Verification of Lock-Free Algorithms with RGITL

Gerhard Schellhorn

joint work with

Bogdan Tofan

Institute for Software and Systems Engineering
University of Augsburg

15.04.2011

Universität Augsburg

Outline

1. RGITL in KIV
   - Background: Interactive Theorem Proving with KIV
   - Interval Temporal Logic (ITL) and Programs
   - Proof Principles
   - Rely-Guarantee (RG)
   - Decomposition Theorems
2. Verification of Hazard Pointers
   - Lock-Free Algorithms
   - Hazard Pointers
   - Proof Sketch
3. Summary and Open Issues

Outline

1 RGITL in KIV
- Background: Interactive Theorem Proving with KIV
- Interval Temporal Logic (ITL) and Programs
- Proof Principles
- Rely-Guarantee (RG)
- Decomposition Theorems

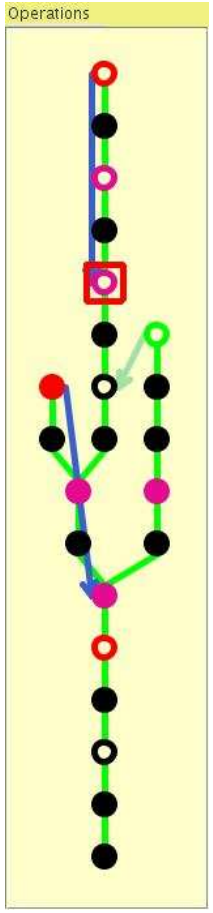
2 Verification of Hazard Pointers
- Lock-Free Algorithms
- Hazard Pointers
- Proof Sketch

3 Summary and Open Issues

Background: Proving Sequential Programs with KIV

KIV is an interactive theorem prover based on

- Structured algebraic specification of data types with higher-order logic
- Sequent calculus with proof trees
- wp-calculus for ASMs and Java
- Proof principle for sequential programs: symbolic execution (+ induction) [Burstall 74] (= incremental computation of strongest postconditions for instructions)
- Semantics (for a fixed algebra): $s \models \varphi$ with state s = valuation of variables



Motivation

Define a logic where:

- Arbitrary temporal properties can be proved
- Native support for interleaved programs exists (no encoding to transition systems)

Define a calculus where:

- Proving contracts for sequential programs should not be more difficult than before
- Since interleaving generates many cases: compositional reasoning (rely-guarantee) is supported

Temporal Logic in KIV: Intervals

- KIV implements a variant of Interval Temporal Logic [Moszkowski]
- Semantics based on Intervals I = sequence of states (I(0), I′(0), I(1), I′(1), . . .)
- I has finite (termination!) or infinite length
- I alternates system steps (I(0),I′(0)), (I(1),I′(1)), . . .
  with environment steps (I′(0),I(1)), (I′(1),I(2)), . . .
  (similar to reactive sequences [deRoever 01])
- Semantics of higher-order-formula φ uses initial state I(0) only: $I \models \varphi \Leftrightarrow I(0) \models \varphi$

Temporal Logic in KIV: Operators

- Variables can be static or flexible
- X, X′, X″ denotes the value of flexible X in I(0), I′(0), I(1)
- predicate p(X,X′) describes a system step (guarantee)
- predicate q(X′,X″) describes an environment step (rely)
- Standard TL operators:
  ◇, □, **until**, **A** (for all paths),
  ○ (strong next state), • (weak next state), **last** (termination)

Programs in Temporal Logic

- Programs α are formulas too:
  $I \models \alpha \Leftrightarrow$ the system steps in I are possible steps of α
- $I \models [X := T]_{X,Y}$
  $\Leftrightarrow I = (I(0), I'(0), I(1)) \wedge I'(0)(X) = I(0)(T) \wedge I'(0)(Y) = I(0)(Y)$
- Frame assumption $[\ldots]_{X,Y}$ as in TLA [Lamport 94],
  but no built-in stuttering
- Programs: sequential (**let**, **while**, rec. procedures) +
  $\alpha \,||\, \beta$ (interleaving), **await** C (block until C holds)
- Typical goal: $\alpha \wedge E \rightarrow P$
  "Executing α in environment E satisfies P"
- Environment assumption $\Box\, X' = X''$:
  the environment never changes variable X.

