

Reasoning control in presence of dynamic classes [†]

Olaf Owe
Department of Informatics
University of Oslo

[†] Talk held at NWPT'00 in Bergen, October 2000
See: <http://www.i.uib.no/~nwpt00>

Background: the project ADAPT-FT

A cooperation project between IFI, Univ. of Oslo and Institute for Energy Technology in Halden: ADAPT-FT, *Adaptation of Formal Techniques to Support the Development of Open Distributed Systems*[†]

We try to combine formal reasoning, object-orientation and openness, by designing a platform integrating:

- **UML** – for graphical interface
- **OUN** – for specification and design
- **PVS** – for performing proofs.

A code generator will generate Java code from OUN designs.

[†] supported by The Research Council of Norway under the strategic research programme for Distributed IT-Systems (DITS).

Goals

A general formalism for specification, refinement, design of, and reasoning about

- object orientation
- distribution
- openness

We desire an approach with **industrial relevance**, with **explicit support** of the main concepts above.

Specific focus of this talk

Treatment of dynamic reconfiguration by means of a **dynamic class** concept, allowing software to be changed during run-time without stopping and restarting the system.

Examples: Banking systems where services are added to users during continuous operation, telephone systems where service to customers is upgraded, etc.

How can we specify, and reason about, dynamic classes.

General setting

object orientation: classes, interfaces, objects, single or multiple inheritance, virtual binding of methods.

distribution: autonomous objects, executing in parallel, communicating by remote method calls, support of asynchronous communication

openness: Dynamic communication patterns by dynamic creation of objects, communication of object identities.

Dynamic reconfiguration by dynamic addition of software, by adding (sub-)classes and (sub-)interfaces, and by a dynamic class concept (redefining a class).

industrial relevance: by suggesting a formalism based on simple mathematics and concepts well-known to programmers, like those of UML.

Reasoning control

A formalism based on simple and well-known principles:

strong typing ensuring that run-time errors such as “method not understood” may not occur, and type-correct actual parameters and object variables.

compositional reasoning so that software units can be written and analyzed independently

textual analysis textual generation of proof obligations,

incremental reasoning control of maintenance of earlier (proven) results, additional proof obligations.

Outline

- problems caused by a dynamic class concept
 - conceptual problems
 - reasoning problems
 - specification problems
- sketch of a solution

The concept of dynamic class

Assume here that a class consists of local and inherited attributes (variables, including ordinary variables or object references) and methods, as well as a semantic specification, for instance an invariant. Consider changing a class by

- adding attributes
- adding methods
- redefining methods
- deleting methods
- deleting attributes
- adding semantic requirements
- redefining semantic requirements

When a class is redefined (at run-time), we assume that an old object of the class will get the renewed capabilities! – allowing us to dynamically change the behaviour of programs.

When a class is redefined, we assume that an old object of the class will get the renewed capabilities.

- **adding attributes:** old objects will then contain these attributes (non-trivial adjustments by the run-time system)
- **adding methods:** old objects will then support these methods
- **redefining methods:** old objects will then support these methods
- **deleting methods:** old objects will no longer support these
- **deleting attributes:** old objects will no longer contain these
- **adding semantic requirements:** these will be satisfied after the redefinition
- **redefining semantic requirements:** these will be satisfied after the redefinition, old requirements could be violated.

Problem: deletion of attributes and methods

Assume that a class C has a local attribute x and that x is deleted.

An old subclass of C may use the x inherited from C in one of its local methods. This will lead to run-time errors, and we have lost the benefits of strong typing.

May disallow such redefinition of C : but this would mean that the redefinition of a class depends on all subclasses of C — which is not desirable!

Deletion of methods leads to the same kind of problems.

Therefore, we will not allow deletion of anything. Thus, the new version of a redefined class forms of subclass of the old version. However, in contrast to a subclass of C , a redefinition of C will change the capabilities of C -objects.

Problem: addition of methods

Assume that a class C is extended with a method m which it did not have earlier.

But, an old subclass of C may have introduced a method m . The formal parameter types of the two methods may then be the same or different:

- The latter case is acceptable if we have **full overloading** (like Java).
(Differences must also be allowed in function-values/out-variables: may use overloading on these, or contra-variance.)
- The former case is OK if a subclass is allowed to **redefine methods – without semantic restrictions**.

Problem: reasoning about dynamic classes

Should we allow a redefined class to violate the old semantic requirement (its invariant)?

Yes: reasoning is difficult, as programs may change behavior suddenly!

No: We may rely on the old invariant, and any results proved by means of the old invariant. Stronger results may be obtained using the strengthened invariant.

Conclusion: The new version of a redefined class should form a “behavioral subtype” (in some sense) of the old.

Problem: reasoning about subclasses

We have allowed unrestricted redefinition of methods in subclasses.

This implies that **a subclass may violate an inherited invariant**, and we cannot impose “behavioral subtyping” on the subclass mechanism.

Reasoning about a variable x typed by C , is difficult since x may refer to an object belonging to a subclass of C which does not satisfy the invariant of C .

Problem: Multiple inheritance

Our solution

We do not allow variables to be typed by classes, but rather by **interfaces**, consisting of (unimplemented) methods and specification of observable behavior.

- A subinterface must respect inherited semantic specifications (by some suitable refinement relation).
- multiple inheritance is unproblematic for interfaces
- A variable typed by interface I may be assigned an object expression whose type is a subinterface of I , or a **new C** expression where C implements (a subinterface of) I .
- A class may claim that it implements a number of interfaces. This gives rise to proof obligations.
- A consequence is that such implements claims are not in general inherited by subclasses (since subclasses are not restricted by behavioral subtyping).
- A redefined class must respect old implements claims (since class redefinition is restricted by behavioral subtyping).
- A redefined class may implement additional interfaces (possibly newly defined ones) thereby making it possible to let old object interact through new interfaces and with new objects of unrelated classes.

Benefits

- a dynamic class concept is supported, allowing redefinition of a class during run-time, allowing old objects to adapt to new versions of software, and at the same time allowing strong typing and incremental reasoning control.

A class redefinition may add methods and attributes and redefine methods.

- forcing the use of interfaces: this gives better abstraction,
- aspect oriented programming: since interfaces allow multiple inheritance (such that one may let one interface focus on one aspect of an object).
- subclasses are not restricted by semantic considerations: this greatly improves reuse of code. Semantics requirements (including “implements clauses”) are not inherited, and may be violated.

Notice that while a redefined class must respect old semantic requirements (including “implements clauses”), a subclass may violate the semantic requirements of the superclass.

Future and ongoing work

- the OUN language (with I.Ryl and E.Johnsen)
- Semantic definition (with E.Johnsen)
- Timeouts and reasoning about time (with E.Johnsen)
- Fault tolerance (with J.Vain)
- Large example: A software buss (W.Zhang)