On Combining Reasoning Control with Distribution and Openness in Object Oriented Systems

Olaf Owe* and Isabelle Ryl†

* Department of Informatics
University of Oslo,
P.O.Box 1080 Blindern,
N-0316 Olso, Norway
Phone +47 22852449
olaf@ifi.uio.no
† L.I.F.L., Université de Lille
Bât. M3, Cité Scientifique
59655 Villeneuve d'Ascq Cedex, France
Phone +33 320336289
Isabelle.Ryl@lifl.fr

1 Introduction

We are focusing on high level design, specification and reasoning of object oriented systems supporting distribution and openness. In particular we wish to investigate how well reasoning control based on textual analysis can be combined with non-trivial aspects of openness such as dynamic reconfiguration. From an intuitive point of view, it may seem that reasoning control based on textual analysis cannot easily be combined with dynamic evolution of programs. We wish to demonstrate that these two concepts can be combined when properly restricted.

The need for dynamic reconfiguration is obvious in areas such as for instance business transaction systems, telephone switching systems and transportation control systems, where interruption of service is not desirable or acceptable. However, the need for certain guarantees is essential, for instance to maintain invariants ensuring that money is handled properly, and for safety critical applications that certain kinds of accidents are avoided. This is especially true in an environment where programs are changing dynamically.

In order to keep reasoning manageable, unrestricted generation of proof obligations at run-time has to be avoided as much as possible. We consider decomposed reasoning based on static (strong) typing, and incremental generation of verification conditions based on textual analysis of pieces of program or spec-

ification text. It is essential that already proven results are maintained when programs are changed, or guaranteed by additional verification conditions. And there should be compositional proof rules for concurrent processes. Even in case no proofs are carried out, the proof principles should serve as valuable guidelines for informal reasoning, or testing.

Most existing formal methods are not allowing the kind of object orientation and openness desired here. This includes the work on process calculi (initiated by CSP or CCS), temporal logic, UNITY, Action systems, Actors, TLA, Z, VDM, FOCUS, TROLL and so on. A more detailed assessment and comparison is given in [18]. Maude [13] is one of few object oriented formalism with great flexibility, for instance the class of an object may change from one state to the other, not even restricted by a subclass relationship. However, reasoning is rather limited, even safety reasoning is not (yet) available. The language ERLANG offers facilities for dynamic reconfiguration, but does not come with a reasoning formalism. UML based approaches such as OCL [21] and CATALYSIS [4] are to some extent discussing specification issues, but does not provide reasoning control of advanced properties such as dynamic reconfiguration.

We consider basic object oriented constructs such as interfaces (without attributes and method implementations), classes (with attributes and method implementations), virtual binding of methods (i.e. dispatched at run-time), combined with single and multiple inheritance.

And we consider a form of partial compilation, such that pieces of code can be compiled at different times and added to a running system, without restarting it. The compilation units may build on each other. A compilation unit may contain new (sub)classes, (sub)interfaces and class extensions/redefinitions. This enables us to design systems where old objects may communicate with newer objects through new interfaces (as well as old superinterfaces). Objects are considered to have internal activity, running in parallel and communicating by (remote) method invocations, thereby supporting distribution. An object may support a number of interfaces, and this number may increase dynamically.

Classes and interfaces are specified by means of invariants and assumptions in a rely-guarantee style [8]. In order to specify observable and fully abstract behavior of interfaces we use the concept of communication history (trace), similar to that of CSP [7]. Class specifications may in addition refer to local attributes. We will here neither focus on the specification language nor the formalism for reasoning. This will be done in a separate paper. Preliminary parts of the language and formal system can be found through the homepage of an associated project [1].

We will look at the challenge of how to combine openness with reasoning control based on textual reasoning. And we wish to investigate which concepts that can be combined with flexibility and generality. For instance it turns out that we must restrict dynamic class extension to behavioral subtyping, otherwise old proven results may be violated. In contrast, the subclass mechanism will not be restricted by behavioral subtyping.