Proof principle 1: Symbolic Execution

- Symbolic execution = Step forwards through an interval
- Advantage: no encoding of programs as transition systems with program counters (as in Step, TLA or Model checking) $\Rightarrow$ readable goals
- Symbolic execution is done in two phases:
  1. Unwind TL ops. and programs; split first step from the rest:

$$\Box\,\varphi \equiv \varphi \wedge \bullet\,\Box\,\varphi$$

$$[X := T; \alpha]_{X,\underline{Y}} \equiv X' = T \wedge \underline{Y'} = \underline{Y} \wedge \circ\,[\alpha]_{X,\underline{Y}}$$

  2. Remove first step of interval (new static vars x_0, x_1):

$$\frac{p(x_0, x_1, X) \wedge \varphi}{p(X, X', X'') \wedge \circ\,\varphi}\ \mathit{step} \qquad \frac{p(x_0, x_0, x_0)}{p(X, X', X'') \wedge \mathbf{last}}\ \mathit{last}$$

Universität Augsburg

Proof principle 2: Compositional reasoning

- Substitution principle:

$$\frac{\alpha \to A \qquad \beta \to B \qquad A \,||\, B \to C}{\alpha \,||\, \beta \to C}\ \textit{abstract}$$

- possible, since interleaving (like other operators) has compositional semantics
- Program α satisfies guarantee G as long as rely R holds:
  $\alpha \;\to\; R(X', X'') \xrightarrow{+} G(X, X')$
  where $R \xrightarrow{+} G \equiv \neg\,(R\ \textbf{until}\ (\neg\, G))$
- Rely-Guarantee rules can be proved as theorems.

Rely-Guarantee Theorem

Theorem

(1) $pre \wedge COp_1 \rightarrow R_1 \xrightarrow{+} (G_1 \wedge (\textbf{last} \rightarrow post_1))$

(2) $pre \wedge COp_2 \rightarrow R_2 \xrightarrow{+} (G_2 \wedge (\textbf{last} \rightarrow post_2))$

(3) $G_1 \vee R \rightarrow R_2, G_2 \vee R \rightarrow R_1, G_1 \vee G_2 \rightarrow G$

(4) $reflexive(G_1, G_2), transitive(R_1, R_2)$

(5) $pre \wedge (R_1 \vee R_2) \rightarrow pre$

then $pre \wedge COp_1 || COp_2 \rightarrow R \xrightarrow{+} (G \wedge (\textbf{last} \rightarrow post_1 \wedge post_2))$

- similar to [Xu,deRoever 97] (except cond. (5))
- Their notation for (1): COp_1 $\underline{sat}$ $(pre, rely_1, guar_1, post_1)$

Decomposition of Linearizability

Theorem (Bäumler et al. 09)

If for all $1 \leq p, q \leq n$, $p \neq q$:

(1) $COp_p \rightarrow R_p \xrightarrow{+} G_p$

(2) $G_p \rightarrow R_q$, $reflexive(G_p)$, $transitive(R_p)$, $R \rightarrow R_p$

(3) $COp_p(CS) \wedge \Box\,(R_p \wedge Abs(CS) = AS \wedge Abs(CS') = AS') \rightarrow skip^*; AOp_p(AS); skip^*$

then $COp_1^* \| \ldots \| COp_n^* \wedge \Box R \rightarrow AOp_1^* \| \ldots \| AOp_n^* \| skip^*$

- COp_p is a concrete algorithm (procedure) that implements an atomic operation AOp_p
- R is the global environment assumption
- Linearizability expressed as special case of refinement
- Most linearizable algorithms allow reduction to two representative processes $\Rightarrow$ reduction proved

Decomposition of Lock-Freedom

Theorem (Tofan et al. 10)

If for all $0 \leq p, q, p \neq q$*:*

(1) $COp_p \rightarrow R_p \xrightarrow{+} G_p$

(2) $G_p \rightarrow R_q$, $reflexive(G_p), transitive(R_p)$, $R \rightarrow R_p$

(3) $reflexive(U), transitive(U)$, $R \rightarrow R_p \wedge U$

(4) $COP_p(CS) \wedge \Box R_p$
$\rightarrow \Box (\neg U(CS, CS') \vee (\Box U(CS', CS'')) \rightarrow \Diamond$ **last**)

then $COP_0^* \| \ldots \| COP_n^* \wedge \Box R \rightarrow \Box$ *progress*

where progress = "some operation active $\rightarrow$ *some operation terminates"*

