PROGRAMMING WITH TRIGGERS

Michał Moskal

instytut informatyki uniwersytet wrocławski
University of Wrocław
Wrocław, Poland

Microsoft Innovation Center Europe
Aachen, Germany

Satisfiability Modulo Theories Workshop
August 2nd, 2009, McGill University, Montreal, Canada

OUTLINE

- case-study: an SMT-powered software verifier applied to a commercial operating system
- tools and methods to make this work
  - trigger-engineering
  - tools:
    - Axiom Profiler – postmortem analysis of the search
    - Model Viewers – analysis of counter examples
    - Z3 Inspector – live view of Z3 operation

WINDOWS HYPERVISOR

- virtualization platform
  - thin layer of software between guest operating systems and the hardware
- essentially a small operating system
  - small by OS standards: 100kloc of C, 5kloc x64 asm
- scheduler, memory allocator, etc
  - lock-free data structures
- shipping with Windows Server 2008

Hypervisor Verification (2007 – 2010)

Partners:

- European Microsoft Innovation Center
- Microsoft Research Redmond
- Microsoft's Windows Div.
- Universität des Saarlandes
- German Research Center for Artificial Inteligence

co-funded by the German Ministry of Education and Research

http://www.verisoftxt.de

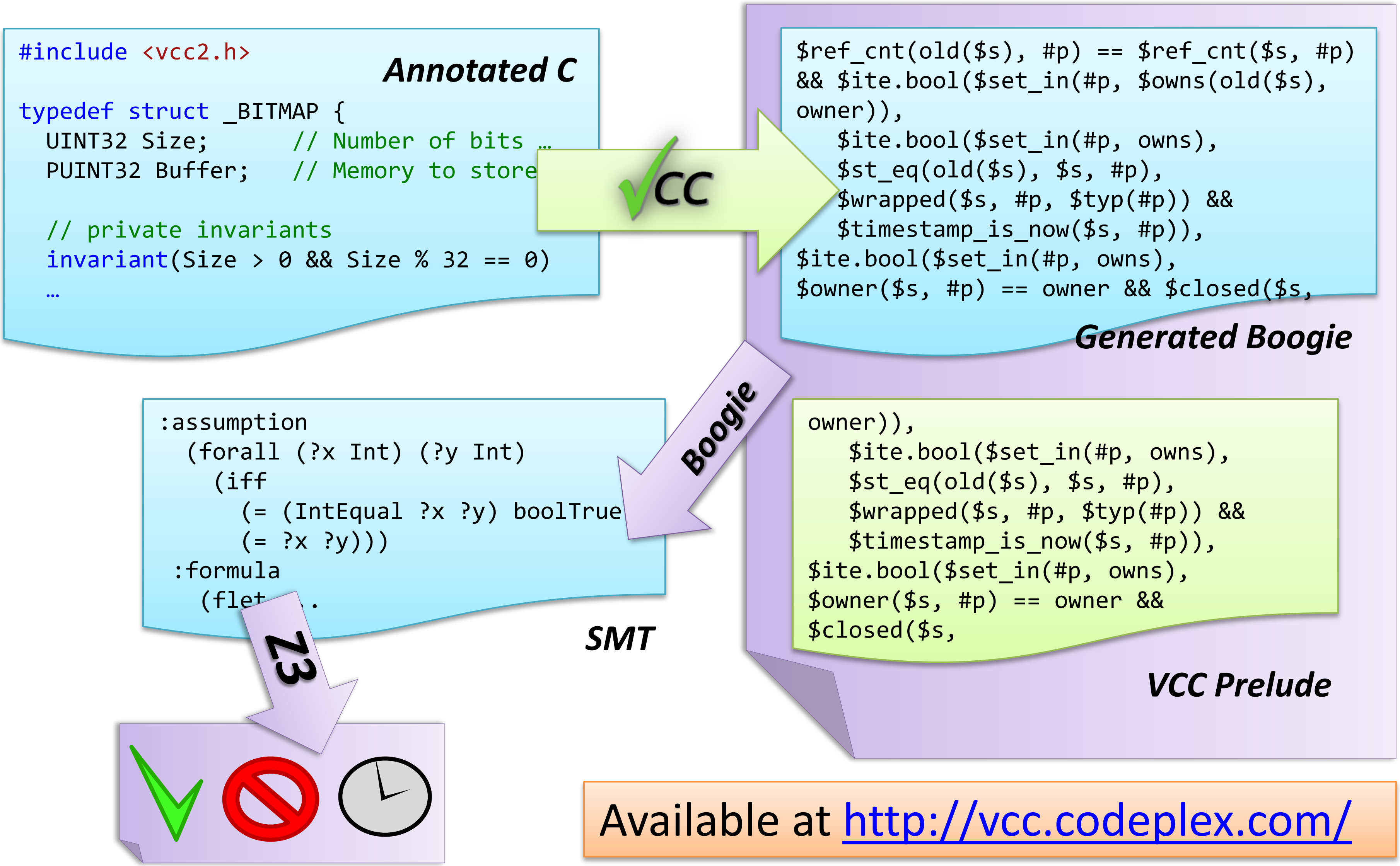
Goal: Full-blown Verification For Everyone

- functional properties
  - but even memory safety depends on functional correctness of complex data structures and concurrency protocols
- automatic
- exercised on real code
  - scalable – modular
  - concurrency
  - not changing existing code
- necessary tool support

VCC

- a deductive verifier for C
- verification methodology centered around
  - two-state invariants
  - ownership system
  - concurrency
- uses Boogie and Z3 (or other Boogie-supported provers)

VCC Architecture



Available at http://vcc.codeplex.com/

A Verification Methodology

- annotation language
  - e.g., first-order logic, higher-order logic, separation logic; + specific features
- specification concepts
  - ownership, type invariants, permissions
- modeling of the programming language semantics
  - how precise, what assumptions, etc.
- specification idioms

Verification Methodology As An SMT Theory

- complex
  - all input language + specification language constructions
- evolving with the verification tool
- not practical to implement as part of SMT solver
- instead encoded using first-order logic

