CSEP590 – Model Checking and Automated Verification

Lecture outline for August 6, 2003
First, we’ll discuss Abstraction Techniques from the last lecture…

Today, we’ll be primarily concerned with discussing the model checker SPIN, and if time permits, the very interesting Bandera system for Java

- **SPIN**
  - Developed by G.J. Holzmann at Bell Labs
  - It is the topic of an annual workshop since 1995
  - Designed for the simulation and verification of distributed algorithms, focusing on asynchronous control in software systems
    - Different than other approaches (synchronized hardware systems)
  - Systems are described in the Promela modeling language
    - Allows for describing each process in the system + the interactions between processes
  - Communication: processes use FIFO comm. Channels, shared variables, rendez-vous comm. (see manual for this)
-The models are bounded and have countably many distinct behaviors
  -=> correctness properties are formally decidable, subject to constraints of computational resources (time, memory)
-SPIN seeks to address some of these constraints, how do you deal with them?
-We’ll see a diagram in class of the basic SPIN architecture
-Why is compilation used?
  -Allows for generation of highly optimized models, specific to property
-SPIN framework
-Models specified in Promela
  -Process templates + instantiation of processes
  -Templates define process behavior
-Templates translated into a finite automaton
  -Global behavior of system created by computing an asynchronous interleaving product of automata, 1 automata per process behavior
-Global system: represented by automaton (state space of system, or reachability graph of system)
-\(LTL\) formula specs translated in a Buchi automaton
-Computes asynchronous product of Buchi automaton and global automaton to get another Buchi automaton, \(B\)
-If language accepted by \(B\) is empty \(\rightarrow\) original spec claim is not satisfied for given system. Nonempty \(\rightarrow\) \(B\) contains all behaviors that satisfy spec
-However, size of global reachability graph can grow exponentially with \# of processes
-SPIN uses complexity management techniques to solve/help

-What are Buchi automata? A quick defn…
-Variant of an NFA for processing words of infinite length
-Accepts a string iff execution of automaton goes thru an accepting state infinite \# of times while processing the string
-Automata generated formally accept only those infinite system executions that satisfy the corresponding \(LTL\) formula
-SPIN uses a nested depth-first search technique to look for these acceptance cycles in the automata
-LTL formula’s are specified in Promela according to the following grammar:

\[ f = p \mid \text{true} \mid \text{false} \mid (f) \mid f \text{ binop } f \mid \text{unop } f \]

- \text{unop} = [] (always), <> (eventually), ! (negation)
- \text{binop} = U (until), && (and), || (or), -> (implies), <-> (equiv)

-Ex: [](pUq) means “always guaranteed that p is true until at least q is true”

-Partial Order Reduction
-SPIN uses this method to reduce the # of reachable states that must be explored to complete a verification
-Reduction: based on the fact that the validity of a LTL formula is often insensitive to the order in which concurrent and independently executed events are interleaved in the depth-first search
-Thus, we can generate a reduced state space with representative classes of execution sequences => collapse equivalent sequence orderings!

-What about memory management in SPIN?
  -The size of interleaving processes is worst case exponential in the # of processes
  -How fix?
    -1) state compression – new method added in 1995 allows for a 60-80% memory reduction in practice with only a 10-20% increase in CPU time
    -2) bit-state hashing – 2 bits of memory are used to store a reachable state. Bit addresses are computed using 2 statistically independent hash functions
       -Good for when exhaustive verification isn’t possible but you still want a good approximation
-Let’s discuss the Promela language syntax and semantics
  -Refer to the Promela language manual for this part of the lecture
-Now, we’ll see a SPIN demo, highlighting the main features of the software and showing a number of demo verifications
  -This should be enough to get you familiar with SPIN to be able to use it on the next problem set
-If time permits, we will discuss Bandera next
-Bandera
  -Seeks to bridge the gap between research and practice in model checking software
  -Integrated collection of program analysis and transformation components enabling automatic extraction of safe, compact finite-state models from program source code (Java)
  -From Java code $\Rightarrow$ SMV or SPIN model, then map verifier output back to the Java source code
  -Why is this a good idea?
- Alleviates the need to construct models by hand
- Has optimizations to deal with state space explosion problem automatically
- Bandera philosophy
  - 1) reuse existing checking technologies
  - 2) automated abstractions
  - 3) customize model based on spec/property
  - 4) open design for ease of extensibility
  - 5) integration with existing software testing/debugging techniques
- What techniques does Bandera use to build tractable models from software?
  - 1) irrelevant component elimination
    - Many program components (classes, threads, vars, code) might be irrelevant to the property being verified
    - Ex: clicking on a menu brings up a certain dialog box is independent of application code
-2) Data abstraction
   - Vars may record more detail than necessary for property being tested
   - Ex: items in a vector, but if property is only concerned with the existence of a certain item in the vector, then the # of vector states can be abstracted to just 2: {ItemInVec, ItemNotInVec}

-3) Component Restriction
   - If 1) and 2) fail, create a restricted model
     - Limit # of components, limit range of vars
   - Idea: many design errors are manifest in small versions of a system => can still be useful for find errors in the actual system
   - How does Bandera do it though?
     - Uses slicing to automate irrelevant component elimination
     - Abstract interpretation module for data abstraction
     - Model generator with built-in flexibility
- Data structures for mapping between model checker error traces and the original source code + a graphical tool for navigating these traces
- Includes a menu-driven library for helping the user to create logic specifications
- Irrelevant components are sliced away from the program
- Data abstractions are applied on the remaining model
- The back-end generates a SPIN or SMV model
- The translator maps back from the verifier to the source code

- What’s a slicer?
  - Given a program P and program statements C = \{s_1 \ldots s_k\} of interest from P called the slicing criterion, the slicer computes a reduced version of P by removing statements of P that do not affect computation of the criterion statements C
  - Bandera slices to remove statements that do not affect the satisfaction of the given property $\phi$
-Recent work has shown that slicing criterion can be based only on the primitive properties in $\phi$
- Slicers are hard to build, especially for Java’s concurrency model!
- Bandera’s Abstraction-Based Specializer (BABS)
  - Automates the model reduction via data abstraction
  - Useful when the specification depends only on the properties of data values, NOT the actual concrete data values themselves
- User can guide abstractions as well with built in libraries and a user input mode