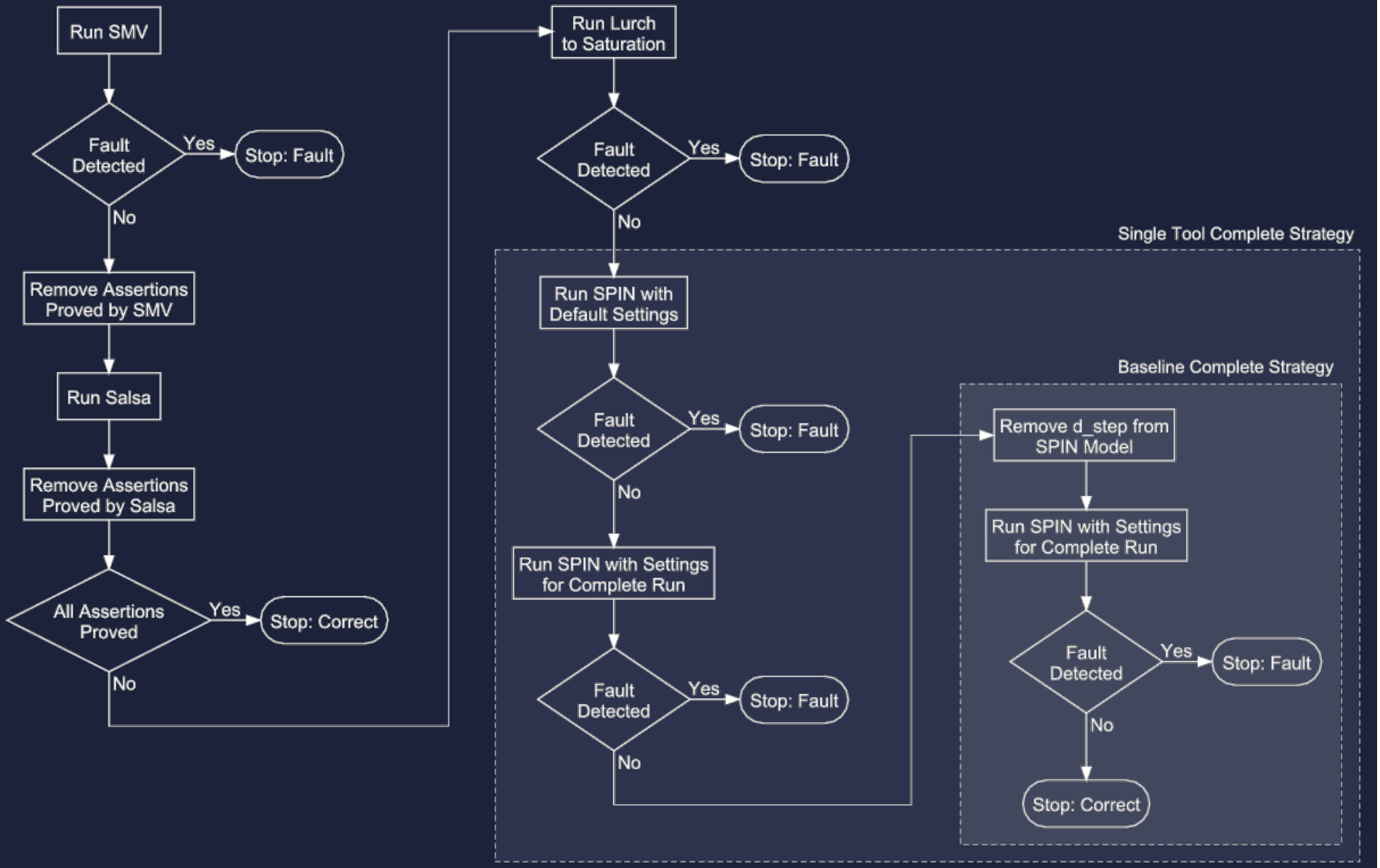
- Complementary performance (memory requirements)
  - For 122 specifications with property violations detected by all tools, best required less than 5 MB and worst required less than 125 MB
  - For 103 specifications impossible for 1 or more tools, best tool required less than 50 MB
  - Just 8 specifications required about 500 MB for best tool



Comparing Subsets of Specifications

- Subsets based on Salsa results
  - Specifications for which Salsa proved fewer user-specified assertions were easier for Lurch, much easier for SPIN
  - Specifications for which Salsa proved fewer generic properties were harder for Lurch
  - Specifications for which Salsa results matched results on the original were easier for Lurch, but harder for SPIN
- Subsets based on mutation operators
  - CRP (constant repl.) harder for all tools
  - EVR (enum type value repl.) + ROR (relational op. repl.) harder for all tools
  - For each tool, a different operator (or pair of operators) was most difficult
  - Two-mutation specifications more difficult for NuSMV but easier for SPIN and Lurch