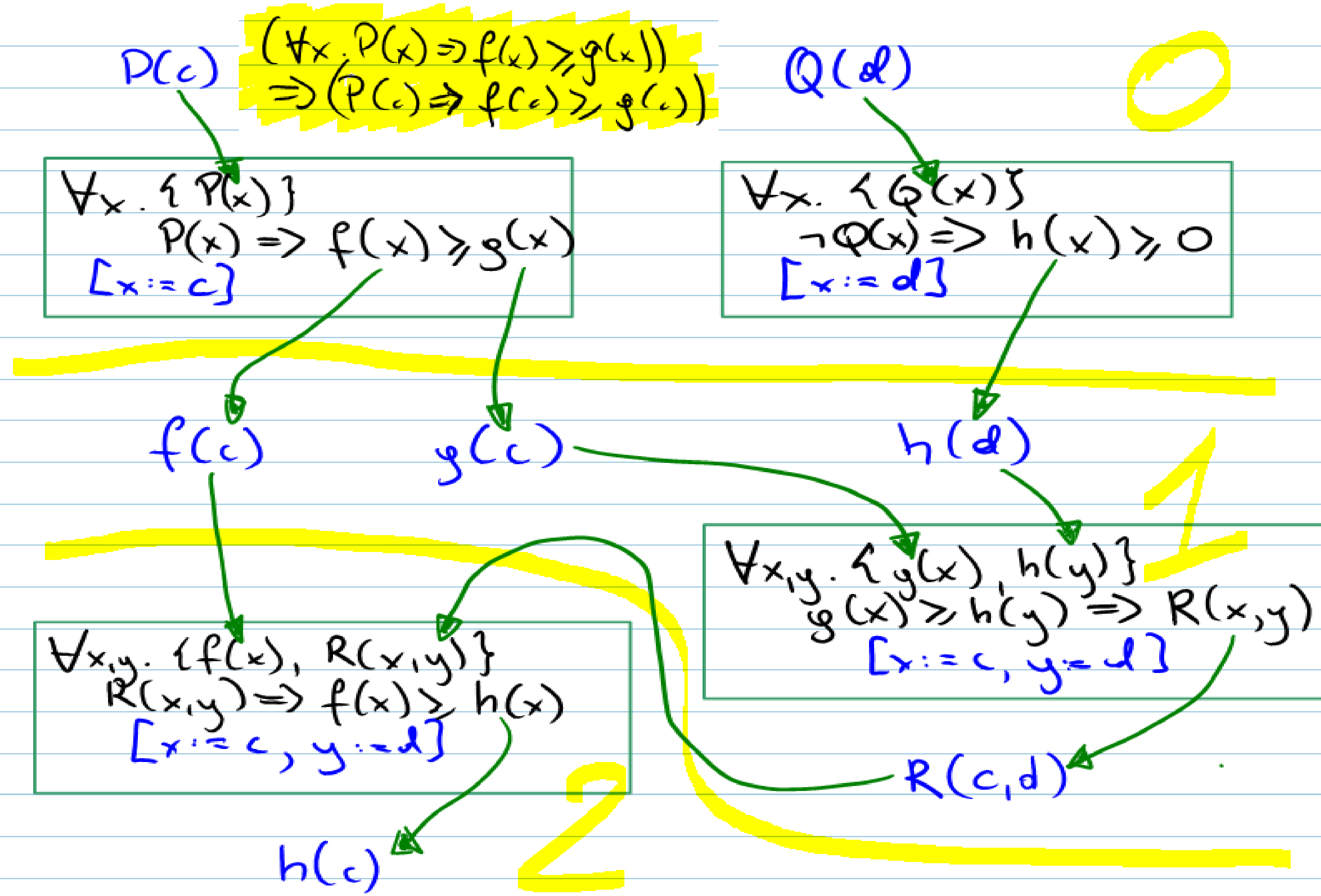
Programming With Triggers

- SMT formulas with quantifiers handled with instantiation
  - guided by E-matching, controlled by trigger annotations
- SMT theory is programmed using triggers

TRIGGERS

- (usually) subterms of the quantified formula, with free variables
- matched against active terms
  - terms with interpretation in the current partial model considered by the SMT solver

CAUSAL DAG


(∀x. P(x) ⇒ f(x) ≥ g(x)) ⇒ (P(c) ⇒ f(c) ≥ g(c))
P(c)
Q(d)
0
∀x. {P(x)} P(x) => f(x) ≥ g(x) [x := c]
∀x. {Q(x)} ¬Q(x) => h(x) ≥ 0 [x := d]
f(c)
g(c)
h(d)
1
∀x,y. {g(x), h(y)} g(x) ≥ h(y) => R(x,y) [x := c, y := d]
∀x,y. {f(x), R(x,y)} R(x,y) => f(x) ≥ h(x) [x := c, y := d]
R(c,d)
2
h(c)

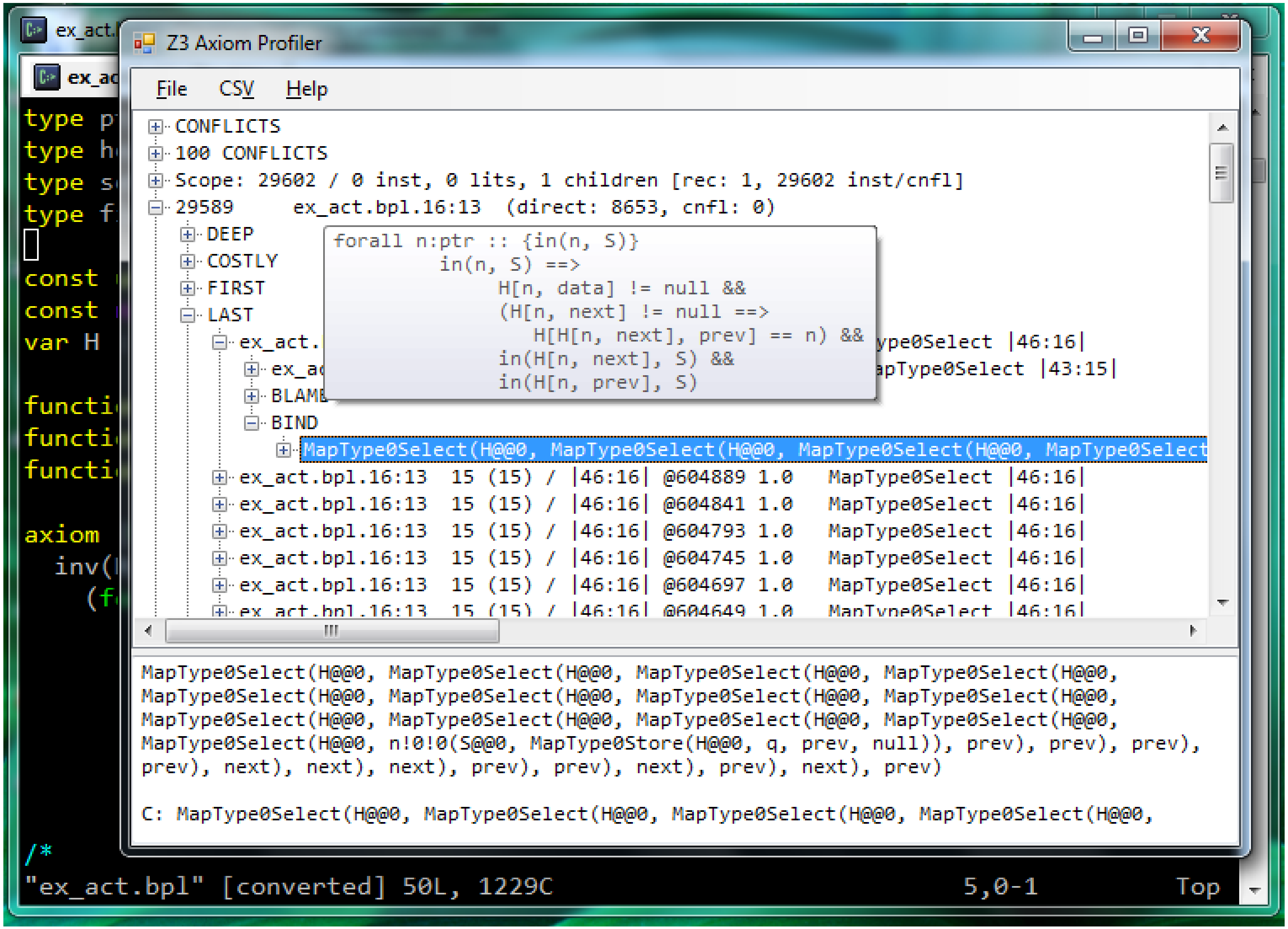
A List Invariant

- `inv(H, S)`: nodes in set S form a doubly-linked list in heap H
  - data is non-null
  - prev link in the next node points back here
  - S is next- and prev-closed

```
(forall H:heap, S:set :: {inv(H, S)}
  inv(H, S) <==>
    (forall n:ptr :: {in(n, S)}
         in(n, S) ==>
              H[n, data] != null &&
              (H[n, next] != null ==>
                 H[H[n, next], prev] == n) &&
              in(H[n, next], S) &&
              in(H[n, prev], S)))
```

DEMO OF THE AXIOM PROFILER

THE AXIOM PROFILER



Prevent Loop By Splitting Next-closedness

```
(forall H:heap, S:set :: {inv(H, S)}
  inv(H, S) <==>
      (forall n:ptr :: {in(n, S)}
        in(n, S) ==>
           H[n, data] != null &&
           (H[n, next] != null ==>
               H[H[n, next], prev] == n))

    && (forall n:ptr :: {in(H[n, next], S)}
         in(n, S) ==> in(H[n, next], S))

    && (forall n:ptr :: {in(H[n, prev], S)}
         in(n, S) ==> in(H[n, prev], S)))
```

