
Incremental Evaluation of OCL invariants in the Essential MOF object model

Miguel Garcia, Ralf Möller

<http://www.sts.tu-harburg.de/~mi.garcia>

2008-03-12

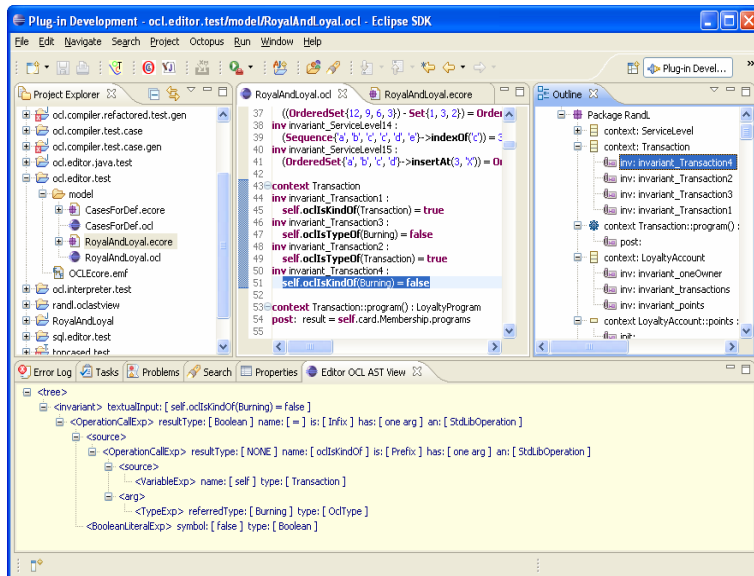


Agenda

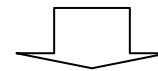
- **Problem Statement**
- **The tools at our disposal:**
 - interception of updates,
 - compile-time and runtime activities
- **Examples**
- **Improvements over previous work**
- **Future Work: Software Transactional Memory**
- **Lessons Learnt**

Not all that looks good works well. What about OCL Invariants? Let's see

- Pros:**
1. OCL allows expressing real-world constraints
 2. Tooling for authoring OCL is catching up with the features of other Eclipse-based text editors (e.g., refactoring)
 3. Compilation into Java is available



```
context LoyaltyAccount
inv invariant_points : self.points > 0
implies transactions->exists(t | t.points > 0)
```



```
public boolean invariant_points () {
    Boolean implies1 = (this.getPoints () > 0);
    if (!(implies1)) {
        implies1 = Boolean.TRUE ;
    } else {
        Boolean exists2 = false ;
        for ( RandL.Transaction t :
            org.eclipse.ocl.util.CollectionUtil.asSet(
                this.getTransactions())) {
            if (!exists2) {
                exists2 = (t.getPoints () > 0);
            }
        }
        implies1 = exists2 ;
    }
    return implies1 ;
}
```

However: Repeated evaluation of OCL invariants causes a noticeable performance slowdown

This problem has two variants:

1. The database case (e.g., objects in a software repository)
2. The main-memory case (e.g., a tool for editing Domain-Specific models). [We cover this case]

Some thinkable workarounds:

1. Manually tweak compiled code to skip redundant evaluations
2. Use only a subset of OCL, avoiding “time-consuming” constructs (e.g., collection operations)

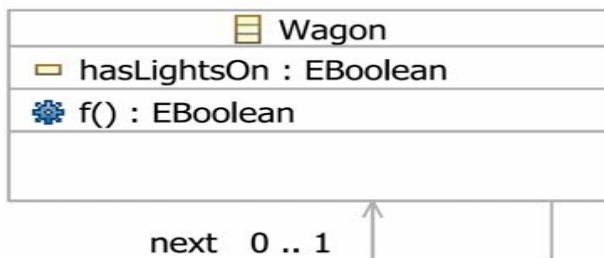
These workarounds are not without disadvantages:

1. Completely defeat the productivity gain of declarative OCL
2. Error-prone

We can do better than that, by replacing
~~from-scratch recomputation~~
in favor of incremental evaluation

Some useful facts about OCL invariants

- They are side-effect-free boolean functions, usually containing field-navigation expressions
- In any given evaluation, control flow constructs (e.g., *if-then-else*) determine that:
 - some data locations are read (making the result depend on them)
 - other data locations **are not accessed**. Therefore, **their updates do not affect the last computed value of the invariant**
- In the *if-then-else* example, if the condition is true then only accesses in the *then* part count towards the result



```
context Wagon
inv    lastWagonHasLightsOn : f()

context Wagon::f()
def    : if next.oclIsUndefined()
      then hasLightsOn
      else next.f()
      endif
```

Technical term: *Memoization* (functional programmers, rejoice!)

