Incremental evaluation of OCL invariants in the Essential MOF object model

Miguel Garcia, Ralf Möller

Institute for Software Systems (STS)
Hamburg University of Technology (TUHH), 21073 Hamburg
http://www.sts.tu-harburg.de/

Abstract: The management of metamodels is supported by runtime environments that enforce the well-formedness of (meta-)model instances. Beyond this basic functionality, additional capabilities are needed in order to successfully establish a toolchain for Model-Driven Software Engineering. We focus on two such capabilities: transactions and efficient evaluation of invariants, not in the usual context of databases but for main-memory runtime engines, an area where no previous work has addressed the combination of Essential MOF + OCL. The realization of this infrastructural support proves feasible but requires a careful design to accomodate the expressiveness of OCL.

1 Introduction

The combined expressive power of Essential MOF (EMOF) [Obj06a] and the Object Constraint Language (OCL) [WK03] has proved satisfactory in Model-Driven Software Engineering (MDSE) to define the abstract syntax and static semantics of custom Domain Specific Languages (DSLs). Benefits include: (a) evaluation of constraints at runtime [GS07]; (b) improved reflection ("models-at-runtime"); and (c) extended type system [Kya05].

Nowadays these benefits are realized by a combination of compilation and framework reuse. The components resulting from model-based generation are ready for integration into authoring tools, for example tools based on a Model-View-Controller architecture. However, these components fall just short of supporting two increasingly important runtime requirements: efficient evaluation of invariants and transparent concurrency. To support them out-of-the-box, we extend our compilation algorithm [GS07] to support incremental evaluation [SB07] and shared-memory transactions [HG06]. To address the first concern, an interception mechanism is used to detect those data locations that have been updated. Invariants dependent on them are candidates for re-evaluation as their cached values may have become stale. Other invariants are not affected, and their evaluation can be skipped. The second technique allows for a programming style where ACID properties are enforced for a block of statements, yet no coding of locking operations is required. In effect, the runtime system keeps rollback logs and detects interactions which lead to failure of memory transactions.

As a motivating example, consider a multi-user editor of statecharts, which allows manip-

ulating a single shared instance of the statechart metamodel. The runtime infrastructure should enforce transaction bracketing to avoid interleavings of read-write accesses by distinct threads that corrupt the shared data structure. Additionally, the runtime engine should report those invariants not evaluating to `true` on instances at transaction commit. The need for concurrency management is not limited to the multi-user case: a single-user tool may run background tasks, or a single user may perform multi-step updates on different views of the same model (e.g., during *round-tripping* between textual and visual views).

Our incrementalization technique makes the DITTO algorithm [SB07], originally formulated to incrementalize a Java subset, available for EMOF. By exploiting the semantics of EMOF + OCL, we realize additional improvements: (a) incrementalizing collection operations required tracking each item as an implicit argument in [SB07], our formulation instead tracks the involved collection objects, not their items, thus reducing the memory footprint. (b) Intermediate results of invariants evaluation are not cached in full, updates to their base data are intercepted instead. (c) Certain cases of infinite recursion can be detected, a capability not present in DITTO. (d) Finally, only those mutator methods on collections that influence the outcome of an evaluation trigger a (costly) re-evaluation. The proposed algorithm is applicable to any EMOF realization, for example Eclipse EMF.

The structure of this paper is as follows. Sec. 2 provides background, including overviews of DITTO (Sec. 2.1), its EMOF adaptation (Sec. 2.3), and the termination analysis of the evaluation of OCL expressions (Sec. 2.2). Sec. 3 details the compile-time aspects of our solution, while Sec. 4 covers the runtime aspects and reviews the design choices made. Sec. 5 addresses support for shared-memory transactions, with Sec. 6 discussing related work and Sec. 7 conluding. Knowledge is assumed about object-oriented query languages. Familiarity with shared-memory concurrency (locks and condition variables) is necessary to follow the discussion in Sec. 5.

2 Incrementalization

An incremental algorithm computes anew only those intermediate results that have been affected by changes in the previous input, reusing cached results for non-affected subcomputations. Manual incrementalization is error-prone, thus motivating automation. Given a finite object population, every no-args (i.e., parameterless, except for `self`) side-effect-free method is amenable to incrementalization. In terms of OCL this comprises: (a) class invariants, (b) derived object attributes, and (c) derived no-args object operations. Such parameterless methods need not be constant, given that they usually navigate the object structure (starting from `self`) to compute their result, thus reading *implicit arguments*.

Checking OCL invariants is beneficial both at debug time (as they combine the advantages of continuous testing and "declarative data-breakpoints") as well as during operational use. Their main disadvantage is the runtime slowdown (100x are not uncommon), as their naïve evaluation may involve traversing entire data structures. An incremental algorithm, instead, reuses cached results of subcomputations whose inputs can be proven not to have changed. This fits the typical runtime behavior of OCL invariants: they aggre-

gate further invariant checks on fragments of a data structure, with those subcomputations usually returning "the same previous value" (i.e., the value leading to a satisfied invariant) even for modified inputs, as most updates preserve consistency. As a result, upstream computations do not become stale, and their evaluation can be skipped. Pointer aliasing complicates keeping track of all program locations that may mutate a given data location, with interception techniques coming to the rescue, as all updates to data locations are only possible through well-known methods in EMOF (setters and their counterparts to mutate collections).