Check a Program

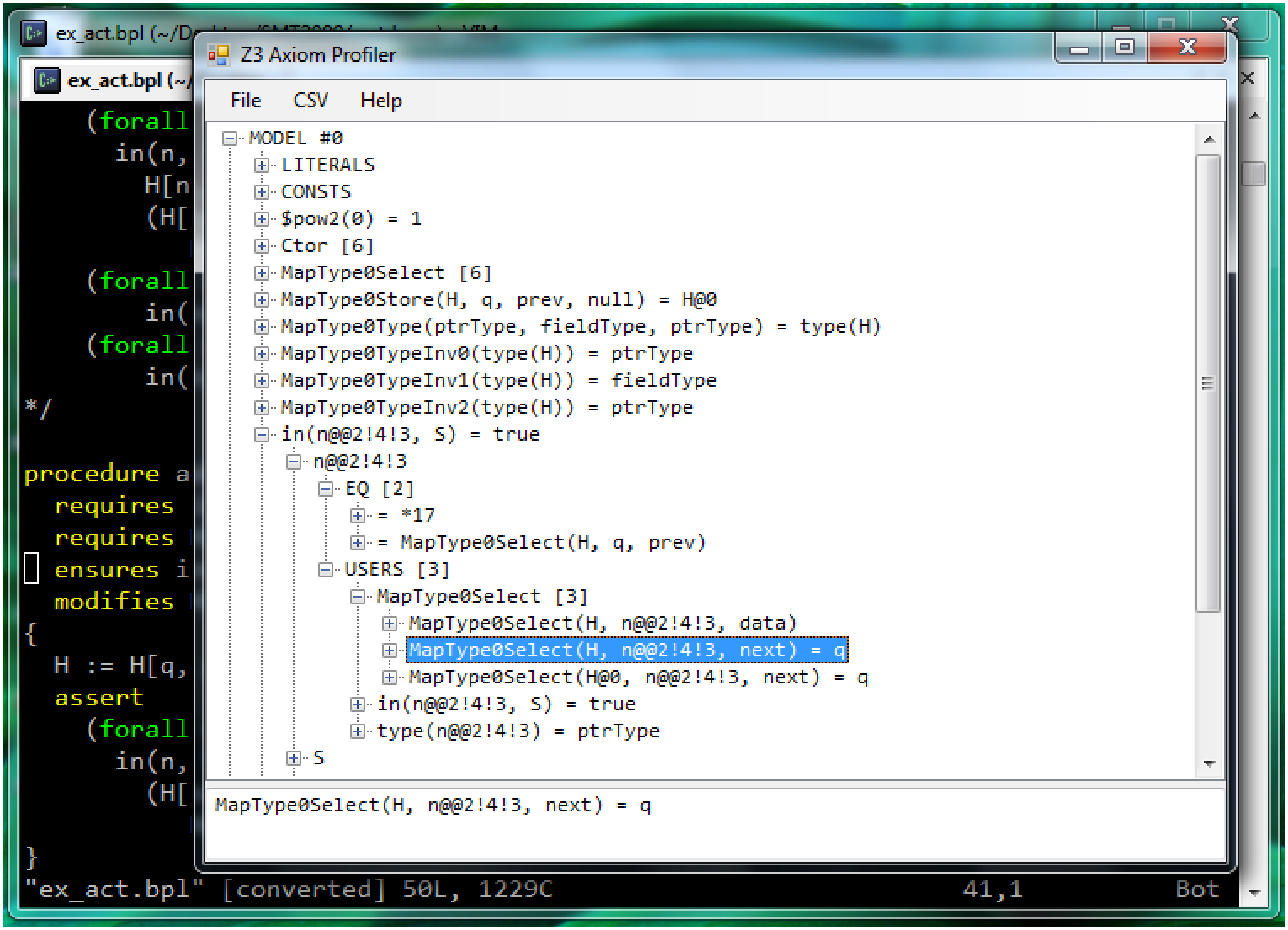
```
procedure add(S:set, q:ptr)
  requires inv(H, S);
  requires H[q, data] == null;
  ensures inv(H, S);
  modifies H;
{
  H := H[q, prev := null];
}
```

Program correct iff formula is UNSAT

```
inv(H, S) &&
H[q, data] == null &&
G == H[q, prev := null] &&
!inv(G, S)
```

DEMO OF THE MODEL VIEWER

THE MODEL VIEWER



WITNESSES

```
(forall n:ptr :: {in(n, S)}
  in(n, S) ==> H[n, data] != null &&
     (H[n, next] != null ==> H[H[n, next], prev] == n)) &&
(forall n:ptr :: {in(H[n, next], S)}
  in(n, S) ==> in(H[n, next], S)) &&
(forall n:ptr :: {in(H[n, prev], S)}
  in(n, S) ==> in(H[n, prev], S)) &&
H[q, data] == null &&
G == H[q, prev := null] &&
!(forall n:ptr :: {in(n, S)}
      in(n, S) ==>
         (G[n, next] != null ==>
            G[G[n, next], prev] == n))
```

```
in(n0, S) &&
G[n0, next] != null &&
G[G[n0, next], prev] != n
```

Z3 thinks that `G[n0, next] == q`

We know: `H[q, data] == null` and `H[n0, next] == G[n0, next]`, and thus we need: `in(G[n0, next], S)` to trigger

But n0 is a skolem constant, so it's hard for the user to introduce it.

ONE STEP AHEAD

```
!(forall n:ptr :: {in(n, S)}
      in(n, S) ==>
        (G[n, next] != null ==>
           G[G[n, next], prev] == n))
```

Whenever proving this thing, look one step ahead, or: i.e., get `in(G[n0, next], S)` activated but don't loop. Please.

Hack the SMT solver to do it :-)

```
!(forall n:ptr :: {in(n, S)}
      {ex_act(in(G[n, next]))}
      in(n, S) ==>
        (G[n, next] != null ==>
           G[G[n, next], prev] == n))
```

