Towards Optimal Runtime Integrity Checking of OCL

Gil Kaspi, Mira Balaban

Information Systems Engineering Department, Computer Science Department
Ben-Gurion University of the Negev, Beer-Sheva 84105, ISRAEL
gilk@bgu.ac.il, mira@cs.bgu.ac.il

Keywords: UML, OCL, integrity checking, runtime checking, model-driven architecture

1 Introduction

Model Driven Engineering (MDE) approaches aim at promoting the role of models in the process of software evolution. When a class diagram is transformed to code, the OCL [1] class invariants are transformed into a runtime Integrity Checking (*IC*) mechanism. This is one of very few transformations, where static declarative information is translated into a dynamic, behavioral procedure. This *IC* mechanism faces decisions concerning *completeness* and *being focused*. Completeness means that following a runtime event *e*, the *IC* mechanism detects and handles all invariant violations caused by *e*. Being focused means that all invariants that are relevant to *e*, i.e., might be violated by *e*, are detected, and only these invariants are checked on objects that might have been affected by *e*. The aim of a class invariant transformation is to obtain optimality: Create a complete and focused *IC* mechanism.

Another problem of invariant translation in a class diagram to code transformation involves problems of mismatch between the class models at the different levels. For example, invariants on the class diagram level recognize associations, while there are no associations on the code level.

A Runtime Integrity Checking Service (*RICS*) is triggered following a change event, which might affect the consistency status of a problem instance. The task of the *RICS* is to detect situations where the initial consistency status is violated, and pass the results to an integrity handling mechanism.

We define the *completeness* and *being focused* properties of procedures for integrity checking of class invariants. Known *IC* mechanisms are either incomplete or demonstrate some form of redundancy (not focused). In this article we describe an implemented optimal (complete and focused) runtime integrity checking procedure. The implementation is built on top of the EMF – uses the tools for Class Diagram to Java transformation and for OCL evaluation, and uses the tool of Cabot and Teniente [2] for analysis of the correlation between OCL class invariants and runtime events.

2 Properties of Runtime Integrity Checking Services

The Completeness Property

The completeness property is defined with respect to awareness to violations. A *complete RICS* is one that detects every constraint violation, following a change event. For example, the naïve approach that checks all constraints on all objects is complete. The problem is that a naïve *IC* approach might include unnecessary constraint evaluations and therefore is inefficient.

The Property of Being Focused

The aim is to achieve an *IC* mechanism that, while not losing completeness, does not check irrelevant constraints, and furthermore, knows to check relevant constraints only on relevant objects. The property of checking only relevant constraints and only on relevant objects is termed *being focused*.

The *being focused* property is defined in terms of *relevance* or *redundancy* notions (redundancy is defined as irrelevance). A constraint is relevant to an event if the event may lead to its violation, and is redundant otherwise. A *RICS* is focused if it avoids checking any form of redundancy.

Static Focus: Static Focus concerns event-kinds and constraints. A constraint *Const* is *redundant* with respect to an even-kind *eKind*, if in every legal instance *I*, the application of an event of kind *eKind* does not invalidate *Const* in *I*. *Const* is *relevant* with respect to *eKind*, otherwise.

Dynamic Focus: Constraints are evaluated on objects of their context classes. A *constraint-instance* is a pair *<Const, o>* of a constraint *Const* and an object *o* from the context class of *Const*. We define a constraint-instance *<Const, o_2>* to be *redundant with respect to an event e(o_1)*(event *e* applied to object o_1), if *<Const, o_2>* is not be violated by *e(o_1)*. Following an event, it is possible that a constraint is violated when evaluated on one object, while still holding on another. Therefore, the dynamic notions of relevance and redundancy introduce finer relations between constraint-instances and events.

A finer form of redundancy, termed *Evaluation Focus*, involves the objects that are accessed during an evaluation of a constraint-instance. For that purpose we observe partial evaluations of constraints. A partial constraint-instance evaluation is *satisfying* if it correctly determines the value of the constraint-instance. It reminds partial evaluation in optimizing compilers. We define an object o_3 to be *redundant with respect to <Const, o_2> and e(o_1)*, if there exists a satisfying partial evaluation of *<Const, o_2>* following *e(o_1)*, that does not invoke o_3. Evaluation focus is a finer notion than plain dynamic focus since it is possible that *<const, o>* is relevant with respect to *e*, and still have an object *o'* which is redundant with respect to *<const, o>* and *e*.

Finally, a *RICS* is *focused* if for every event, the evaluation of relevant constraint-instances does not lead to evaluation redundancy. A *RICS* is *optimal* if it is complete and focused.