Proposed Combination Strategy

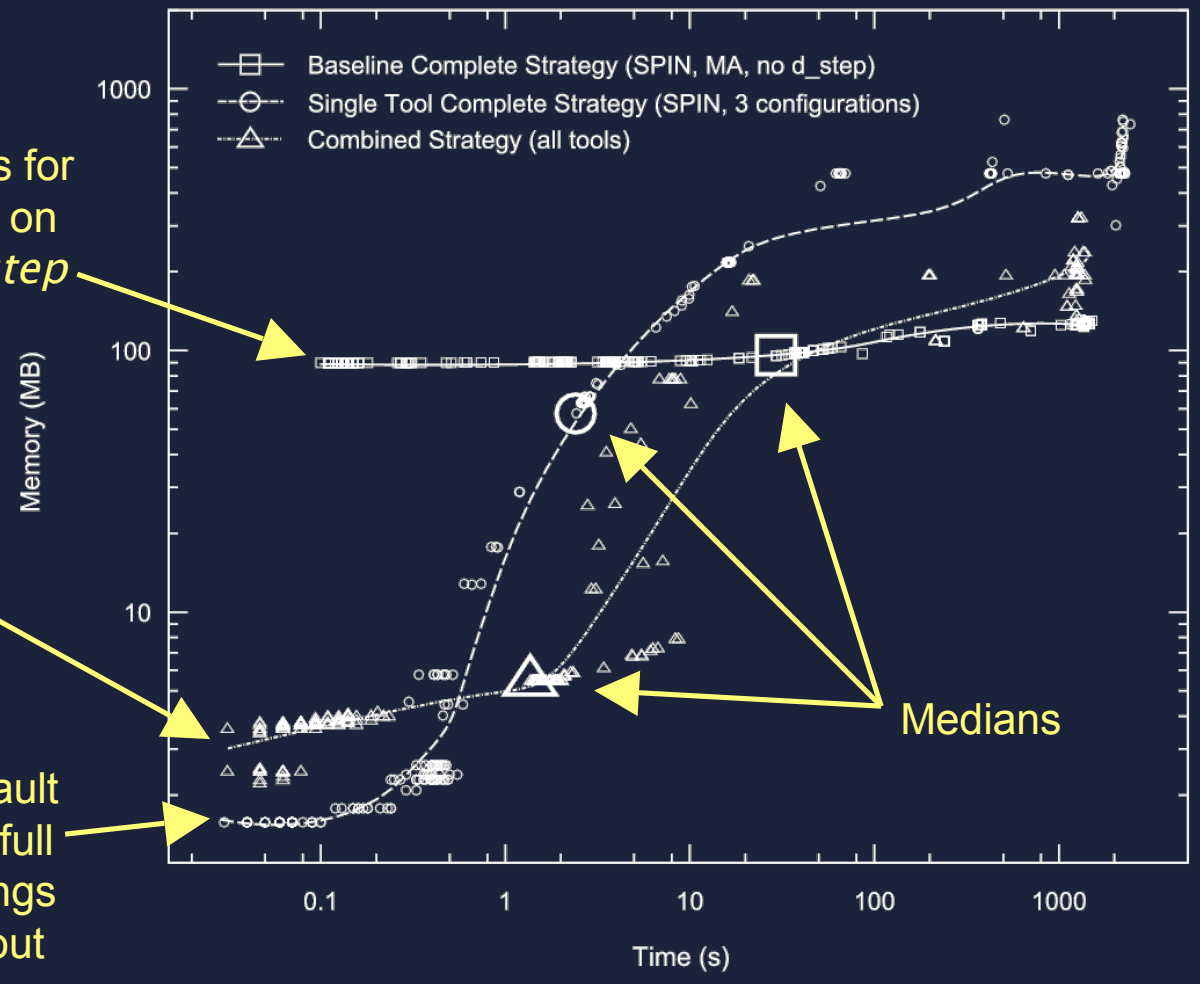


Proposed Combination Strategy (3)

SPIN, with settings for full verification run on model without *d_step*

Combined Strategy (SMV, Salsa, Lurch, SPIN in 3 modes)