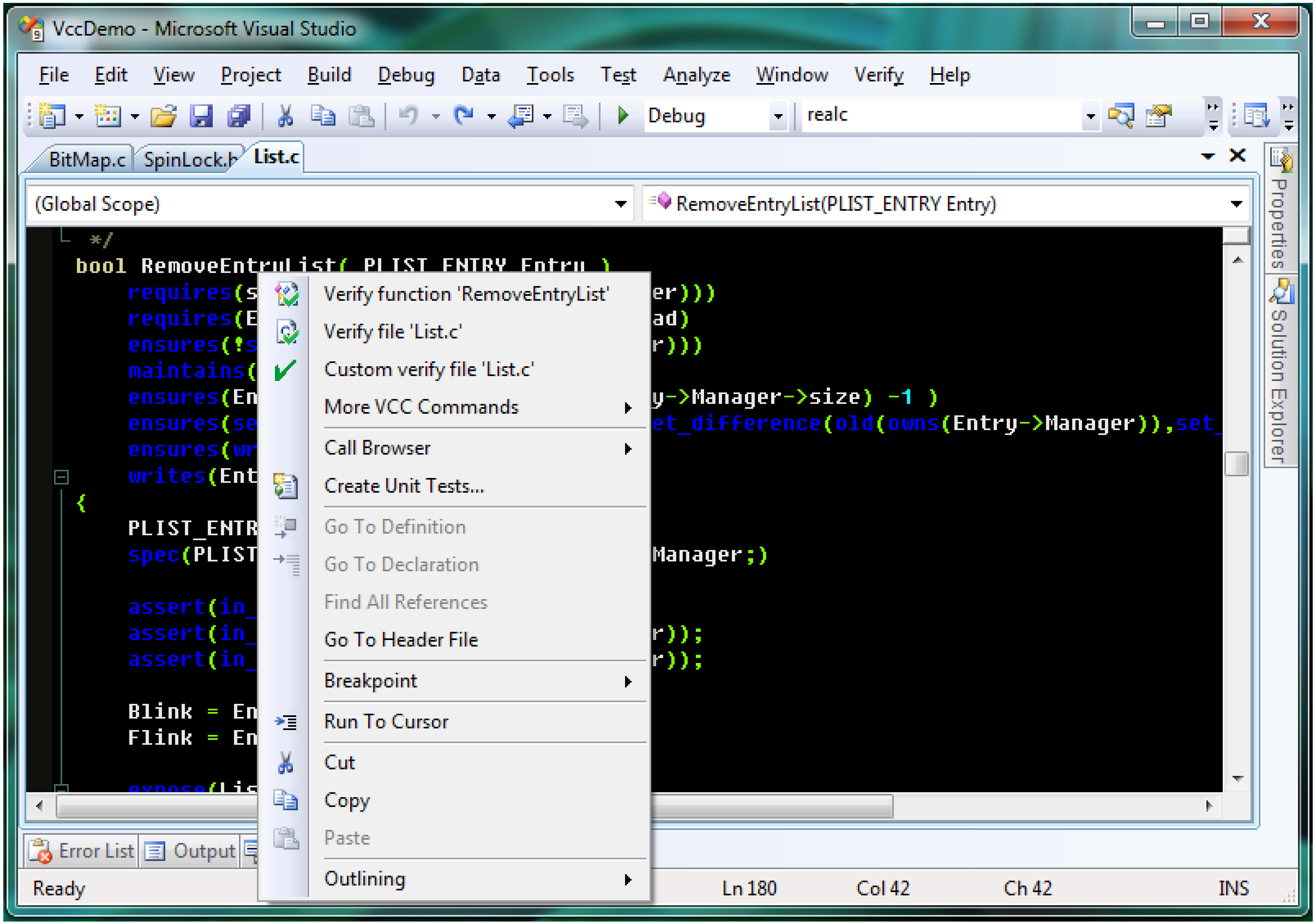
EXISTENTIAL ACTIVATION

When proving this quantifier, use lemma trigged by this

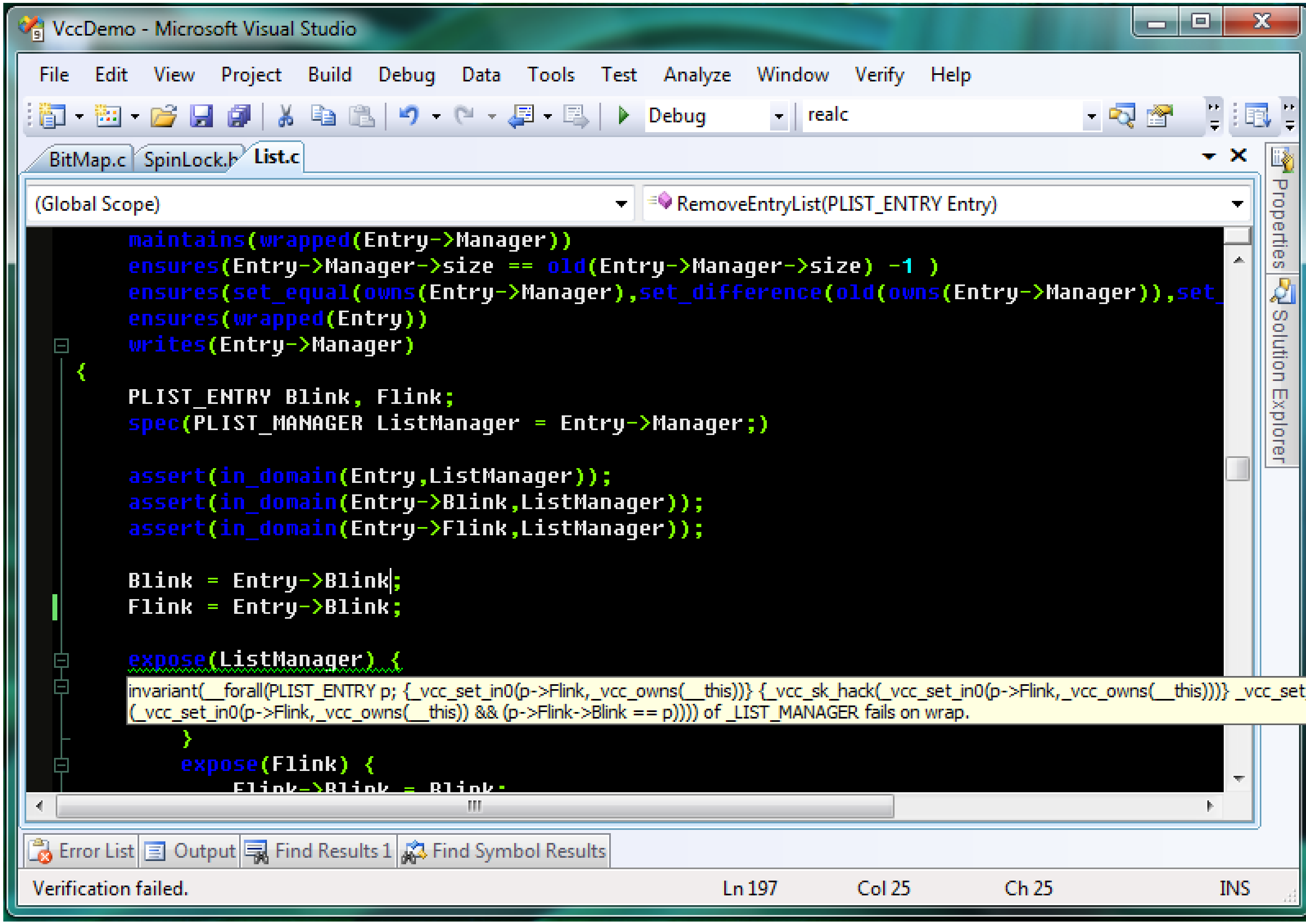
```
(forall H:heap, S:set :: {inv(H,
  inv(H, S) <==>
       (forall n:ptr :: {in(n, S)}
         {ex_act(in(H[n, next], S))}
         in(n, S) ==>
            H[n, data] != null &&
            (H[n, next] != null ==>
                H[H[n, next], prev] == n))

    && (forall n:ptr :: {in(H[n, next], S)}
          in(n, S) ==> in(H[n, next], S))
    && (forall n:ptr :: {in(H[n, prev], S)}
          in(n, S) ==> in(H[n, prev], S)))
```