Two candidate techniques to incrementalize OCL invariants are: (a) memoization [SB07] and (b) view materialization [AFP03]. A spreadsheet analogy can be used to explain the operation of materialization: the availability of changed inputs triggers the recomputation of dependent values, avoiding redundant recomputations by using a *dynamic dependency graph* (DDG). Because of object instantiation, updates, and garbage collection, the topology of the underlying object population changes at runtime and the DDG has to be kept in-synch with it. In contrast, memoization happens on-demand: whenever a function is invoked, the cached values for its inputs are compared to those in the current *system snapshot*. If they match, the cached ("memoized") return value can be reused.

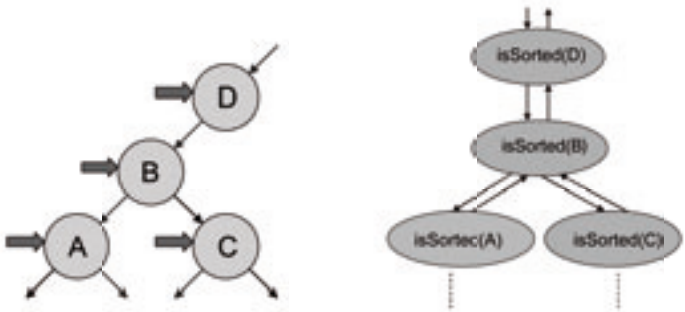
A big pitfall of unoptimized memoization involves subcomputations (a special case of input): in order to compare their cached and updated values, a subcomputation needs in principle be invoked, which in turn may need to invoke its own subcomputations (if any) to decide whether to reuse memoized return values or not. For example, assuming the usual recursive formulation of the `height()` function on trees:

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

Given a node n and a cached evaluation of `height()` for each of its children, knowing that the set of children has not changed does not entitle to reuse the cached n`.height()`, as the topology downstream may have been updated. In terms of memoization, knowing that the implicit arguments have not changed (the children of n) does not preclude subcomputations (another kind of input) from having changed. Implicit arguments comprise those data locations acccessed directly by a function evaluation, not by its callees.

Optimistic memoization [SB07] assumes instead such subcomputations will behave as in the typical case, thus skipping their invocation. This may lead to mispredictions, which are detected in all cases, as discussed in Secs. 3 and 4 (incrementalization is sound and complete, i.e. neither false-positives are reported nor broken invariants are overlooked).

Our mechanism of choice to achieve incrementalization is thus *collection-aware optimistic memoization*, to be activated on-demand at transaction-commit time achieving the same effect as naïve evaluation of all invariants on all instances. Read accesses take place within a transaction and thus do not observe partial results (dirty reads). The interaction with the mechanism for shared-memory transactions (Sec. 5) is safe, as such mechanism can cope with both read and write accesses. In effect, the incrementalization concern is orthogonal to memory atomicity, with the former being layered upon the latter.



(a) Fragment of a binary tree

(b) Dynamic Dependency Graph (DDG) for `isSorted()`

Figure 1: Visualization of the invocations for `isSorted()`

2.1 A first example of incrementalization

In essence, DITTO caches a *function evaluation* by collecting additional information at runtime besides computing its result. In particular, the actual arguments are recorded in a global data structure, with such bookkeeping information being kept across invocations. Reading an object-field during a function evaluation also results in the pair *(instance, field ID)* being tracked as an *implicit argument* of that particular invocation. During the update phase of a transaction (in the transaction body), all setters are intercepted and thus a mapping enables the lookup (using *(instance, field ID)* as key) of those function evaluations that require re-evaluation (also called "refreshing the cached return value"). Another event that forces recomputation is garbage collection of an implicit argument, a situation detected by tracking instances with *weak references*[1]. At the time a weak reference is created or a `WeakHashMap` entry is made, a listener is registered to be notified upon the referenced object becoming unreachable.

In order to introduce terminology, a Java method for checking whether a binary tree is locally sorted (Listing 1) is reproduced with modifications from a DITTO presentation[2]. For the fragment of the binary tree displayed in Figure 1(a), `isSorted()` is invoked for each node marked with an arrow, to compare the values (letters in this case) displayed inside each node. In terms of our tooling approach, invariants are formulated in OCL and translated automatically into Java [GS07].

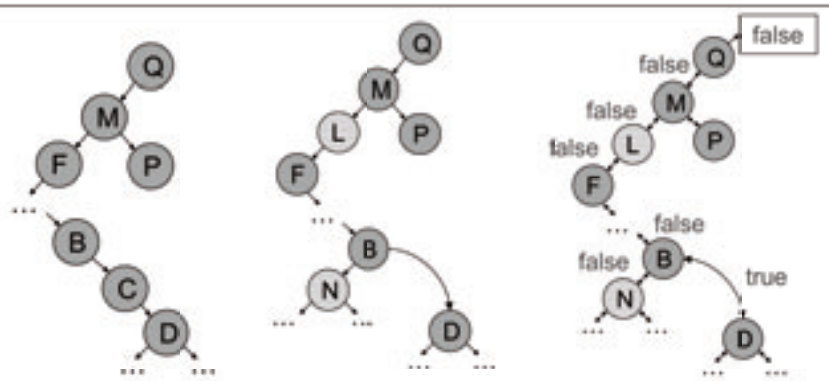
A visualization of the Dynamic Dependency Graph (DDG) for `isSorted()` appears in Figure 1(b). Without memoization, no record is kept of invocations once they have terminated, i.e. no "call-stack unrolling" is available in-memory. A formal definition of DDG is given in Sec. 2.3, each DDG node contains for example a cached return value.

