Verification of programs in practice

December 15, 2005

– Verification of Java programs (Pathfinder, ESC/Java, Bandera)
– Proof Carrying Code
– Combinations with static analysis

Java PathFinder (NASA Ames)

early effort to verify code written in usual programming languages
Java PathFinder 1.0: translation from Java to PROMELA (Spin)
– language similarities: treatment of dynamic object creation, threads
– missing aspects (floating point); Spin needs complete model/source

Java PathFinder 2.0: standalone verifier, written in Java
[G. Brat, K. Havelund, S. Park, W. Visser '00]
General architecture:
explicit state model checking
– usual technique for representing large states (structures):
each structural value stored only once and encoded as integer
custom Java virtual machine for model checking, with
– exploration algorithms (issue forward/backward steps in own JVM)
– nondeterministic environment using special methods captured by JVM

Java PathFinder (cont.)

Verification techniques:
Static analysis
– for constructing program abstractions (by *slicing*)
– for identifying partial order reduction conditions
using the SVC (Stanford Validity Checker) theorem prover

Runtime analysis, for detecting potential error conditions:
– race conditions in access to shared variables
– accessing semaphores in different order (potential deadlock)

Performance and verified systems
– coded in Java, 10x slower than Spin; speed: thousands of states/sec.
– verified a control agent for state space operation
– a fragment of a distributed operating system (14 classes, 1 kloc)

Now a SourceForge project: `http://javapathfinder.sourceforge.net`

Abstraction in Java PathFinder

Source-level transformations generate another Java program that operates on abstract predicates.
Abstractions is expressed as special annotation `class Abstract;`

```
Abstract.remove(x) // abstracts away x
Abstract.addBoolean("x0", x == 0) // adds predicate x == 0
```

One can also abstract away predicates over several classes:
`Abstract.addBoolean("xGTy", A.x > B.y);`
– generates a predicate for each object pair instantiated from classes A and B
– possible explosion in the number of needed predicates

ESC/Java (DEC/Compaq SRC)

ESC = Extended Static Checking; initially for Modula 3, then Java

– not a model checker, but a static analyzer
– can detect errors such as null references, out-of-bound indices
– for more complex properties *annotations* are used
(invariants, preconditions, postconditions, null/non-null conditions)
– allows modular verification, separately for each method
– verification done using a theorem prover (Simplify)
– modules with unavailable source supplanted by specification files

Similar static analyzers exist for C (`lint`, evolved into `splint`)

Bandera [Kansas State U.]

A modular verifier for Java programs
– o front-end for program simplification (program slicing)
– a library of frequently-used abstractions
(the user specifies for each variable the desired abstraction)
– ability to restrict the model to a small number of module instances
– a generator for a finite model in a generic format
(guarded command language, easily translated to other specifications)
– interfaces with usual verifiers (SMV, Spin, PVS theorem prover)
– a specification language, and support for formulas based on patterns

Proof Carrying Code

[Necula & Lee '96, Necula '97]
– A method for safe execution of untrusted code, e.g. net applets
– code consumer defines a set of safety rules
– code producer delivers the code coupled with a formal proof that it satisfies the safety rules
– consumer uses a simple proof checker to establish validity of code and received proof

Key idea: checking a proof is a simple mechanical task (simple checker; small, verifiable trusted code base) while generating a proof is hard (burden falls on code producer)

PCC: Structure of system

Producer
– certifying compiler, generates native code with annotations (e.g. invariants)
– verification condition generator (VCGen)
VC = predicate whose validity guarantees safe execution
– proof generator (starting from VCGen)

Consumer
– a proof checker: validates correspondence between received code and proof
(possibly assisted by a VCGen identical to producer's)

PCC: verification condition generation

– The notion of VC: linked to the rules for program correctness based on preconditions and postconditions established by Floyd (1967)
– a VC is not necessarily the weakest precondition
(can be simpler, easier to express and prove)

```
for (i = 0; i < length(a); i++) s += a[i];
```

VC built from predicates of the form

$s : int \wedge l \geq 0$ premise

$i \geq l \rightarrow s : int$ postcondition

$i < l \rightarrow \mathit{saferd}(mem, a + i) \wedge i + 1 \leq l \wedge s + \mathit{rd}(mem, a + i) : int$ invariant

PCC: advantages

– Validating a proof is much easier than finding it
⇒ simple for the consumer, who need only trust proof checker
– Method is tamper-proof: no change in code and/or proof can go undetected (if a changed program checks, it is still safe)
– Verification is performed once, statically; allows subsequent safe execution without inserting run-time checks
– Allows to trust the compiler completely and even to debug it during the development process.

CCured: strongly typed C code

[Necula et al '01]: C programe correct from typing point of view
– combination of type inference and runtime checking
– type inference used to establish as much as possible of the code as being type-safe
– run-time checks are inserted in the rest of the source to ensure correctness of memory access
– basic idea: extending the type system with pointers qualified as `SAFE` (just dereferenced), `SEQ` (for arrays) and `DYNAMIC` (any access)
– additional fields (base and length) are introduced for pointers which are not `SAFE`
– slowdown factor: 1 - 2 (compared to 10-100 for Purify)
– analysis also allows detection of errors

Extensions for metacompilation

Context: *lightweight and semi-formal methods*: sacrifice part of soundness/completeness guarantees to increase practical applicability

Metacompilation [Engler et al '00]: for bugs in operating system code
– allows to check well-defined high-level (API) rules, (e.g., "variable `x` is protected by semaphore `s`", "interrupts must be reenabled")
⇒ by defining a meta-semantics accessible to the compiler
– rules are specified as automata which transition while analyzing a relevant pattern of source code
– an augmented C compilet applies these extensions to semantic analysis (propagation through the control flow graph, locally or globally)
– results: hundreds of errors in Linux, OpenBSD, Exokernel, etc.

Related approach: automatic extraction of models from source code
– by slicing (also for operating systems)
– models are then analyzed by a model checker