VCC IN ACTION

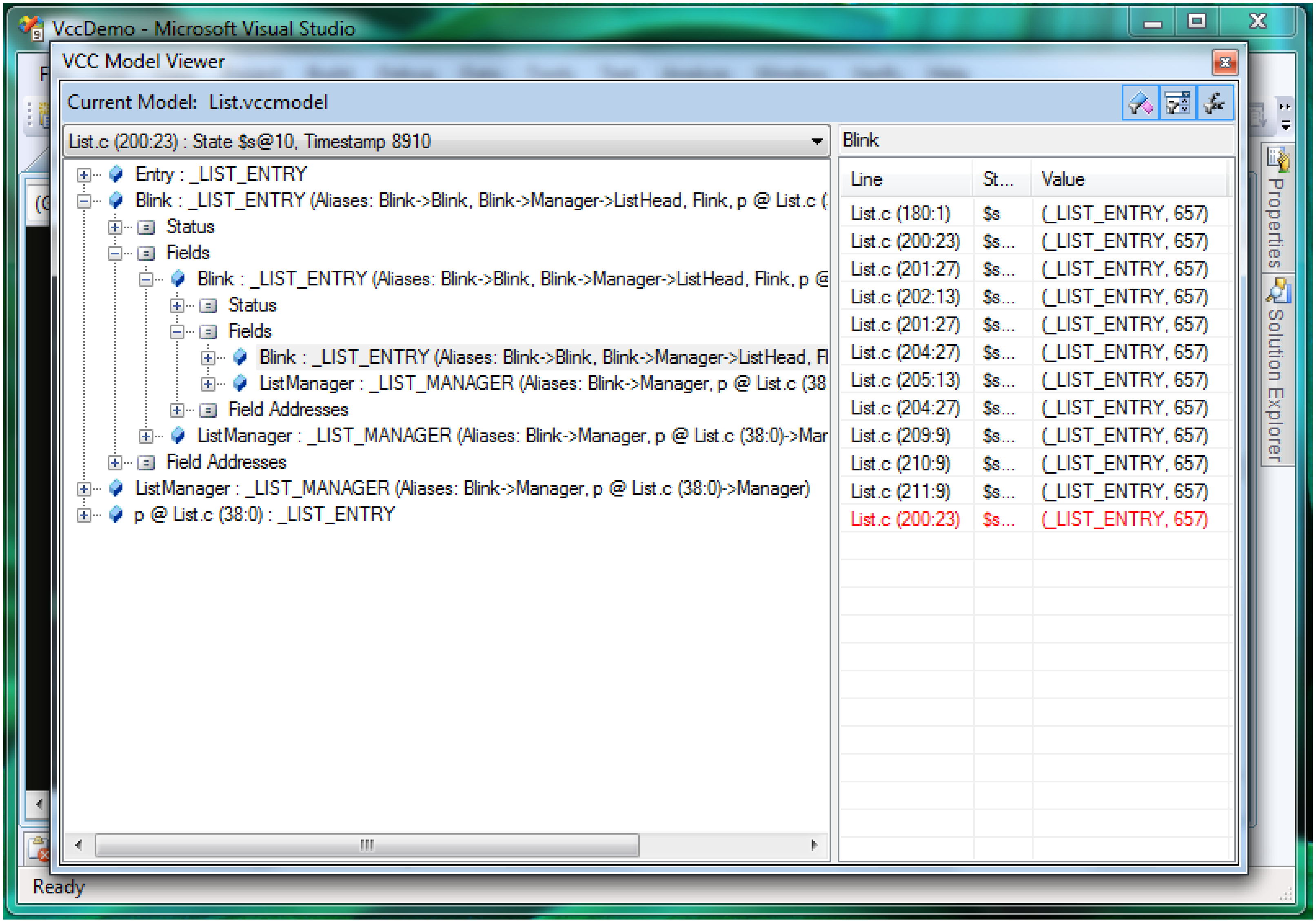
VCC Visual Studio Plugin



VCC: Failed Verification Attempt



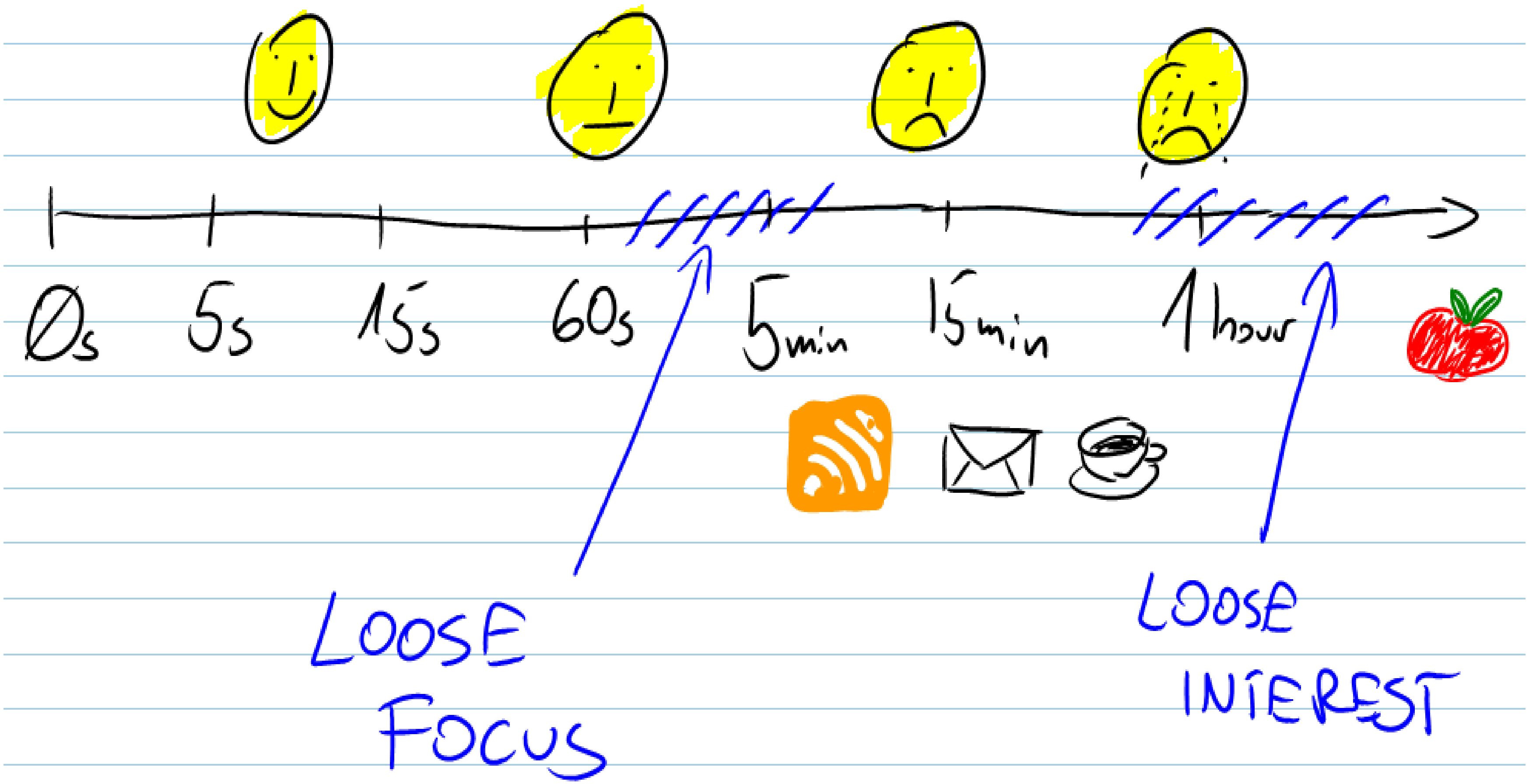
VCC-Specific Model Viewer



Working With VCC

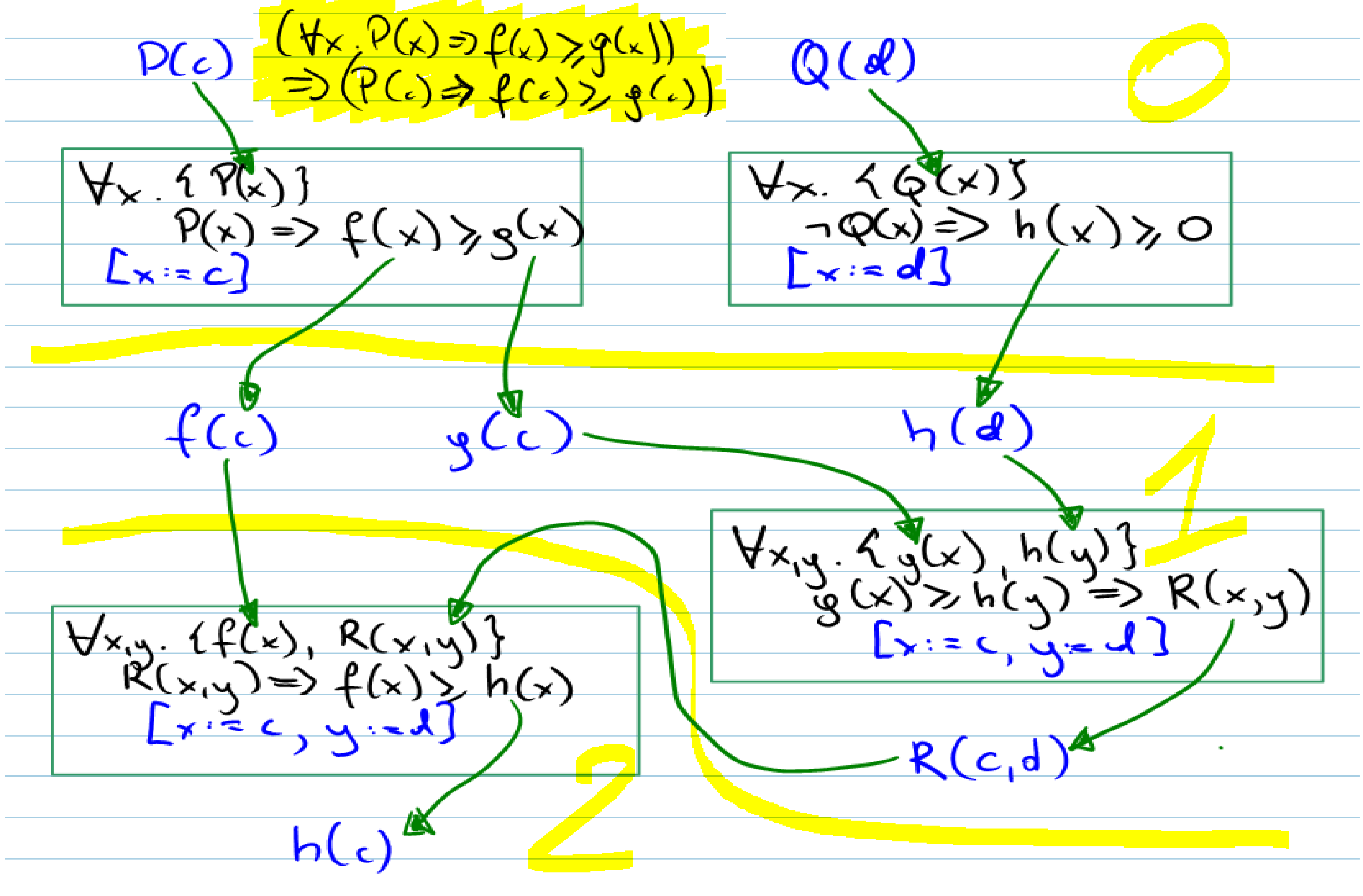
- write a version of the spec
- verify, fail
- add assertions or look at the model to see why it failed
  - for bigger functions, do it a couple of lines at a time, moving focus window down
- repeat