[1] An introduction to weak references is `http://weblogs.java.net/blog/enicholas/archive/2006/05/understanding_w.html`. A longer but slightly out-of-date discussion appears in `http://java.sun.com/developer/technicalArticles/ALT/RefObj/`

[2] AJ Shankar, presentation at PLDI'07, `http://ditto-java.sourceforge.net/ditto.ppt`

Listing 1: A non-instrumented Java method to check whether a binary tree is locally sorted

```
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);
}
```



(a) Tree before update (b) After insertion of L, N and deletion of C (c) Upstream propagation in the DDG after misprediction for B

Figure 2: Incremental evaluation of the invariant `isSorted()` on a binary tree

As depicted in Figure 2 (a) and (b), a particular update on a tree consists in adding nodes `L`, `N` (the second out-of-order) and removing node `C`, as part of a single transaction, which should fail. During such transaction, at least one field in each of `M`, `L`, and `B` has been updated (i.e., fields that are read by `isSorted()`, for some invocations). Refreshing a DDG node (i.e., evaluating a function again so as to replace a *stale* cached result) is necessary after one or more of its implicit inputs have been updated.

At this point, the DDG contains stale information: although `L` is lexicographically smaller than `M`, `N` does not come in the alphabet before `B`, yet the current DDG node for `isSorted(B)` has a cached result of `true`. DITTO introduces an optimization to minimize the cost of refreshing, once the invocations with modified inputs have been identified. First, the invocation for `M` is executed again ("replayed"), which in turn activates for the first time an invocation for node `L` (resulting in a node being added to the DDG). The problem with evaluating `isSorted(L)` is that it depends recursively on `isSorted(B)`, whose cached result is not only stale but also wrong. However, DITTO assumes (for now) that downstream invocations need not be replayed (thus *optimistic*). This decision is suspect until replaying `isSorted(B)` reveals a wrong assumption was made. In that case, a propagation from callees to callers as depicted in Figure 2(c) will be performed.

Setters and collection mutators are intercepted using *Code Instrumentation*, a technique to automatically transform programs. For example, *profiling* is commonly implemented that way, to gather information about the performance of the instrumented program.

2.2 Termination behavior of the evaluation OCL expressions

The evaluation of OCL expressions containing recursive function invocations is not guaranteed to terminate in a finite number of steps (in the general case). This can be seen for: `def stackOverflow( arg : Boolean) : Boolean = stackOverflow(arg)` Similar examples can be built for indirect recursion. However, the special case of non-recursive expressions over a finite object population is relevant for our incrementalization technique. For them, termination can be shown by case analysis on the structure of OCL language constructs, as discussed next.

Applying a function outside its domain (e.g. requesting the first element in an empty list or dividing by zero) results in `OclInvalid`: unlike in Java, no exception is thrown. A finite population is obtained when applying `allInstances()` to a class (in contrast, the expression `Integer.allInstances()` is not well-formed). OCL is a function-based rather than a functional programming language: functions are not first-class-citizens, lambda abstractions cannot be built with the available language constructs and thus cannot be passed as arguments or returned as values. Regarding collection operations, the non-recursive subcases of `LoopExp` amount to linear iteration (`select`, `reject`, `exists`, `forAll`, `collect`, `one`). The remaining subcases can be desugared to their `iterate` form as defined in the OCL standard ([Obj06b], Sec. 11.9 and A.3.1.3). `iterate()` in turn can be expressed as left-fold, a primitive recursive function with a finite-depth tree expansion (under the stated assumptions of finite population and non-recursive invocation).

2.3 Adaptation to Essential MOF + OCL

The object models of Java and EMOF mostly overlap but neither of them is a proper subset of the other. An EMOF implementation (such as Eclipse EMF) enforces the semantics of EMOF by mediating the manipulation of Java objects through generated methods. For example, besides assigning a new value, setters also take care of performing "reference handshaking" for the participants in a bidirectional association. Reflecting the distinction between value and object types, a "field" in an EMOF class can be either an *attribute* or a *reference* (with the term *structural feature* covering both). Besides setters, a multiplicity-many structural feature has additional methods used at runtime to update a collection c. Intuitively, not all of these mutators will impact an OCL function taking c as argument. For example, adding an element impacts `c.size()` while changing the position of an item does not. In terms of EMF, method `c.add(newElem)` (for the collection in question) should be instrumented, while `c.move(fromPos, toPos)` may be left as is. We exploit this fact by analyzing at compile-time the ASTs of OCL expressions and by extending the EMF code generation process, applying techniques reported in [Gar07] and [GS07] resp.

EMOF collections may act as both explicit and implicit arguments in an OCL expression, with the former being possibly an intermediate result (computed as an OCL expression at the invoker's call site). In order to qualify as an implicit argument, a read-access must happen on a field of an application-level instance, as opposed to being performed upon an intermediate result, which stays on the heap only during a call stack activation. Intermedi-

ate results do influence the outcome of a function, however they can change only if their base data (implicit arguments, subcomputations) has changed. Therefore, tracking updates to base data is enough to signal the need for refreshing a cached computation.