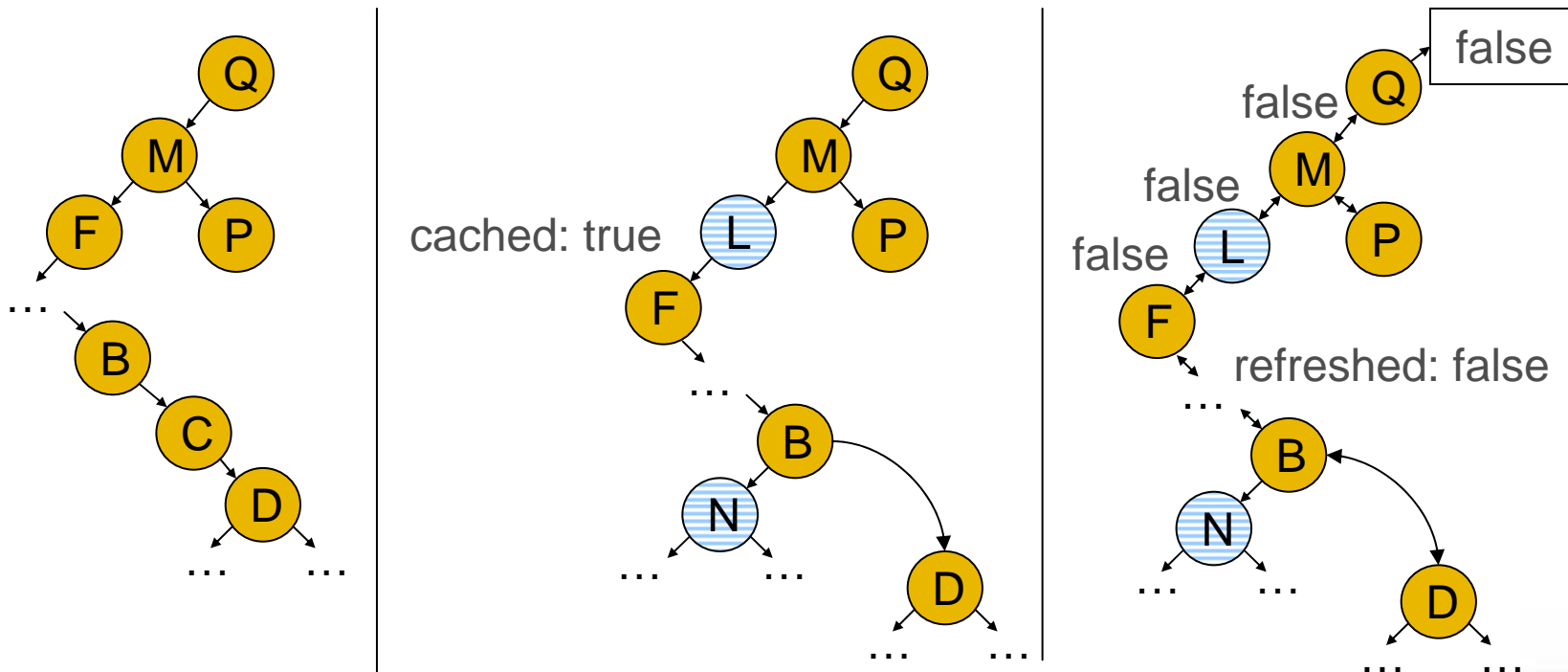
- What if instead of evaluating a function one could simply look up its result in a table? (Sounds like a CPU vs. memory tradeoff)
- It goes by the name of memoization. In our setting the trick is to:
 - determine the key used to lookup in such table (i.e., the inputs to a function evaluation, nothing more, nothing less)
 - Manage the table to account for updates in the underlying object population (e.g., prune those entries whose inputs have been garbage collected)
- “Inputs” is a catch-all term encompassing:
 - Explicit arguments: those in the argument list, plus *self*
 - Implicit arguments: data locations in the heap accessed with “dot navigation”
 - Subcomputations: invocations to functions defined in OCL

```
context Node def : height() : Integer =  
if children->isEmpty()  
  then 1  
  else 1 + children->collect(c | c.height() )->max()  
endif
```