Verification Attempt Time vs Satisfaction and Productivity



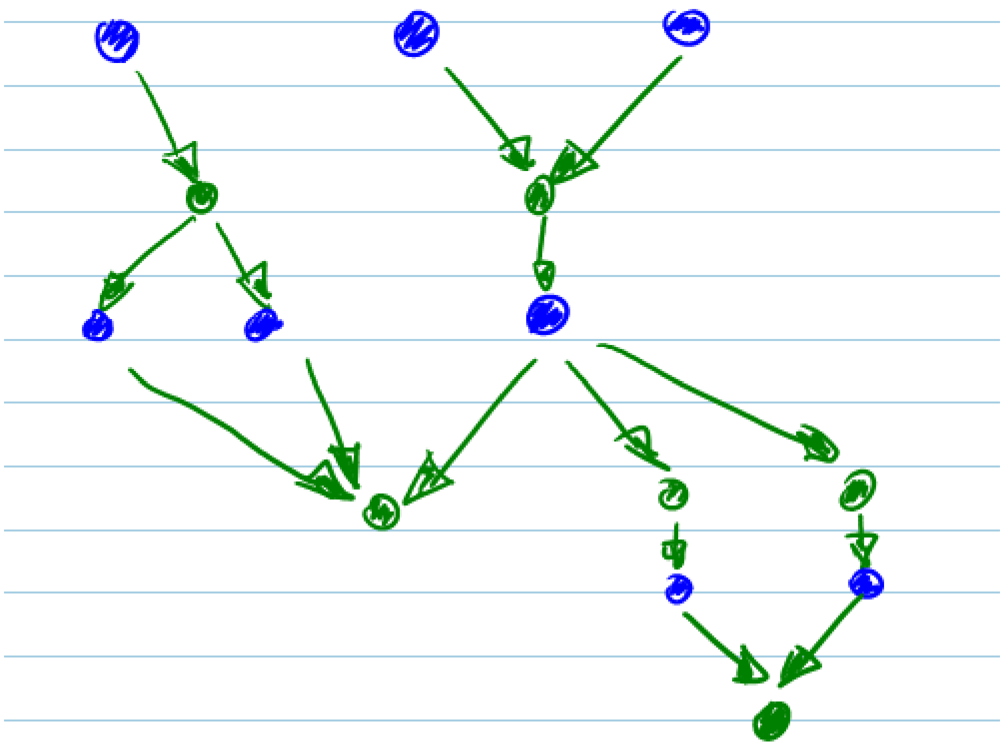
WANT IT TO GO FASTER?
PROFILE!

CAUSAL DAG

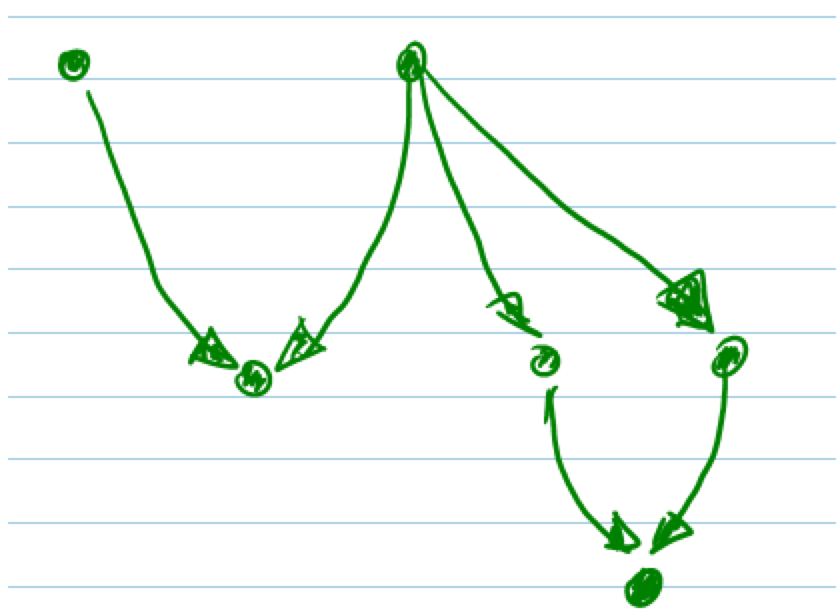


(∀x. P(x) ⇒ f(x) ≥ g(x)) ⇒ (P(c) ⇒ f(c) ≥ g(c))
P(c)
Q(d)
0
∀x. {P(x)}
P(x) => f(x) ≥ g(x)
[x := c]
∀x. {Q(x)}
¬Q(x) => h(x) ≥ 0
[x := d]
f(c)
g(c)
h(d)
1
∀x,y. {g(x), h(y)}
g(x) ≥ h(y) => R(x,y)
[x := c, y := d]
∀x,y. {f(x), R(x,y)}
R(x,y) => f(x) ≥ h(x)
[x := c, y := d]
R(c,d)
2
h(c)