Given that collection operations are common in OCL specifications, it pays off to devise dedicated optimizations beyond those in DITTO, considering their distinguishing features: (a) mutating a collection does not affect its object identity; (b) collection mutators are invoked on the collection object itself, not on some object holding the collection in one of its fields; (c) after passing a collection c as argument to a setter in a different instance, the same collection c can be obtained through getters in different instances; (d) internally, collections are realized as binary trees, linked lists, or some other data structure with fields having at-most-one multiplicity. Instead of instrumenting at this low level, we adopt a dedicated mechanism other than *(instance, field ID)* to track collection arguments.

As in DITTO we collect additional information during a function invocation, associating it to a node in the DDG (a "DDG node" from now on). Such nodes can be regarded as 5-tuples consisting of: (a) the unique identifier for the evaluated function, this ID is obtained by reflection; (b) the *self* reference to the target instance; (c) the *explicit arguments* passed by the invoker; (d) the data locations of *implicit arguments*, resulting from read accesses to structural features performed during an execution of the instrumented function (but not by its callees); and (e) *subcomputations*, represented as references to DDG nodes. Which particular inputs of kinds (d) and (e) are accessed during a given activation can be traced back ultimately to (b) and (c) values, as dictated by the logic of the invoked function. The same implicit argument may show up in different DDG nodes, therefore mappings are used to point from them into DDG nodes, thus accelerating lookup. These mappings, C_{map} and F_{map}, for collection and non-collection arguments resp., are needed during the update phase of a transaction (Sec. 4.1).

DITTO builds DDG nodes for OCL collection operations by instrumenting the accesses to single-valued fields to add the *(instance, field ID)* pairs to the list of implicit arguments in the current DDG node. Whenever a subcomputation is invoked, the DDG is searched using as key the triple *(self, function ID, explicit arguments)*. Collections acting as explicit arguments introduce a large overhead, as DDG lookups now involve testing for value-equality the actual and recorded arguments. We avoid this penalty by instantiating a new DDG node only for OCL functions with no collection arguments. The instrumented versions of these functions still track accesses to structural features and subcomputations, only that they add them to the DDG node at the top of a thread-local stack (D_{stack}, used during transaction updates, Sec. 3.1). Such node stands for the caller of the current function (there is always one caller, as all incrementalizable OCL constraints have no arguments at all).

The described design decision (not having dedicated DDG nodes for some functions) is a departure from traditional memoization, grounded on the observation that DDG nodes serve two different purposes: (a) making readily available a return value, thus saving time; and (b) tracking dependencies, as needed to detect when refreshing should be performed (thus assuring correctness). Our scheme still achieves (b), as implicit arguments and subcomputations *are* collected (in the DDG node for the invoker). On the downside, *some* function invocations that *could* have been found in the DDG (those for OCL-defined operations with coll-args) will result in recomputation instead of memoization.

Similarly, *collection-valued* functions also involve a time-space tradeoff. Of the OCL incrementalizable functions, only derived attributes and operations may return collections. Given that the lookup of DDG nodes for this kind of functions will be fast (as they lack collection arguments) we adopt the decision to have their instrumented version *create* a new DDG node, caching their result using strong references. Without memoization, the lifetime of some of these results would have been limited to single call-stack activations (i.e., strictly temporary results). Our design makes them outlive such invocations, albeit increasing the memory footprint. In the particular case of derived attributes and no-args operations, the chosen scheme was the only sensible, as these OCL constraints are interpreted as defining *materialized views*. Additionally, caching a collection return value still results in an equality test between collections, after the DDG node had been marked dirty and recomputed (the test determines whether the new value should percolate up the invocation hierarchy). However, this expensive comparison occurs less frequently as compared to DDG lookup. For these reasons, we believe the pros of memoizing coll-valued functions outweigh on average the cons, although contrived cases can be devised where the opposite is the case.

3 Incrementalization algorithm: Compile-time activities

The OCL-defined functions subject to instrumentation are determined by transitive closure over the caller-callee relationship, taking as starting point the union of (a) class invariants, (b) derived attributes, (c) and derived no-args operations. Whenever an operation is added to this set, all its override-compatible operations in subclasses are also added. We call the resulting set *Instr*. The declaration of each function in *Instr* can be uniquely identified at runtime, as EMOF reflection assigns compile-time IDs that can be woven into the generated instrumentation code (e.g., into the code to look up DDG nodes).

3.1 DDG lookup

The first Java statements generated for an OCL function *f* in *Instr* assign to the local variable *cDN* the current DDG node, if any. This lookup is performed differently depending on whether *f* has one or more collection arguments or not:

1. *f* has one or more coll-args. As no dedicated DDG nodes are kept for *f*, the return result must be computed afresh (however, calls performed by *f* may be resolved in the DDG). The current DDG node is peeked from D_{stack} (i.e. read but not popped).
2. otherwise, a lookup using *(self, f's ID, explicit arguments)* is performed against the globally-shared DDG, with one of two outcomes:
   (a) if found, and the node is not dirty, the cached return value is returned to the caller. Otherwise the function will be re-evaluated, which implies clearing the dirty bit, the implicit arguments, and the subcomputations in the found node.

(b) if not found, this is the first invocation for the triple in question. A new DDG node $newDN$ is instantiated, assigned to *cDN*, with its sets of implicit arguments and subcomputations initially empty. Before adding $newDN$ to the set of subcomputations of the caller node (i.e., the node, if any, at the top of D_{stack}), an optional check can be made whether doing so would establish a cycle in the DDG, thus preventing some cases of stack overflow (but not *runaway recursion* where explicit arguments are different for all invocations), a safety measure not present in from-scratch recomputation nor in DITTO.

