Extending Murφ to Analyze Classes of Errors

Presented by Sascha Konrad and Jesse Sowell

Outline

- Overview
  - Motivation
  - Approach
  - Structure
  - Implementation
  - Examples and Analyzer
    Preliminary Conclusions
  - References

Motivation

- Murφ is a rule-based model checker that employs an exhaustive search algorithm
- Some verifiers support the capability of providing more than one error trace of a protocol specification in one verification run
- Goals:
  - Extend Murφ to provide all invariant violation traces
  - Develop an application that analyzes the invariant violation traces
- Analyzing invariant violation traces:
  - Allows analysis of traces for commonality
    Provides more information about flaws
  - Accelerates the review and modification cycle
Approach

- Modify the Murφ verifier to deliver all error traces
- Save traces to an external file
- Parse the output and build the “deadtree”
- Analyze “deadtree” for invariant trace commonalities

Structure of the Murφ Analyzer

Murφ Breadth-first Search Algorithm

Initial State

Generate all start states
StartState->AllStartStates()

Generates all next states and adds them to the queue
/curstate = queue->dequeue() /Rules->AllNextStates()

Check deadlock
[deadlock==1]

Exit and print error
[deadlock==0]

Deadtree file
Deadtree analyzer
Deadtree commonalities

Special purpose verifier
Your favorite C++ compiler

Murphi specification file
Murphi compiler

C++ specification file
C++ analyzer files provided by Murphi
Add Method

Implementation of changes to the Murϕ Verifier

- No changes to the Murϕ compiler
- Changes applied to the Murϕ special purpose verifier include files
- Implementation goals:
  - Keep changes to the system as small as possible
  - Generate a complete invariant violation trace for each invariant violation
  - Generate output that can be processed by "deadtree" analyzer

Sequence diagram before modification
Examples and Analyzer

- Examples
  - E4Test
  - Lin-modified

- Deadtree Analyzer
  - Common ancestor
  - Analysis techniques
E4Test

- Very simple specification
- Designed to test:
  - Error trace generated by modified enumeration tool
  - Merging of error traces into “dead tree”
  - Testing of common ancestor analysis
- What is the E4Test?
  - Tree of boolean values
    - Each value dictates a “path choice” in the tree
  - Nondeterministic
  - So what does it look like?

Common Ancestors

- Common ancestor
  - Each leaf is ultimately a descendant of the root
  - There may be a non-root ancestor shared by all leaves
- What does it tell us?
  - Common ancestor tells us who can generate an invariant violation
  - All descendants of the common ancestor will ultimately cause an invariant violation
  - State may contain information that leads to an invariant violation
Lin-Modified

- Counting example
- Starts at 1
- Counts up by either 3, 5, or 7
- Easy example of common ancestor

Analyzer and Classes of Errors

- Common ancestor is the root of a subtree
  - All nodes of this subtree contribute to one or more possible invariant violations of a given type
- May look for common ancestor of related invariant violations
- Currently, classes of invariants do not exist in the language
  - "Classes" are currently the invariant violations (leaves) grouped into a leafset
Preliminary Conclusions and Further Work

- Common Ancestors
  - Common ancestors may show a localization of a set of invariant violations.
  - Based on this localization, one may look for initial indications of the invariant violation in:
    - The path to the common ancestor
    - Descendants of the common ancestor
  - Indications may lead to the creation of more general invariants that may catch the violation sooner

- Path differences (further work)
  - Can we do commonality analysis on the paths themselves?
  - Can we analyze the elements of the state that contribute to the invariant violation in each node of the path?

References