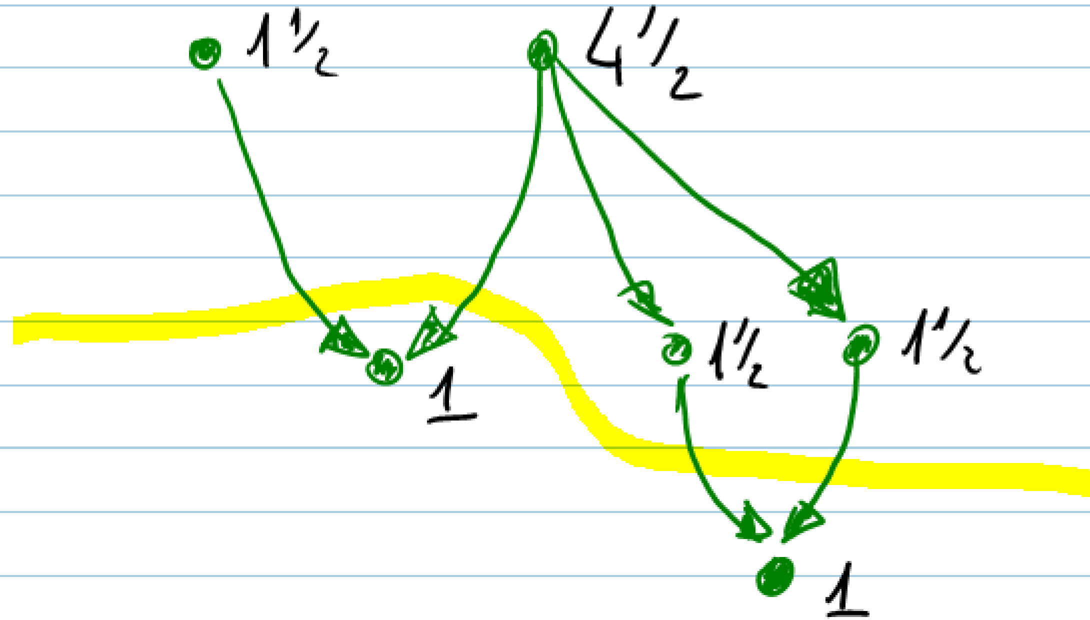
Cost In The Causal DAG



Cost In The Causal DAG

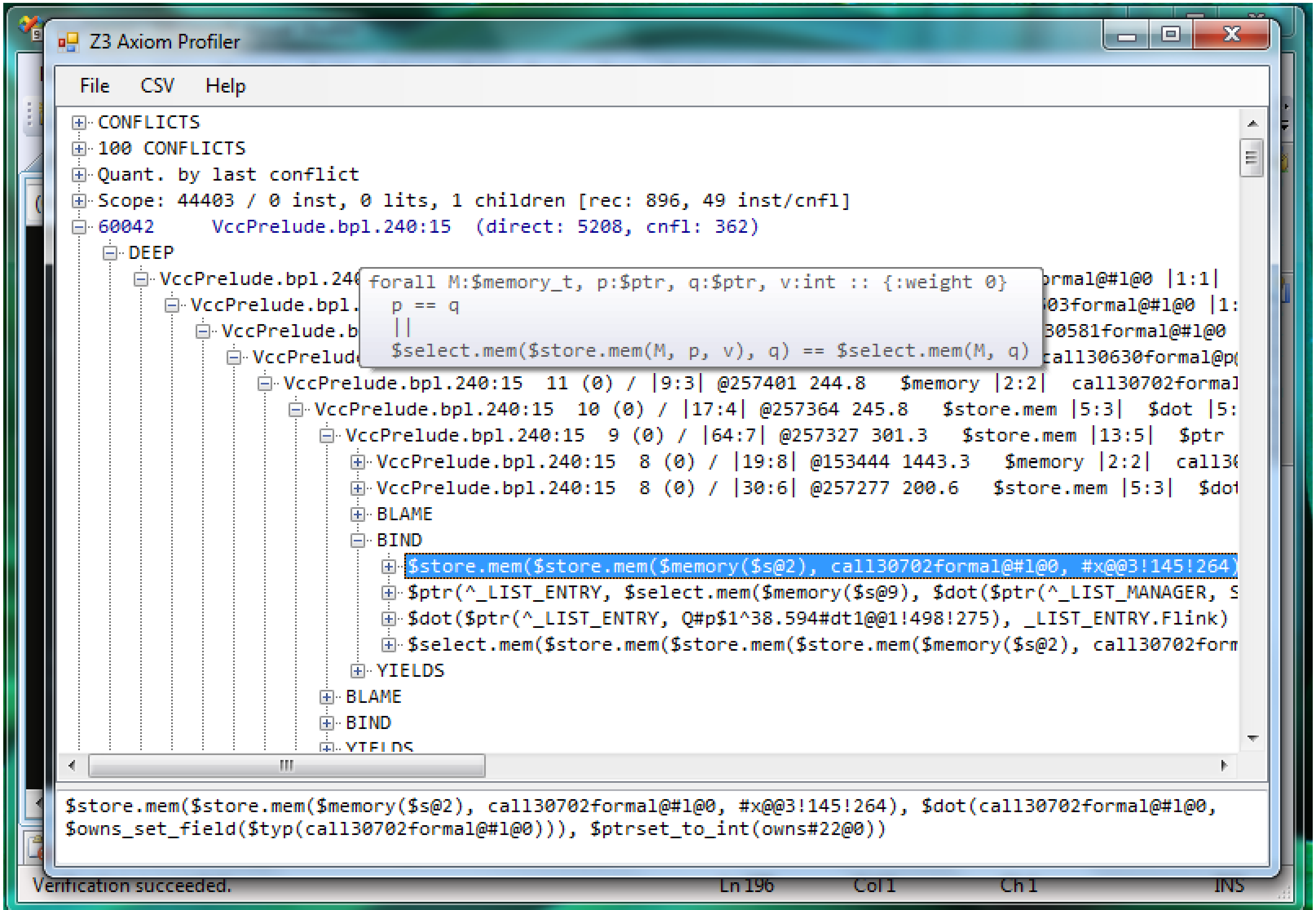


Cost In The Causal DAG



$$c(n) = 1 + \sum_{n \to m} \frac{c(m)}{\mathrm{indeg}(m)}$$

Browsing The Causal DAG

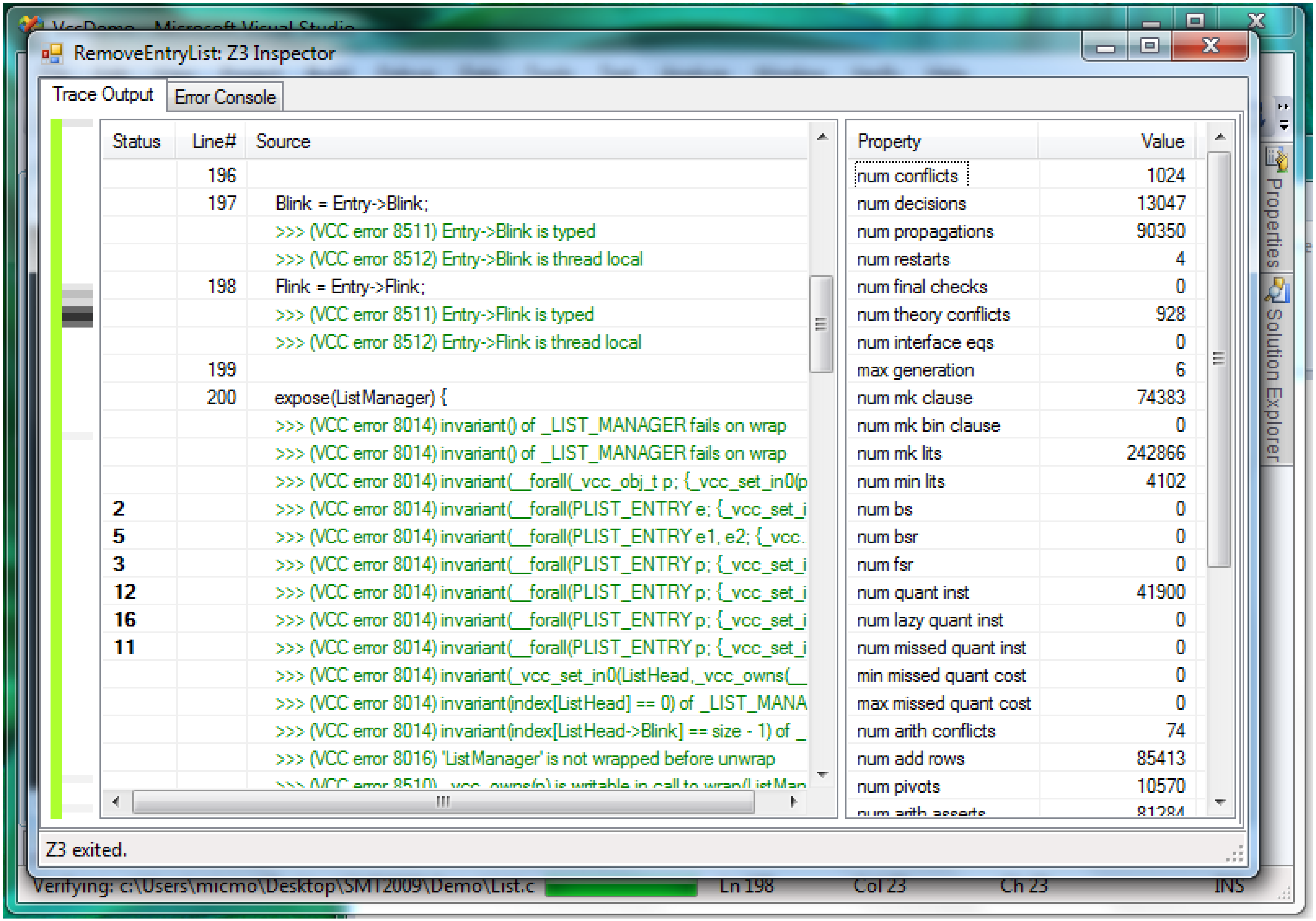


THE INSPECTOR:
CONTROL TO THE PEOPLE

THE INSPECTOR

- ask Z3 to state the current model from time to time
  - in forms of labels
- translate such models to error messages
- display all possible error message and blink the current one