Finally, for functions in *Instr* lacking collection arguments (cases 2.a and 2.b), the generated code pushes *cDN* into the thread-local stack D_{stack}, and pops it just before returning.

3.2 Implicit arguments and their setters

Executions of the generated code having reached thus far can rely on a current DDG node, reachable via the non-null *cDN*. The instrumentation code must abide by the evaluation semantics of OCL constructs. For example, the condition part of an `if-then-else-endif` is evaluated first, depending on which one of the two other branches will *not* be evaluated. Correspondingly, only the inputs (implicit arguments and subcomputations) for the evaluated branch are to be added to *cDN*. This is achieved by choosing the order to visit subnodes of an AST subtree according to the OCL construct in question. In the `if-then-else-endif` example, a visitor for code generation will visit first the condition part, generating code that at runtime will leave the result in a temporary local variable `condPartResult`. A Java "`if (condPartResult)` $\{s_1\}$ `else` $\{s_2\}$" is generated next, with the statement blocks s_1, s_2 resulting from visiting the `then` and `else` subnodes of the OCL `if` subtree.

The above code generation scheme accomodates the injection of statements to store references to implicit arguments and to subcomputations just before they are accessed. For example, when visiting an OCL `PropertyCallExp` AST node (which stands for a field read-access) code to capture the target instance and the field declaration is generated. Such code will add at runtime an entry with that key to F_{map} (one of the two *implicit args* $\rightarrow$ *DDG nodes* maps, the other being C_{map} for collection mutators). The receiver of the getter indicated by the `PropertyCallExp` will never be a temporary object: no OCL construct results in objects being instantied by generated Java code, and thus must be application-level, possibly referenced through a local variable or an explicit argument (in contrast, temporary collections can be instantiated). Moreover, the receiver object is not a collection, as only method calls can be performed on OCL collections (represented by `OperationCallExp` AST nodes). The accessed field may have multiplicity > 1. What code (if any) is generated to instrument collection mutators other than setters is the topic of the next subsection. The general rule that no derived results are tracked, but instead updates to their base data, can thus be seen at play for field accesses. After all functions in *Instr* have been visited, the set of structural features that may influence their results is known, and their setters look up DDG nodes at runtime as described in Sec. 4.1.

3.3 Operations on collections and their mutator methods

DITTO considers no mutators other than field setters. If left uninstrumented, changes performed through collection mutators (`add(newElem)`, `setItem(pos, elem)`, etc.) will go unnoticed to the incrementalization infrastructure (intercepting these mutators is the counterpart to the reduction in implicit arguments achieved by tracking collection objects instead of their items). Instead of flatly instrumenting *all* collection mutators, the generated code will be qualified to monitor *certain* mutators, depending on the function taking the collection as argument. This function must be one in the OCL Standard Library, as all user-defined functions fall under the "subcomputations" category (Sec. 3.4), in particular those with one or more coll-args.

For incrementalization purposes, the OCL built-in functions taking (one or more) collection arguments can be classified into: (a) those accessing each item in the collection; and (b) those aggregating a result. All iterator constructs ($source$`->forAll(`$boolCond$`)`, $source$`->select(`$boolCond$`)`, in general all subtypes of `LoopExp` in the OCL metamodel) fall into the first category, while $source$`->isEmpty()`, and $source$`->first()` are examples of the second category. The $source$ fragment stands for a collection-typed subexpression providing an argument for the function following the `->`.

The analysis to determine the subset of collection mutators that triggers re-evaluation also takes into account the most specific type of the source collection. For example, $source$`->collect( e |` $exprOnE$`)` maps $exprOnE$ to each `e` item in $source$. Given that OCL is strongly typed, it can in general be known at compile time whether $source$ is (a) set or bag, or (b) sequence or ordered set. In the first case, the result of the `collect()` is invariant under reorderings of $source$. Therefore, `move(from, to)` is *not* among the mutators to watch for when visiting the subtree for $source$ in the AST. Once a field access is reached in the course of that visit (i.e., a `PropertyCallExp` subtree is reached), the generated instrumentation code will *not* trigger a false-positive upon invocation of `move(from, to)` on the source of the `PropertyCallExp`, which may itself be an ordered collection, as for example the field holding chronologically ordered `publications` in class `Researcher` in the expression `self.publications->asSet()->collect(p | p.authors->size() )->max()` that finds the largest number of co-authors.

3.4 Subcomputations

As for subcomputations, after generating instrumentation code for operations on collections (as per the previous subsection) the only `OperationCallExp` subtrees not yet translated are those standing for invocations to operations defined by the user using OCL. No special code is needed at the caller site other than the usual invocation, as the current DDG node has already been pushed onto D_{stack}, and the lookup of a DDG node for the callee (if any) is performed by the callee itself.

An error scenario to avoid is for a function f_1 in *Instr* to invoke a non-instrumented function f_2, as f_2's execution would not leave a trail of its dependencies, with the incremental-

ization infrastructure later not being able to properly react to changes in f_2's inputs. This failure scenario is ruled out by the construction procedure of *Instr* (transitive closure over the static caller-callee relationship, including override-compatible methods in subclasses). Moreover, there are no "volatile" functions in OCL (i.e. functions that return a fresh value on each invocation, such as `System.currentTimeMillis()` or `RAND()`), thus reducing the amount of dirty DDG nodes that would otherwise require recomputation.

