An Overview of a Formalism for Open Distributed Systems

Olaf Owe Isabelle Ryl

Department of Informatics
University of Oslo
{olaf,isabelle}@ifi.uio.no

Abstract

We introduce here a notation called OUN intended for both specification and design of open distributed systems, supporting formal reasoning.

OUN is a formal notation, based on trace semantics, and is highly object oriented. Classes and interfaces may be specified in OUN using high-level object oriented concepts including multiple inheritance of interfaces and classes, implementation of several interfaces by a class, dynamical addition of interfaces and classes, and dynamical extension of classes. Requirement specifications are given in the form of rely/guarantee specification.

We focus here on the concepts that have been introduced in this notation to deal with particular aspects of open distributed systems like openness, flexibility, reflection, and dynamical reconfiguration.

Corresponding author: Isabelle Ryl (isabelle@ifi.uio.no)

An Overview of a Formalism for Open Distributed Systems

Olaf Owe   Isabelle Ryl

Department of Informatics
University of Oslo
{olaf,isabelle}@ifi.uio.no

This extended abstract gives a short overview of a new formalism for open distributed systems specification and design: OUN (Oslo University Notation). A complete definition of OUN can be found in [5].

We are aiming at a formalism which allows us to specify and develop object-oriented open distributed systems in a way that enables safety and liveness reasoning about them. As much as possible, we want to include high level object-oriented programming language concepts supporting openness, but restricted such that reasoning is manageable. In particular, generation of proof obligations at run-time has to be avoided. Thus, system reasoning control is based on static typing and static proofs, and the generation of verification conditions is based on static analysis of pieces of program or specification text.

We allow a form of partial compilation, such that pieces of code can be compiled at different times and added to a running system, without restarting or stopping it. The compilation units may build on each other in a natural way. In particular, a compilation unit may add functionality to an old class, by means of so-called "class extension". Each partial compilation will generate a set of verification conditions to be proved. The language ensures that already proven verification conditions do not need to be reproved when adding new compilation units, not even when dynamically extending a class with new methods, but sometimes new verification conditions are needed.

Neither Java nor Corba are suitable for static reasoning. We therefore reconsider object-oriented language constructs supporting both openness and reasoning control. Even though the main focus of this work is at the specification level, we choose to sketch a high level object-oriented design language supporting our goals. In particular we present an imperative class concept allowing dynamical extension of methods and support of new interfaces. This enables us to design systems where old objects may communicate with newer objects through new interfaces (as well as old super-interfaces).

Due to static typing, software errors such as "method not understood" and "illegal parameter types" may not appear at run time. (However, hardware errors might be reflected by network errors, and objects not responding.) The formalism ensures that proved invariants can not be violated, say, by dynamical extensions of classes or addition of new interfaces or classes.

Objects are considered to have internal activity, running in parallel. An object may support a number of interfaces, and this number may increase dynamically. The notation offers a large degree of *flexibility*: several communication paradigms can be used, operations can be redefined without severe syntactic or semantics restrictions, unrestricted overloading of operators is allowed, multiple inheritance is allowed. Finally, certain kinds of *reflection* is considered: for example one may ask an object if it supports a given interface.

In the first stage of the work, we have limited ourselves to consider safety properties (corresponding to safety properties in [1] except deadlock freedom), stated by invariants and a kind of rely-guarantee specifications similar to that of [4]. This allows us to use a rather simple semantics, based on trace sets.

Fundamentals of the notation

At the most abstract level, the specification language allows us to specify objects, interfaces, and contracts. At the design level, the programming language allows us to define classes. In both cases, requirement specifications consist of invariants and assumptions about the behavior of the environment, given as formulas on the communication history of the (sub)system. The OUN notation leans on a standard trace semantics, similar to that of CSP (without refusals and divergences) [3], except that objects have identity.

Object variables and object parameters are typed by interfaces, even at the design level. 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 is fixed but may be extended dynamically by adding operations and thereby implementing additional interfaces. A class is said to *implement* an interface if all operations in the interface are implemented in the class (with the same parameter lists, except for formal parameter names) and if the requirement specification of the class implies that of the interface after appropriate projection of the history. In addition there is an underlying language for defining data types, including predefined types such as integer, boolean, character and text. This sub-language is not discussed further; however, we use that of [2].

Openness: Dynamical extension of programs

For a traditional sequential system it is usual to consider the program as a single piece of program text (perhaps with an external library). An open distributed system may be the result of several program pieces written at different times, and perhaps at different locations. One way of achieving openness is to allow incremental addition of code without restarting or stopping the overall system. These program units may then depend on each other in non-trivial ways.

We consider a partial order ($<$) of compilation units, reflecting their availability in time. A unit may then depend on another one if and only if the latter is less than the former in the ordering. For instance units developed on different locations, at more or less the same time, may not depend on each other and are unordered. Compilation units developed at the same location are assumed to be totally ordered.

The kinds of system additions that might be allowed depend on the programming language (and operating system). In particular we are interested in concepts based on the principles of object-orientation. We do not limit ourselves to a particular existing language (even though our notions are inspired by Java and Corba). And as already stated we limit ourselves to language constructs that enable us to stay within the framework of static typing and avoiding runtime tests creating proof obligations.

We consider the following kinds of dynamical program extensions: addition of new classes and subclasses, addition of new interfaces and sub-interfaces, addition of new "implements" claims, addition of new operations to a class, strengthening the invariant of a class by an additional conjunct. And of course, a compilation unit may create new objects (of new and old classes), as may the execution of an operation. These issues are described a bit more below.

1. New objects may be created dynamically, and their identities may be submitted as operation parameters, allowing old objects to be aware of newer ones, and vice versa.

2. New (sub-)classes and new (sub-)interfaces may be added to a running system without recompiling the whole system. Thus a running system may be extended by new classes and interfaces, as well as new subclasses and new sub-interfaces, without stopping the running system.

   Notice that the combination of 1 and 2 allows old objects to talk to new objects of new classes through old super-interfaces.

3. Additional "implements"-claims may be stated in a compilation unit when needed. In order to establish the fact that a given class implements a given interface, one needs a syntactic check as well as a semantic check which in general generates proof obligations. Such a claim may be stated outside the class or interface (say, at the outmost level of a compilation unit), allowing such relationships to be established incrementally, even when the compilation units defining the class C and the interface I are unordered!

   For instance, this allows relating an old class to a newer interface, or an old interface to a newer class.

4. A class may be extended by adding one or several new operations, and the invariant may be strengthened, see above. Assuming that the run-time system will implement such an extension by loading the code for the operations (in the appropriate place), and then making references to them from the object representing the class (say from a table of operations local to the class), no restart of the run-time system is needed.

5. Added operations are inherited by any (old or new) subclasses (unless already defined in the subclass). Since classes cannot be modified from other locations, and since the compilation units on one location form a total order, it is not possible to (directly or indirectly) modify classes in conflicting ways.

The combination of the above allows old and new objects to communicate trough new interfaces! The dynamical additions and extensions are allowed in such a way that they have no influence on already stated properties. So the system is incremental, the security of old parts of the system cannot be violated by any new part (provided the generated verification conditions are proved).

References

[1] Alpern, B., and Schneider, F. B. Defining liveness. *Information Processing Letters 21*, 4 (Oct. 1985), 181–185.

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

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

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

[5] Owe, O., and Ryl, I. OUN: a formalism for open, object-oriented, distributed sytems. Tech. rep., Department of Informatics, University of Oslo, Norway, 1999. http://www.ifi.uio.no/˜ adapt/.