Optimistic subcomputations: go ahead and use the cached result, correct later if it was stale

Life and times of a misprediction in a sorted binary tree:

```
boolean isSorted(Tree t) {  
    if (t == null) return true;  
    if (t.left != null && t.left.value >= t.value) return false;  
    if (t.right != null && t.right.value <= t.value) return false;  
    return isSorted(t.left) && isSorted(t.right);  
}
```



Improvements made possible by EMOF + OCL over previous work on Java memoization

Two optimizations reducing the runtime memory footprint:

- no need to track individual collection items: the collection object suffices to keep track of mutations (e.g., `move(from, to)`, `add(item)`, etc.)
- intermediate results are not cached in full (updates to their base data signal that recomputation should take place)

An optimization reducing the triggering of redundant recomputation:

- Only those mutator methods on collections that influence the outcome of an evaluation trigger a (costly) re-evaluation.

For example, the result of `size()` is invariant under reorderings of the base collection. Therefore, `move(from, to)` on that collection should not result in flagging invocations as dirty

A bonus safety check:

- Certain cases of infinite recursion can be detected

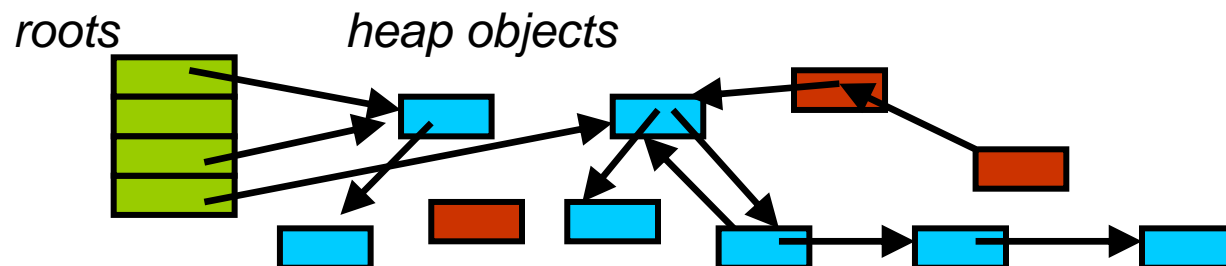
Future Work: Software Transactional Memory (STM) in EMOF managed runtimes

The idea, intuitively:

```
void deposit(...) { atomic { ... } }
void withdraw(...) { atomic { ... } }
int balance(...) { atomic { ... } }

void transfer(Acct from, int amt) {
  atomic {
    //correct and parallelism-preserving!
    if(from.balance() >= amt && amt < maxXfer) {
      from.withdraw(amt);
      this.deposit(amt);
    }
  }
}
```

With incrementalization, we are already tracking updates:



Reproduced from: Dan Grossman, *Software Transactions: A Programming-Languages Perspective*, http://www.cs.washington.edu/homes/djg/slides/grossman_googleFremont08.ppt

Lessons learnt

Typical costs and benefits of extending a modeling infrastructure:

- Learning curve made steep by code-only documentation (online docs cover introductory to intermediate topics)
- Once that entry barrier has been overcome, additional functionality comes at a much lower cost (as exemplified by transactional memory)

Further challenge: being early adopters of static analysis/verification:

- Beyond basic analyses (supported by FindBugs, CheckStyle plugins)
- Hoare-style verification of Java bytecode (<http://sdg.csail.mit.edu/forged/>)
- Type system improvements, such as (Java 7) Object and Reference Immutability (related to JSR-308, <http://groups.csail.mit.edu/pag/jsr308/>)
- Typestate checking in the presence of aliasing, where the typestate of a reference depends on that of reachable objects (e.g., the `Iterator` interface). Work in this direction: <http://www.research.ibm.com/safe/>
- Verification of temporal properties (call sequences, some liveness properties). A representative prototype: <http://lifc.univ-fcomte.fr/~jag/>

And many more. So, let's stay tuned for Modellierung 2009!