4 Incrementalization algorithm: Runtime activities, Consequences

4.1 Update Phase of a Transaction

During the compile-time phase described in Sec. 3.2, all setters potentially affecting a function in *Instr* have been instrumented. At runtime, each such setter looks up in the globally-shared F_{map} zero or more DDG nodes, using *(instance, field ID)* as key, and marks each found node as dirty. This step is no different from DITTO's, save the implementation technique (code generation in EMOF vs. Java bytecode instrumentation in DITTO). The callers of the found DDG nodes are not yet marked as stale, because the assumption that their return values will prevail is going to be validated at the time mispredictions are detected and resolved. The previous value of a field was not stored in the F_{map} entry, therefore any setter invocation (even those leaving the same value as-is) results in one or more DDG nodes being marked dirty. Again, a time-space tradeoff.

The code generated for collection mutators uses as key *(collection, mutator ID)* to look up zero or more DDG nodes in the globally-shared C_{map}. This map is populated by the code generated as per Sec. 3.3. Both F_{map} and C_{map} are implemented with Java's `WeakHashMap`s, so as not to interfere with the normal garbage collection of application-level objects when becoming unreachable from other application-level objects.

4.2 Commit Phase Activities

Transaction commit involves four phases: (c.1) pruning DDG nodes with any garbage collected input; (c.2) invoking computation of incrementalizable functions for new objects; (c.3) refreshing dirty DDG nodes; and (c.4) handling mispredictions.

Phase (c.1) *Pruning*. As updates are performed by application-level code, application-level objects being tracked as (explicit or implicit) arguments may be garbage collected. Removing their entries from F_{map} and C_{map} is taken care of by the `WeakHashMap` infrastructure, but the DDG nodes these entries target must be explicitly pruned (they might be referenced from caller DDG nodes, thus GC alone will not do the trick). Pruning a node also results in flagging as dirty all nodes directly depending on it. Transitively dependent nodes however are not yet considered as stale because the assumption that their return values will prevail is going to be validated at the time mispredictions are detected and resolved. Pruning a node may leave some of its callees unreachable over the subcom-

putations relation (this may also happen as a consequence of refreshing in phase c.3). Such nodes may be kept in a dedicated, DDG-owned set to preven their GC (with the expectation of later use) or traded for memory right away. In the latter case, recomputation of dirty nodes will repopulate the DDG with those subcomputations not found by memoization.

Phase (c.2) *Incrementalizing functions for new objects*. OCL's `allInstances()` are tracked using the AspectJ-based mechanism of [WPN06], which reports the instantiations made after the last run of commit-phase. On those instances, their instrumented invariants, derived attributes, and derived no-args operations are invoked for the first time.

Phase (c.3) *Refreshing dirty nodes* The optimal ordering to refresh dirty nodes is breadth-first over the subcomputations relation as shown in [SB07]: assume *f(x)* and *g(y)* need refreshing, with *g(y)* a transitive callee of *f(x)*. Upon replaying *f(x)* it may well be the case that *g(y)* is not invoked anymore (neither directly nor transitively through *f(x)*'s callees). Breadth-first search will thus not reach *g(y)*. As with previous callees not used anymore as subcomputations (as determined in the pruning phase), if *g(x)* is not a top-level invariant, its unreachable DDG node may be left unpruned to survive a number of incrementalization rounds, or traded for memory right away.

Phase (c.4) *Handling mispredictions*. After the refresh phase, some nodes are marked as having a return value different from that previously cached. The callers of such nodes are then suspect, as their own return values need to be corroborated. Bottom-up refreshing (from callees to callers) proceeds until (a) a node is reached where the cached and newly computed return values match; or (b) a root node is reached (a node for a top-level invariant, derived attribute, or derived no-args operation). Bottom-up walking always terminates (there may be several callers for the same DDG node, but the DDG is acyclic).

By now, all of the incrementalizable functions have up-to-date values for all application-level, not garbage collected objects.

4.3 Consequences of the design choices made

The original description of optimistic memoization [SB07] restricts the usage of return values from subcomputations by forbidding passing them as explicit arguments in further subcomputations or using them in loop conditionals. This conservative measure is motivated by the real danger that a mispredicted return value could lead to infinite recursion or an exception being thrown in the memoized version, while from-scratch recomputation would have terminated normally. The termination behavior of OCL expressions (Sec. 2.2) allows relaxing this restriction, by forbidding only recursive invocations from taking as explicit arguments the results of previous subcomputations (all other callees terminate, in particular all functions in the OCL Standard Library). If this less restrictive ban is also lifted, only those OCL-defined functions that would have looped forever in from-scratch recomputation will not terminate when evaluated by optimistic memoization (even with cycle detection in the DDG). In this sense, the chosen incrementalization technique is as robust as the base case.

Incrementalization is oblivious to the particular way a function is computed, thus provid-

ing leeway at compile time in choosing a particular implementation. For example, OCL provides no dedicated syntax for expressing equijoins, with a custom function being usually defined to encapsulate the rather awkward building of cartesian product and selection. In the long run, OCL should be extended with query constructs as found in LINQ (*Language INtegrated Query*, [MBT07]). In the meantime, the product-selection pattern can be detected at compile-time, to generate instead an instrumented version using indexes as described in [WPN06]. Another optimization involves "small functions", e.g. functions having only explicit arguments and lacking both implicit arguments and callees: their DDG nodes are terminal and computing them anew is faster than memoizing them. A visitor can be used to determine the average complexity of an OCL expression [Gar07] to choose these functions. Incidentally, the Java implementation of some functions in the OCL Standard Library (e.g., `size()`) already incrementalize their computation.