3 Towards an Optimal Runtime Integrity Checking Procedure

The *IC* procedure is based on the notions of *scope of an event*, and *derived constraints*. The scope of an event *e(o)* is the set of all constraint instances that are not redundant with respect to *e(o)*. The problem is that *scope(e(o))* might include a constraint instance *<Const, o'>* on a different object *o'*. Moreover, the evaluation of such constraint instances might include evaluation redundancy. The idea of an optimal *IC* procedure is to replace a *<Const, o'>* in *scope(e(o))*, by a *derived constraint instance <Const', o>* such that: *1)* It is defined on the object of *e; 2)* It can correctly replace *<Const, o'>*; and *3)* It reduces evaluation redundancy. A *derivedScope(e(o))* includes all derived constraint instances for constraint instances in *scope(e(o))*. The procedure IC_{derive} below is complete, since a derived scope includes all and only relevant constraint instances, and has increased evaluation focus. If *derivedScope(e(o))* guarantees evaluation focus, the procedure is optimal.

IC_{derive}(e(o)) =
  For every <Const', o> in **derivedScope(e(o)):** Const'(o)

In order to compute *derivedScope(e(o))* we rely on the static analysis of event kinds and derived constraints, of Cabot and Teniente [2]. For each OCL invariant in a model, they analyze the relevant event kinds, and associate them with derived constraints. We define *staticScope(eKind)* as the set of all derived constraints in their analysis. We show that at runtime, for *e(o)* of kind *eKind*, the set of all *<Const, o>* such that *Const* is in *staticScope(eKind)* is a good *derivedScope(e(o))* since it guarantees completeness and correctness of IC_{derive}, and reduces evaluation redundancy.

For a *captureEvent(e(o_1))* method and an $IC_{derived}$*(e(o_1))* method, if *captureEvent(e(o_1))* captures all events, and *IC* is optimal, than the resulting *RICS* is optimal.

4 Architecture of a Towards Optimal Runtime Integrity Checking Service

The implemented *RICS* is based on the IC_{derive} described above, where the runtime derived scope of an event is constructed from a static repository of derived constraints, indexed by event kinds. Its optimality depends on the evaluation focus of the constraints in the repository, and on the completeness of a procedure for capturing events. Its architecture is described in Fig. 1, and runtime interaction between the components is described in Fig. 2.

Fig. 1. Runtime Integrity Checking Service – architecture

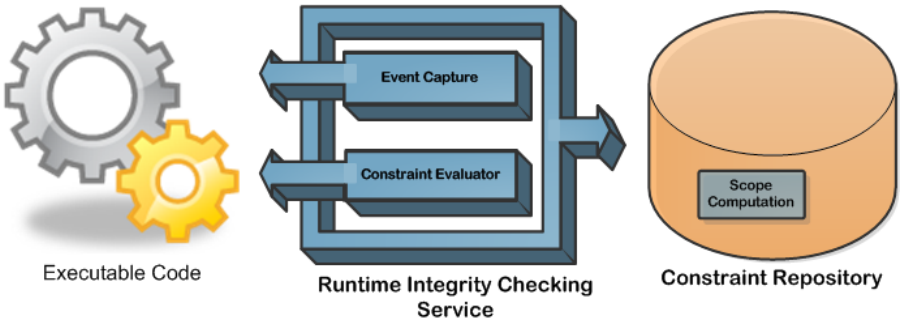
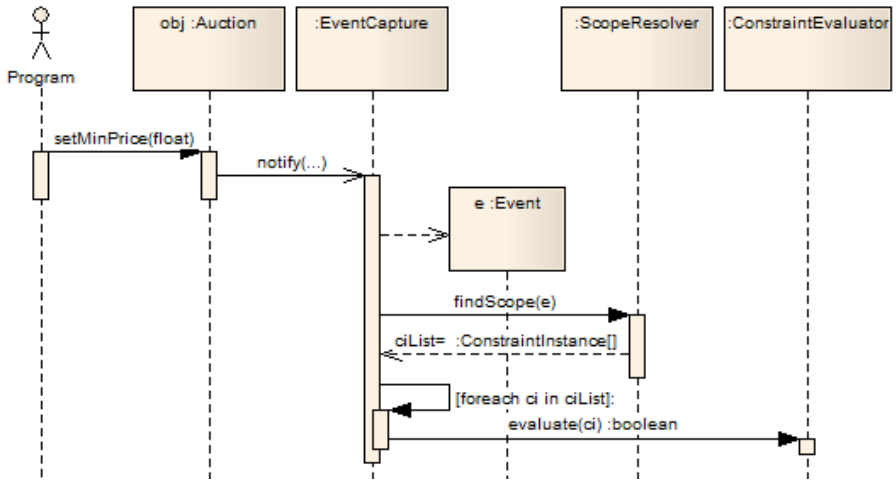


Fig. 2. Runtime Integrity Checking Service – sequence



The advantage of this architecture is the flexibility it offers for changes: *1)* The content of the constraint repository can be modified without any code changes; *2)* The constraint repository is agnostics to the runtime implementation; *3)* Easy adaptation to different executable code.

The procedure is implemented within the EMF, using the tool of Cabot and Teniente. We plan to further improve our implementation so that it becomes framework independent, and generalize the *RICS* analysis to apply to other constraint languages

References

[1] OMG, "OCL 2.0 Specification," 2006.
[2] J. Cabot and E. Teniente, "Incremental Evaluation of OCL Constraints," *Advanced Information Systems Engineering*, 2006, pp. 81-95.