2 Dynamic class extension

We assume that at run-time an object belongs to a class with fixed name; but the contents of this class may change, due to the dynamic class construct. We consider a general and flexible form of dynamic class extension/redefinition: During run-time a class may be extended with more attributes and more methods, and possibly have methods redefined, with the effect that old (as well as new) objects of the class are extended with the new attributes and may perform the new methods. A class may thereby support more and more interfaces. A dynamic class concept can be found in CLOS and Smalltalk, which are based on dynamic typing. Also languages based on static typing such Java and C++ have been extended to allow dynamic types [12, 6].

One could also consider dynamic deletion of attributes and methods, however, this may lead to violation of strong typing. For instance, the deletion of a method which is called in a subclass could cause a run-time error like "method not found". Even though deletion could potentially be useful, we will here stay within the framework of strong typing, and will not consider dynamic deletion of attributes or methods.

It is rather obvious that we must restrict dynamic class extension to behavioral subtyping, otherwise old proven results may be violated. A class B is said to be a *behavioral subtype* of A, if a B-object will behave as an A-object, consistent with the specification of possible A-behaviors in some sense. For instance, a method resulting in "true" (say for certain input) could be redefined so that its result is "false". This could clearly cause violation of old proven results, which may state that the method results in "true".

A stronger requirement would be to disallow method redefinition in dynamic class extensions, but this would disallow useful and natural class extensions. For instance consider a class Bank providing the basic functionality for transactions inserting and depositing funds to accounts. A useful extension would be one that keeps track of the transaction history. Clearly, we would need to add attributes to represent the transaction history, add methods to observe the history, and redefine methods so that insertions and deposits are recorded.

We will not here discuss implementation issues ourselves, but refer to [12] which talk about implementation of dynamic classes. Their concept of dynamic extension is in principle unrestricted, but for simplicity and efficiency they must restrict class extensions a bit, especially redefinition of methods. In order to deal with new attributes they use locks, but argue that this is reasonably efficient. Even though they have not been interested in semantic issues, it seems that class extensions representing behavioral refinement is efficiently implementable with their work, and gives us a general concept of class extension suitable for our purposes.

2.1 Inheritance versus behavioral subtyping, a conflict

When a superclass is extended, its old subclasses will inherit the new attributes and the new or redefined methods, and thus are implicitly extended as well. We

wish to allow a class extension mechanism which is general and not restricted in an inappropriate manner (such as restrictions depending on information in subclasses). A consequence is that methods should be overloaded since an extension of a superclass may introduce a method with the same name as a method in a subclass with arbitrarily different parameters. And in case the (number and types of) parameters of the two methods match, their semantics may be arbitrarily different since they may be written independently of each other. The latter means that we should allow *unrestricted redefinition of methods in subclasses*. In presence of out-parameters, full overloading is combined with virtual binding using contra-variance for matching actual parameters to formal ones.

As in Java, one may then let the static analysis of a call, say $x.m(a)$, determine the formal parameter types (based on matching against the operations offered by the static class of the expression x using the static type of the actual parameter list a), and at run-time choose the redeclaration (with the above formal parameter types) closest to the actual class of x. Full overloading is thereby supported (however, as in Java, a given call may be statically ambiguous if the (minimal) formal parameter types are not unique).

Unrestricted redefinition of methods is natural in object oriented languages and useful for reuse of code, but causes problems with respect to reasoning control, as most formalisms supporting inheritance are based on the notion of (weak or strong) behavioral subtyping [14, 11, 10, 2, 3], i.e. it is required that a subclass B is a behavioral subtype of a superclass A. This means that where you expect an A-object, but get a B-object, no surprises will result, in some sense. This is essential in order to avoid reverifying the client code when based on a superclass.

In most object oriented programming languages, it is indeed the case that a subclass object may be used as a superclass object. In the presence of virtual binding, static information about invoked methods is in general not available. Therefore behavioral subtyping is the natural way to treat subclassing in a formalism for reasoning. This means that requirement specifications of superclasses are inherited. However, this puts severe semantic restrictions on the subclass mechanism, added methods must maintain the inherited class invariant (which limits the updates of inherited attributes) and redefined methods must obey the old specification. This imposes a significant restriction to the kind of subclassing traditionally allowed in object oriented languages (from Simula to Java). Even though there are different notions of behavioral subtyping, stronger and weaker ones, it is obvious that many useful subclass definitions will be semantically illegal (see [17] for further discussion).