5 Future Work: Shared-memory Transactions

Simon Peyton Jones reviews in [Jon07] the perils associated to the explicit-locking programming model: (a) taking too few locks opens the door to data structure corruption; (b) taking too many locks may inhibit concurrency or cause deadlock; (c) the language does not preclude taking the wrong locks, as the connection "which data is guarded by which locks for which operations" exists only in the mind of the programmer; (d) taking locks in a wrong order eventually causes deadlock; (e) performing error recovery once data structure corruption has taken place is extremely tricky; among others. Another shortcoming is the requirement to know the internals of individually atomic operations, if they are to be bracketed into a composite transaction: the set of required locks may be data-dependent and thus known only at runtime.

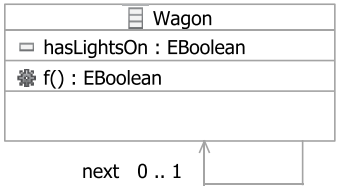
We aim to adapt and integrate into our compilation algorithm the techniques for shared-memory transactions presented in [HG06]. The adaptations involve subsetting (as the EMOF object model lacks arrays, native code, built-in classes, and static fields, methods, and initializers) as well as new developments. We aim at doing away with the source-to-source translation step (from AtomJava into Java) and extend instead the Eclipse EMF API with calls for transaction bracketing.

We believe incrementalization can coexist as-is with memory transactions. For example, a rollback never turns objects made unreachable back into reachable (instead, the log prevents them being GCed until successful commit). Therefore, prunning of DDG nodes need not be undone. A rollback may however restore implicit arguments back to their values at transaction start. In between, transaction progress marked DDG nodes dirty, which is not undone as the ensuing redundant recomputation will not deliver a wrong result.

Grossman [Gro06] establishes an analogy between mechanisms for shared-memory transactions and those for garbage collection: both began being manually applied, later compiled as source-to-source translations, to finally become part of the runtime system (JVM for garbage collection, with memory transactions still waiting to reach that stage).