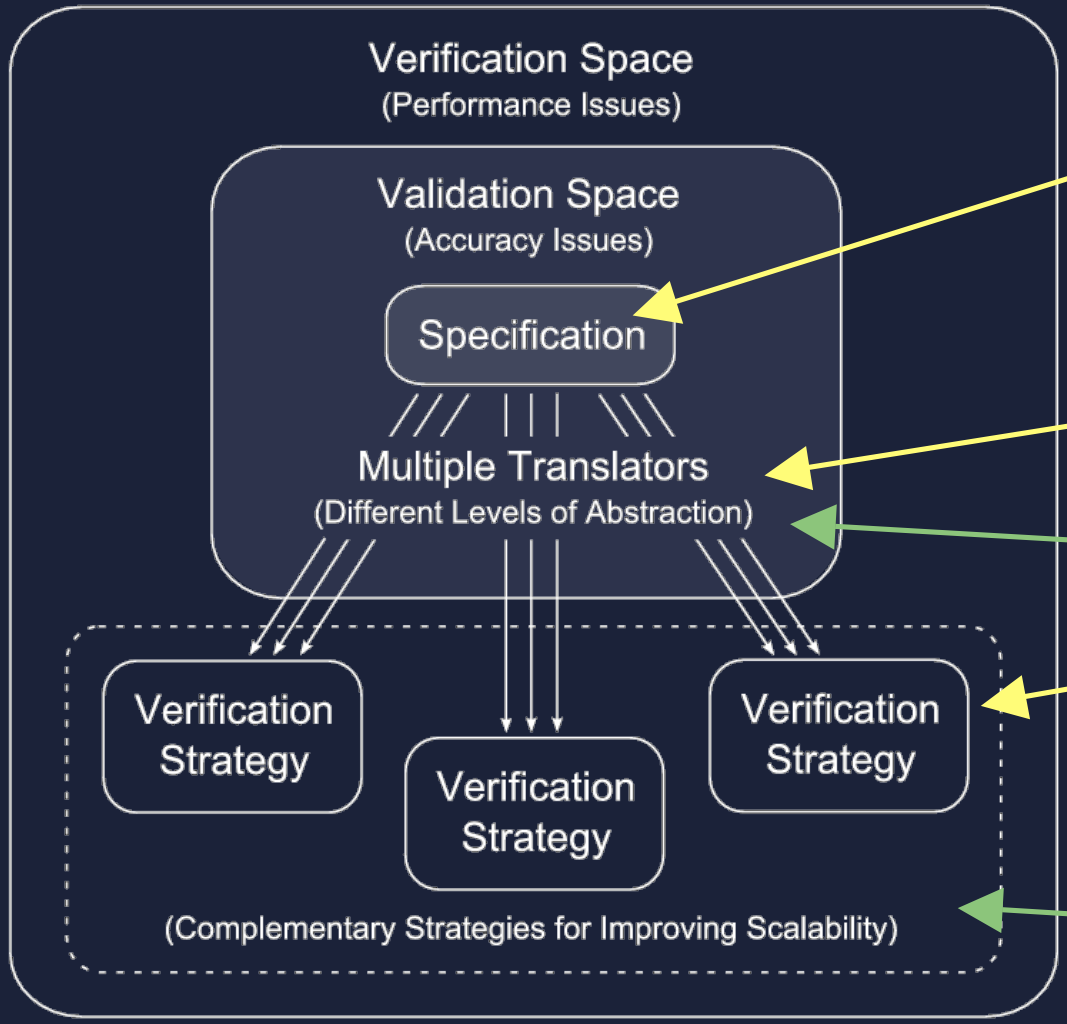
SPIN, 3 Modes: with default settings, with settings for full verification run, with settings for full run on model without *d_step*



Overview (6)

- Three Key Results
- Introduction
- Related Work
- Motivating Examples
- Case Study
- Conclusion
  - Conceptual Model of Verification Challenges
  - Open Research Questions
  - Summary

Conceptual Model of Verification Challenges



Start with a specification (i.e., some software artifact and definition of correctness)

Information from specification moves through a validation space via translation

Accuracy Issues

In verification space, check that part of specification representing behavior is consistent with part representing correctness

Performance Issues

Open Research Questions

- If verification tools cannot provide 100% confidence in their results...
  - Because of hidden assumptions
  - Incompleteness
  - Accuracy issues in translation
- What level of confidence can verification tools provide?
  - How can we quantify and compare the level of confidence provided by individual verification methods?
  - How can we measure confidence for a combination of methods?
- Would conclusions from our case study be confirmed by additional similar experiments?
  - On other SCR models, within the framework of the SCR Toolset?
  - On other software artifacts, in other verification frameworks?
  - To what degree can our assessment of individual tools be generalized to verification of other models, in other frameworks, etc.?

Open Research Questions (2)

- *Why* do the verification strategies we considered work the way they do?
  - Are there measurable attributes of input models that could be used to predict the performance of the different verification strategies?
  - Can we learn from these kinds of experiments how verification algorithms could be modified to improve their effectiveness?
    - If tools' performance is very sensitive to minor changes in the input model, might it also be very sensitive to minor changes in the verification algorithm?
- What is the best role for incomplete random search in automated verification?
  - Is there any way to measure how much confidence of correctness is provided by a random search run in which no property violation is detected?
  - Can saturation tell us anything about the size and structure of the unexplored portion of the model?
  - Could the performance of random search be used to predict the performance of other verification strategies?

Summary

- Software verification offers significant benefits but with significant costs
  - Validation cost
    - Expertise in modeling languages and system to be verified, domain knowledge
  - Verification cost
    - Expertise in verification methods, computational resources
- Strategies for decreasing these costs can be combined to create a strategy that is more accurate and more efficient
  - *Choose the right tool at the right time?* No, *Use all the tools all the time*
  - Multiple translation strategies give insight into accuracy issues, facilitate validation
  - Multiple verification strategies check each other's results, can be cascaded to improve performance