On one hand we have seen that subclassing respecting behavior is needed, in order to be able to reason, and on the other hand we have seen that this kind of subclassing does not go well together with the ideas of code reuse, which are essential in object oriented programming.

A partial solution to this, suggested by several authors [9, 3, 2, 17], is to distinguish between an abstract and a concrete specification of a class. Then object reasoning is governed by the abstract specifications, and behavioral subtyping is only required between the relevant abstract specifications. A concrete

specification is used to prove that a class satisfies the corresponding abstract specification. An abstract specification could for instance be disallowed to refer to state variables. However, if there is exactly one abstract specification for each class, code reuse may still be difficult in that a subclass may break the inherited abstract specification, unless it is very weak– in which case reasoning power is severely limited.

2.2 Typing by interfaces

As a solution to the problem above, we suggest the use of interfaces as follows:

- An interface contains syntactic declaration of methods (as in Java) together with semantic (rely-guarantee) requirements, specifying observable behavior, but does not contain code nor attribute declarations (i.e. variables).
- A class may implement many interfaces. The implements relationships can be stated incrementally, by special "implements-clauses", resulting in verification conditions (see more below).
- Neither requirement specifications nor implements-clauses are inherited at the class level (which means that subclassing is not restricted by behavioral subtyping).
- There are independent inheritance hierarchies for interfaces and for classes. Thus a subclass need not implement an interface even if a superclass does!
- We allow multiple inheritance of interfaces and classes. (One may of course restrict oneself to single inheritance of classes for reasons of efficiency, or when the target implementation language does so.)
- Finally we suggest that all object variables and object parameters are typed by interfaces, even at the design level! Classes can then be used to declare subclasses and to create new objects.

This simplifies reasoning, since an interface is used to describe observable behavior of a certain aspect (role) of an object. However, at run-time a given object will belong to one class, which may be extended dynamically by adding operations and thereby possibly implementing additional interfaces.

An object variable x declared of interface F must at run-time refer to a class implementing F; thus the assignment $x := e$ is legal if the static type of the expression e is F, is a subinterface of F, or e is a **new**-construct of the form

$$\textbf{new}\ C(l)$$

where C is a class implementing F, and l a list of (type correct) actual parameters. The object may know of other objects through the list of actual parameters (l) and through actual parameters of method calls made to the object throughout its life.

Aliasing of objects does not cause the well-known reasoning problems for sequential programs: Since each object may have its own activity, we may only rely on its invariant. Inside a class we assume that each method body forms a critical region for the process associated with the current object, allowing Hoare style reasoning for the local attributes, whereas any external object may change between each statement, but only such that its invariant (i.e. observable specification according to the given interface) holds.

A class is said to *correctly implement* an interface if all operations in the interface are implemented in the class and if the requirement specification of the class refines that of the interface (see [15] for a suitable definition of class refinement). When the class of an object implements several interfaces we say that the object *supports* these interfaces. For every stated implements-clause between a class and an interface, there will in general be a verification condition ensuring that the class implements the interface correctly.

The combination of the interface and class mechanisms as described above does not seem to exist in any other object oriented language offering requirement specification. The cost of allowing unrestricted redefinition of methods in classes, is that all object variables must be typed by interfaces. However, several object oriented books, including [5], encourage such use of interfaces.

One may still argue that sometimes it is necessary to know the exact class of an object. In order to compensate for this, we may allow a construct where the programmer may test if an object has a given class, and program accordingly "inside" the construct (using the given class in the static type analysis), similar to the inspect-when-do construct of Simula. As long as the test checks the exact class (not superclass), we may rely on the invariant of the given class for reasoning purposes.

Other kinds of reflection may be allowed as well: for example one may test an object to see if it supports a given interface (and program accordingly "inside" the construct). And interactive users may ask an object about what interfaces it supports and call any method in one of these.

2.3 Multiple inheritance

Whereas it is unproblematic to allow multiple inheritance for subinterfaces, it is less so for subclasses, in presence of a dynamic class construct. For an object o of a class A, a method invocation, say $o.m(..)$, may lead to execution of a method inherited from a superclass B. However, a dynamic extension of another superclass C may introduce another method m (with the same kind of parameters), which obviously need not be consistent with the invariant of A. (Note that if C is a subclass of B the added version of m would need to respect the invariant of B, since the extension must be a behavioral subtype).