SCREENSHOT


RemoveEntryList: Z3 Inspector
Trace Output
Error Console
Status
Line#
Source
196
197
Blink = Entry->Blink;
>>> (VCC error 8511) Entry->Blink is typed
>>> (VCC error 8512) Entry->Blink is thread local
198
Flink = Entry->Flink;
>>> (VCC error 8511) Entry->Flink is typed
>>> (VCC error 8512) Entry->Flink is thread local
199
200
expose(ListManager) {
>>> (VCC error 8014) invariant() of _LIST_MANAGER fails on wrap
>>> (VCC error 8014) invariant() of _LIST_MANAGER fails on wrap
>>> (VCC error 8014) invariant(__forall(_vcc_obj_t p; {_vcc_set_in0(p
2
>>> (VCC error 8014) invariant(__forall(PLIST_ENTRY e; {_vcc_set_i
5
>>> (VCC error 8014) invariant(__forall(PLIST_ENTRY e1, e2; {_vcc.
3
>>> (VCC error 8014) invariant(__forall(PLIST_ENTRY p; {_vcc_set_i
12
>>> (VCC error 8014) invariant(__forall(PLIST_ENTRY p; {_vcc_set_i
16
>>> (VCC error 8014) invariant(__forall(PLIST_ENTRY p; {_vcc_set_i
11
>>> (VCC error 8014) invariant(__forall(PLIST_ENTRY p; {_vcc_set_i
>>> (VCC error 8014) invariant(_vcc_set_in0(ListHead,_vcc_owns(__
>>> (VCC error 8014) invariant(index[ListHead] == 0) of _LIST_MANA
>>> (VCC error 8014) invariant(index[ListHead->Blink] == size - 1) of _
>>> (VCC error 8016) 'ListManager' is not wrapped before unwrap
Property
Value
num conflicts 1024
num decisions 13047
num propagations 90350
num restarts 4
num final checks 0
num theory conflicts 928
num interface eqs 0
max generation 6
num mk clause 74383
num mk bin clause 0
num mk lits 242866
num min lits 4102
num bs 0
num bsr 0
num fsr 0
num quant inst 41900
num lazy quant inst 0
num missed quant inst 0
min missed quant cost 0
max missed quant cost 0
num arith conflicts 74
num add rows 85413
num pivots 10570
Properties
Solution Explorer
Z3 exited.
Verifying: c:\Users\micmo\Desktop\SMT2009\Demo\List.c
Ln 198
Col 23
Ch 23
INS

SOME NUMBERS

LIMITS ON MATCHING DEPTH

- matching depth on one of the list functions
  - 10 heap related (10 heap updates in the function)
  - 6 user-defined (different levels of expansion of the invariant)
  - total of 17
- the looping example shows that number of instances can be easily exponential with depth

SCALE OF PROBLEMS

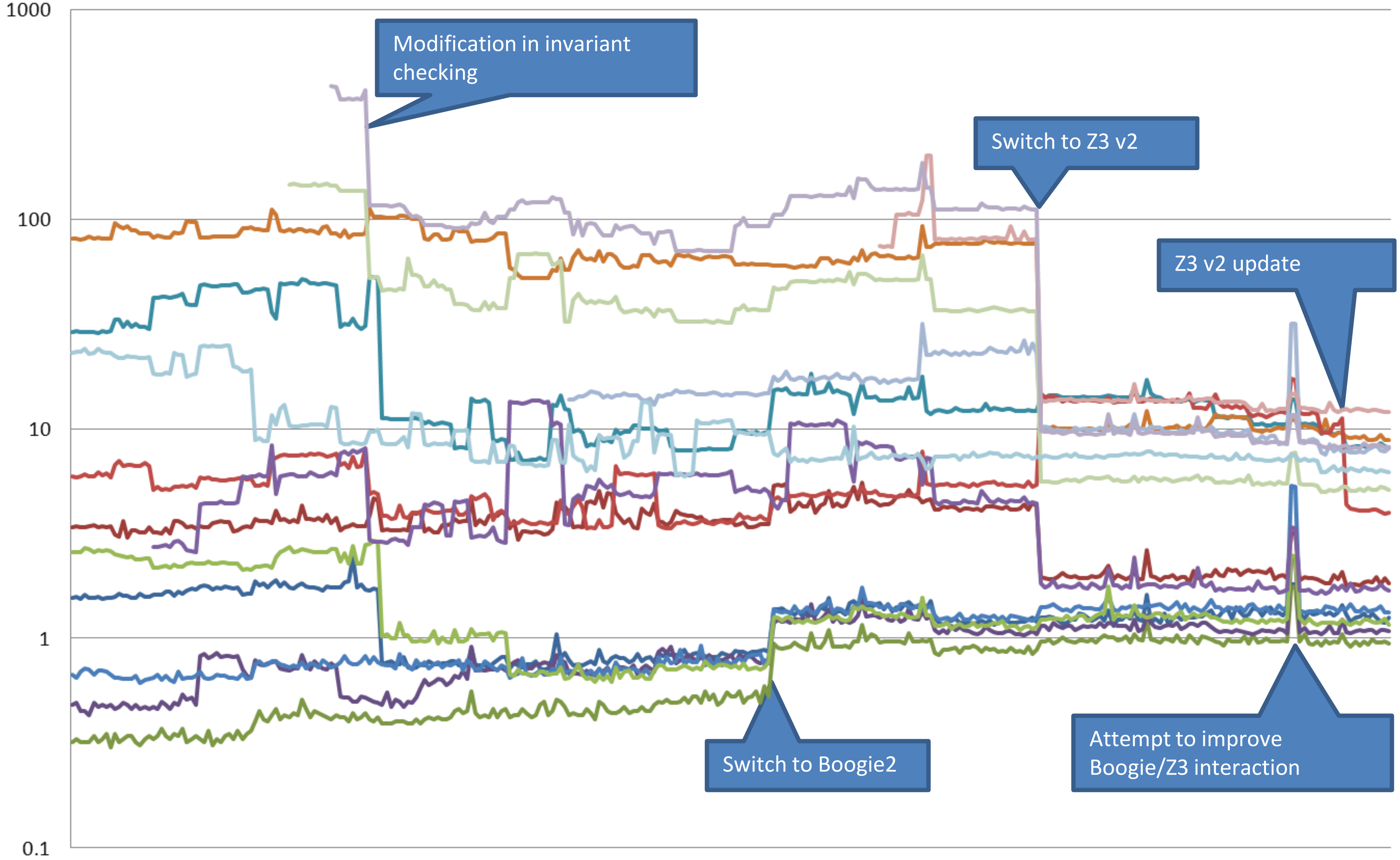
- prelude: 300 quantifiers, 50 multi-triggers
- Hypervisor program-specific background predicate:
  - 300 types, 1500 fields => 13000 axioms
  - type description dwarfs VC for the function itself
- even Z3 E-matching indices didn't take that lightly
- without proper guidance any SMT solver is likely to be lost

PERFORMANCE, PERFORMANCE, PERFORMANCE

Experience from the Hyper-V verification

- **successful** verifications:
  - typical: 0.5–500s, average 25s
  - current max: 2 500s
  - all time max: 50 000s (down to 1 000s with Z3v2)
- acceptable time for **interactive** work: < 30s
- annotations (since Nov 2008):
  - 15 000 lines
  - 400 functions
  - ca. 20% of codebase verified

VCC Performance Trends Nov 08 – Mar 09



SUMMARY

SUMMARY

- program a custom theory for the SMT solver
- like for a “normal” programming language
  - debug models (model viewers)
  - profile traces (axiom profiler)
  - profile with sampling (the inspector)
- but:
  - lack of clear semantics
  - possibly not the best programming model

VCC Is Available!

- source code available for non-commercial purposes at http://vcc.codeplex.com/
  - includes the SMT tools!
- further information linked from there

THANK YOU!