6 Related Work

The efficient evaluation of OCL invariants is also the goal of Altenhofen [AHK06] and Cabot [CT06], where a methodology is presented to determine at compile-time the navigation paths from an updated *(instance, field ID)* back to objects with one ore more invariants depending on it. This analysis is only possible for non-recursive OCL expressions, as illustrated in Figure 3: a forward-only list of `Wagons` is constrained by invariant `lastWagonHasLightsOn`, which is fulfilled for a train as long its last wagon has the lights on. Our incrementalization mechanism can handle the addition, deletion, or update of `Wagons` anywhere in the list, by updating the topology of the DDG. Computing a reverse navigation path from the last to the first element in the list would instead require unwinding the call hierarchy for a particular execution trace, which is known only at runtime. Besides providing a detailed account of our algorithm and a termination analysis, our work also differs from [AHK06] in that we reduce the number of DDG nodes marked dirty by instrumented collection mutators. Once an invariant is recomputed as per [AHK06], it is done from-scratch: there's no memoization cache to hit, and therefore no materialized views can be maintained.




```
context Wagon
inv     lastWagonHasLightsOn : f()

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

Figure 3: An example where data accesses traverse links known only at runtime

Discrimination networks [HBC02] have been proposed for active databases as a generalization of the original Rete algorithm [For90]. Broadly speaking, there is a correspondence between the distinguishing feature of Rete networks (storing materializations of partial rule bodies, which are shared among all the activation conditions where they appear) and the sharing of subcomputations in optimistic memoization (which also skips recomputation for successful DDG lookups). Our implementation does not yet detect duplicate OCL fragments as in Rete: two invariants with the same body but different names result in duplicate DDG nodes. This is an area for improvement, made difficult by the fact that OCL expressions are many-form (different syntactical expressions for the same function). Other than that, we believe that a derivative of the Rete algorithm handling full EMOF + OCL would strongly resemble our proposal, modulo terminology.

Rete-based rule-engines employ a proprietary query language to express the activation conditions of production rules, and force the programmer to specify a subset of the object population to monitor for updates (the *working set*). The semantics of OCL class invariants call instead for tracking all instances. A compile-time analysis of the ASTs of such expressions limits runtime instrumentation overhead to only those object-fields participating in *some* OCL incrementalized expression. The alternative (monitoring all

updates) has the potential advantage of allowing incrementalizing ad-hoc expressions at runtime. As for invariants (an important use case), this is not reasonable: invariants do not come and go. The best of both worlds (low monitoring overhead and incrementalization of ad-hoc expressions) could be achieved with instrumentation techniques that allow for (un-)deploying interceptors at runtime, such as the debug API (the Java Virtual Machine Tool Interface, JVMTI). We follow instead a more mainstream approach by extending a compilation algorithm for EMOF + OCL, using Eclipse technologies.

Algorithms for efficient recalculation of spreadsheets are reviewed by Sestoft [Ses06]. Automatically checking invariants at transaction boundaries is addressed for Concurrent Haskell by Harris et.al. in [HJ06].

7 Conclusions

The acceptance of EMOF as the mainstream approach to metamodeling has spurred a number of innovations in the tooling for authoring DSLs, most of them leveraging the Eclipse EMF implementation. The well-defined semantics of EMOF allows adding orthogonal capabilities to EMOF-enabled runtime environments (e.g, transparent persistence, change notification, versioning, refactoring support), thus increasing the appeal of the EMOF object model. We expect support for incrementalization, tracking of invariants, and shared-memory transactions to be generally useful across a variety of domains.

References

[AFP03] M. Akhtar Ali, Alvaro A. A. Fernandes, and Norman W. Paton. MOVIE: an incremental maintenance system for materialized object views. *Data Knowl. Eng.*, 47(2):131–166, 2003.

[AHK06] Michael Altenhofen, Thomas Hettel, and Stefan Kusterer. OCL Support in an Industrial Environment. In Birgith Demuth, Dan Chiorean, Martin Gogolla, and Jos Warmer, editors, *OCL for (Meta-)Models in Multiple Application Domains*, pages 126–139, Dresden, 2006. University Dresden. `http://st.inf.tu-dresden.de/OCLApps2006/topic/acceptedPapers/03_Altenhofen_OCLSupport.pdf`.

[CT06] Jordi Cabot and Ernest Teniente. Incremental Evaluation of OCL Constraints. In Eric Dubois and Klaus Pohl, editors, *CAiSE*, volume 4001 of *LNCS*, pages 81–95. Springer, 2006. Project homepage `http://www.lsi.upc.edu/~jcabot/research/IncrementalOCL/index.html`.

[For90] Charles L. Forgy. Rete: a Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Expert systems: a software methodology for modern applications*, pages 324–341, 1990.

[Gar07] Miguel Garcia. How to process OCL Abstract Syntax Trees, Eclipse Technical Article, 2007. `http://www.eclipse.org/articles/article.php?file=Article-HowToProcessOCLAbstractSyntaxTrees/index.html`.

[Gro06] Dan Grossman. Software Transactions are to Concurrency as Garbage Collection is to Memory Management. Technical Report 2006-04-01, University of Washington Department of Computer Science and Engineering, April 2006. http://www.cs.washington.edu/homes/djg/papers/atomic_analogy.pdf.

[GS07] Miguel Garcia and A. Jibran Shidqie. OCL Compiler for EMF. In *Eclipse Modeling Symposium at Eclipse Summit Europe 2007, Stuttgart, Germany*, 2007. http://www.sts.tu-harburg.de/~mi.garcia/pubs/2007/ese/oclcompiler.pdf.

[HBC02] E. N. Hanson, S. Bodagala, and U. Chadaga. Trigger Condition Testing and View Maintenance Using Optimized Discrimination Networks. *IEEE Transactions on Knowledge and Data Engineering*, 14(2):261–280, 2002.

[HG06] Benjamin Hindman and Dan Grossman. Strong Atomicity for Java Without Virtual-Machine Support. Technical Report 2006-05-01, Dept. of Computer Science and Engineering, University of Washington, Seattle, WA, USA, May 2006. http://www.cs.washington.edu/homes/djg/papers/atomjava_tr_may06.pdf.

[HJ06] Tim Harris and Simon P. Jones. Transactional memory with data invariants. In *First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2006. http://research.microsoft.com/users/simonpj/papers/stm/stm-invariants.pdf.

[Jon07] Simon Peyton Jones. *Beautiful Code: Leading Programmers Explain How They Think*, chapter 24. Theory in Practice. O'Reilly Media, Inc., 2007. http://research.microsoft.com/~simonpj/papers/stm/beautiful.pdf.

[Kya05] Marcel Kyas. An extended type-system for OCL supporting templates and transformations. In *M. Steffen and Gianluigi Zavattaro (Eds), Formal Methods for Open Object-Based Distributed Systems (FMOODS 2005), Lecture Notes in Computer Science, number 3535*, pages 83–98. Springer-Verlag, 2005.

[MBT07] Erik Meijer, Gavin Bierman, and Mads Torgersen. Lost in Translation: Formalizing proposed extensions to C#. In *OOPSLA*, 2007. http://research.microsoft.com/users/emeijer/Papers/oopslafp017-bierman.pdf.

[Obj06a] Object Management Group. Meta Object Facility (MOF) Core Specification, formal/06-01-01, http://www.omg.org/docs/formal/06-01-01.pdf, Jan 2006.

[Obj06b] Object Management Group. OMG OCL Specification v2.0, formal/2006-05-01, May 2006. http://www.omg.org/technology/documents/formal/ocl.htm.

[SB07] Ajeet Shankar and Rastislav Bodík. DITTO: Automatic incrementalization of data structure invariant checks (in Java). In *PLDI '07*, pages 310–319, New York, NY, USA, 2007. ACM Press. http://www.cs.berkeley.edu/~aj/cs/ditto/.

[Ses06] Peter Sestoft. A Spreadsheet Core Implementation in C#. Technical Report TR-2006-91, IT University, Copenhagen, Denmark, Sept. 2006. http://www.itu.dk/people/sestoft/corecalc/ITU-TR-2006-91.pdf.

[WK03] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley, Boston, MA, USA, 2003. ISBN 0321179366.

[WPN06] Darren Willis, David J. Pearce, and James Noble. Efficient Object Querying for Java. In Dave Thomas, editor, *ECOOP*, volume 4067 of *LNCS*, pages 28–49. Springer, 2006. http://www.mcs.vuw.ac.nz/~djp/files/WPN_ECOOP06.ps.