Let us assume that inheritance is formed by means of disjoint union allowing several methods with the same name and parameter types to be inherited. For simplicity we assume the same for attributes with the same name and type. This means that extensions made to superclasses will not cause already existing subclasses defined by multiple inheritance to be meaningless. Calls of methods

that are not uniquely determined by name and parameter types (at compilation time), may be qualified by a superclass name (indicating the appropriate superclass hierarchy) in order to be syntactically legal.

Assume first that a class (with multiple inheritance) implementing an interface is such that the interface methods are unique in the class, at least at the time of the analysis of this property (later superclass extensions may violate the uniqueness). In order to allow incremental reasoning, we would require a dispatch mechanism which first finds the older – in terms of compilation date – method implementation (i.e. the oldest matching method implementation of a superclass, for given name and parameter types), and then bind the invocation to this implementation, unless there is a redefinition in the subclass path down to the class of o, in which case the redefinition closest to the class of o is taken (as usual with virtual binding). This seems to be reasonably easy to implement, with little overhead.

If a class (with multiple inheritance) implementing an interface is *not* such that the interface methods are unique in the class, we would need specific information about in which superclass hierarchies to find the different interface methods. For instance, a natural way to implement the method invocation $o.m(..)$ would be to include information about the interface of o in the translated code, and then at run-time bind to the closest implementation of m inside the specified superclass hierarchy of the class of o (unless m is implemented in the class itself). For each interface supported by the class, the information about which superclass hierarchy each method belongs to can easily be updated incrementally.

For initializing code (constructors) one may as default perform those of the superclasses, in the order given in the multiple inheritance list. However, when this is not desirable, one may redefine the initializers/constructors (explicitly reusing those of superclasses).

3 Conclusions and future work

We have motivated a notation based on general principles known from formal methods, combined with the essential object oriented concepts, and with more flexibility and more dynamic considerations than existing formal methods. We insist on static typing, which means that the software will be reliable in the sense that type errors will not occur, and method calls to remote objects will always be syntactically correct.

It has been essential to be able to combine static typing with non-trivial dynamic behavior: Objects, interfaces and classes may be added dynamically, and old and new objects may communicate by means of new interfaces (as well as old ones).

Object variables and object parameters are typed by interfaces (rather than classes) which not only helps reasoning, but makes software more reusable, more abstract (disallowing write-access to remote variables) and more understandable, and is essential in order to allow and control dynamic behavior.

Subinterfaces are inheriting semantical constraints while subclasses do not.

At the class level, we allow unrestricted redefinition of operations, possibly violating inherited invariants. This opens up for flexible reuse of code, as argued in [17], and gives the same notion of subclassing as for instance in Java. Since reasoning and typing are based on interfaces, and since subinterfaces must respect interface refinement, already proven verification conditions cannot be violated by adding subclasses and redefining operations.

We consider a form of dynamic class extension, which enables us to extend a class dynamically, respecting inherited invariants. Together with dynamic creation of objects and addition of interfaces, this allows non-trivial dynamic behavior. The class extension mechanism may be seen as a restricted version of capabilities in CORBA and Java RMI, staying inside the framework of static typing.

As the class extension mechanism has no other restrictions than that of behavioral subtyping, we have seen that a consequence is that redefinition in subclasses must be semantically unrestricted, and operations must be overloaded. Thus the object oriented concepts of our proposal are both orthogonal (i.e. can be used with great flexibility) and are well integrated. Reasoning control is achieved by the generation of verification conditions for each program unit, while the language provides a guarantee that already proven verification conditions are not violated.

This paper gives an informal discussion of the general concepts, omitting all details about requirement specification, refinement and reasoning. Some of the ideas are more formally treated in [15], presenting a language called OUN, where also several examples are given. Efforts are engaged in developing an OUN Toolkit based on the PVS [16] Toolkit (see [19]). This toolkit provides for OUN editing, syntax- and type-checking. It is part of an integrated platform which provides in addition facilities for model checking and proof checking in PVS. In the platform, UML and OUN are used as complementary notations [20].