- Predicate U ("unchanged") describes conditions under which $COP_p(CS)$ terminates in environment R_p.
- At any time, COP_p eventually terminates ($\Diamond$ **last**), if:
  - It updates the shared state itself $\neg U(CS, CS')$, or
  - It encounters no interference $\Box U(CS', CS'')$
- Theorem holds for weak fair and nonfair interleaving

Outline

1 RGITL in KIV
- Background: Interactive Theorem Proving with KIV
- Interval Temporal Logic (ITL) and Programs
- Proof Principles
- Rely-Guarantee (RG)
- Decomposition Theorems

2 Verification of Hazard Pointers
- Lock-Free Algorithms
- Hazard Pointers
- Proof Sketch

3 Summary and Open Issues

Lock-free Algorithms

- Do not use locks to protect global data structures from interference
- Instead: Try and retry principle with three steps
  - Make local copy old of pointer Global to data structure.
  - Prepare modification by computing new.
  - Compare and set (CAS) snapshot.
- If CAS returns false due to interference retry.

```
CAS(old, new; Global) {
    Atomic {
        if Global = old then {
            Global := new; return true }
        else return false } }
```

Hazard Pointers: Basic Idea

- Problem: Deallocation after removal from data structure leads to illegal concurrent access (and the ABA problem)
- Either rely on garbage collection, use modification counters or use hazard pointers
- Invented in [Michael 04] (similar: [Herlihy et al. 04])
- Every process X stores reference it wants to access in hazard pointer record: HPR[X] := r and validates this entry
- Idea: Validated entries are protected from deallocation
- To schedule r for deallocation, put in retired list $rlist_X$
- Scan removes all references in $rlist_X \setminus HPR$ using local vars. $i_X, lhp_X, lhplist_X$ to collect pointers from HPR

Verification Challenge

- Verification with hazard pointers is much more complex than assuming GC (various automated proofs for GC)
- [Michael04] gives temporal characterization, arguing about time points in the past
- [Parkinson, Bornat, O'Hearn 07] propose hazard pointers as a challenge for mechanized verification. They use ghost code to characterize unvalidated hazard pointers.
- Idea: Characterize validated hazard pointers:
  "As soon as reference r has been validated by X, no other process Y will deallocate it in its Scan-Operation".

$$
\begin{array}{l}
\mathit{Hazard}_X \wedge \mathit{Scan}_Y \wedge \mathit{ltop}_X \in (\mathit{rlist}_Y \setminus \mathit{lhplist}_Y) \rightarrow \\
\quad \textbf{if } \mathit{lhpUsed}_Y \textbf{ then } i_Y < X \vee (i_Y = X \wedge \mathit{lhp}_Y = r) \\
\quad \textbf{else } i_Y \leq X
\end{array}
$$

Outline

1 RGITL in KIV
- Background: Interactive Theorem Proving with KIV
- Interval Temporal Logic (ITL) and Programs
- Proof Principles
- Rely-Guarantee (RG)
- Decomposition Theorems

2 Verification of Hazard Pointers
- Lock-Free Algorithms
- Hazard Pointers
- Proof Sketch

3 Summary and Open Issues

Summary

- Interactive verification approach based on a variant of ITL with explicit programs and interleaving
- KIV implements TL-calculus based on symbolic execution
- Compositional semantics of interleaving allows to derive rely-guarantee rules, decomposition for safety and liveness properties
- Mechanized verification of several lin. algorithms, among them Michael&Scott's queue, Treiber's Stack with hazard pointers and modification counters
- for more algorithms and KIV proofs see
  http://www.informatik.uni-augsburg.de/swt/projects/lock-free.html

Open issues

- General verification principles for non-atomic refinements (“ modulo stuttering”) involving programs (not just transition systems!) is still an open issue
- Generalize linearizability theorem
- Two complete fragments: RG and pure ITL, but no completeness theorem
- Current automation: rewriting & heuristics
- Future work: Better automation
  - Shape Analysis (TVLA) to automate reasoning about heaps
  - Separation Logic to encode disjoint heaps (compatible with TL? with types?)
  - Computing invariants/relys as fixpoints