Acknowledgment

We are grateful for feedback from the ADAPT-FT project group. In particular, Ole-Johan Dahl, Einar Broch Johnsen, Ketil Stølen and Issa Traoré have provided valuable advice.

References

[1] The ADPAT-FT project homepage. http://www.ifi.uio.no/~adapt/.

[2] America, P. Designing an object-oriented programming language with behavioural subtyping. In *Foundations of Object-Oriented Languages, REX School/Workshop (1990)*, J. W. de Bakker, W. P. de Roever, and G. Rozen-

berg, Eds., vol. 489 of *Lecture Notes in Computer Science*. Springer-Verlag, Noordwijkerhout, The Netherlands, 1991, pp. 60–90.

[3] Dahl, O.-J. *Verifiable Programming*. International Series in Computer Science. Prentice-Hall, New York, N.Y., 1992.

[4] D'Souza, D. F., and Cameron Wills, A. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, Reading, Mass., 1999.

[5] Fowler, M., and Scott, K. *UML Distilled: Applying the Standard Modeling Object Language*, second ed. Object Technology Series. Addison-Wesley, 2000.

[6] Hjálmtýsson, G., and Gray, R. Dynamic C++ classes. In *Proc. of the USENIX 1998 Annual Technical Conference* (Berkeley, USA, 1998), USENIX Association, pp. 65–76.

[7] Hoare, C. A. R. *Communicating Sequential Processes*. Prentice Hall, 1985.

[8] Jones, C. B. *Developments Methods for Computer Programms – Including a Notion of Interference*. PhD thesis, University of Oxford, 1981.

[9] Jones, C. B. *Systematic Software Development Using VDM*, second ed. Prentice-Hall International, Englewood Cliffs, New Jersey, 1990. ISBN 0-13-880733-7.

[10] Liskov, B., and Wing, J. M. A new definition of the subtype relation. In *Proc. of the European Conference on Object-oriented Programming (ECOOP'93)* (Kaiserslautern, Germany, 1993), O. Nierstrasz, Ed., no. 707 in Lecture Notes in Computer Science, Springer-Verlag, pp. 118–141.

[11] Liskov, B., and Wing, J. M. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems 16*, 1 (1994), 1811–1841.

[12] Malabarba, S., Pandey, R., Gragg, J., Barr, E., and Barnes, J. F. Runtime support for type-safe dynamic Java classes. In *Proc. European Conference on Object-Oriented Programming (ECOOP'2000)* (Sophia Antipolis, France, to appear).

[13] Meseguer, J. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science 96*, 1 (1992), 73–155.

[14] Meyer, B. *Object-Oriented Software Construction, 2nd Ed*, second ed. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1997.

[15] Owe, O., and Ryl, I. OUN: A formalism for open, object-oriented, distributed sytems. Research Report 270, Department of Informatics, University of Oslo, Norway, 1999. http://www.ifi.uio.no/~adapt/.

[16] Owre, S., Rushby, J., Shankar, N., and von Henke, F. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering 21*, 2 (1995), 107–125.

[17] Soundarajan, N., and Fridella, S. Inheritance: From code reuse to reasoning reuse. In *Proc. 5th Conference on Software Reuse (ICSR5)* (1998), P. Devanbu and J. Poulin, Eds., IEEE Computer Society Press, pp. 206–215.

[18] Stølen, K. A comparison of eleven specification languages. Tech. Rep. HWR-523, OECD Halden Reactor Project, Norway, 1998.

[19] Traoré, I. The UML specification of the Integrator. Research Report 275, Department of Informatics, University of Oslo, Norway, 1999. http://www.ifi.uio.no/~adapt/.

[20] Traoré, I., and Stølen, K. Towards the definition of a plateform supporting the formal development of open distributed systems. Research Report 271, Department of Informatics, University of Oslo, Norway, 1999. http://www.ifi.uio.no/~adapt/.

[21] Warmer, J., and Kleppe, A. *The Object Constraint Language: Precise Modelling with UML.* Object Technology Series. Addison-Wesley, Reading/MA, 1999.