Electronic Notes in Theoretical Computer Science **14** (1998)
URL: `http://www.elsevier.nl/locate/entcs/volume14.html` 8 pages

Systematic Development of Concurrent Object-Oriented Programs

Paulo Borba [1]

Departamento de Informática
Universidade Federal de Pernambuco
Recife, Brazil

1 Introduction

Concurrent programs are very important to society. Bank accounts and cash machines, airplane ticket reservation, air traffic control, supermarket stock control, and simulations are just a few important application domains where concurrency is usually a major issue. Concurrency has specially been useful for developing applications that immediately react to user input, no matter what processing is being carried on; that is certainly not the case of the current version of Windows95! We are now observing that an even greater impact of concurrent programs on society is resulting from the development of information systems based on the Internet and WWW.

However, it is extremely difficult to develop concurrent programs. We have already seen several consequences of that, when programs simply don't work properly because of subtle errors. The same difficulty applies when concurrency is inherent in the application—several agents are simultaneously competing for the same resources—or when concurrency is implemented to speed-up programs.

Interference, which is inherent in concurrency, is the main cause of those difficulties. The execution of a concurrent program may be interfered by the execution of other concurrent programs simultaneously sharing or accessing the same resources. In fact, subtle problems may result from interference, being difficult to predict the behaviour of concurrent programs. Testing, which is not so effective for assuring correctness of sequential programs, is even less effective for concurrent programs, since those are often highly nondeterministic due to the vast range of possible interferences.

All that indicates that we need tractable ways for developing concurrent programs. In this position paper we will outline techniques and tools that

[1] Supported in part by CNPq, Grant 301021/95-3. Email: `phmb@di.ufpe.br`. WWW: `http://www.di.ufpe.br/~phmb`.

might be useful for supporting systematic development of high quality concurrent programs. Our approach is based on the integration of formal methods with object-oriented languages. Indeed, this integration has been quite fruitful for sequential systems [16,20,25]; we believe that it can also be useful for development of concurrent programs.

2 Object-oriented Languages

By providing concepts that favour reuse, maintenance, and incremental development of software, object-oriented programming is being successfully used for development of sequential applications. However, an yet promising aspect of object-oriented programming is supporting tractable ways for dealing with concurrency [17]. There is certainly potential for that: objects are naturally distributed, and some object-oriented concepts can be quite useful for controlling interference. In an object-oriented framework, shared variables are actually view as (part of) objects. Access to those variables (objects) is then restricted to a specific number of methods. The activation of those methods is controlled by the programmer, who can then determine a level of granularity for the execution steps in which interference cannot occur.

Concurrent object-oriented programming is actually a compromise between shared variable and communication based approaches for concurrency. In fact, it provides an interesting and smooth integration of concurrency with traditional approaches for development of sequential systems, being possible to reduce the main problems of concurrency with shared variables, and contrasting with the *ad hoc* integration of process algebras and sequential approaches.

However, language support is obviously not enough for development of high quality concurrent programs. For instance, it is clear that although Java's [13] support for concurrency is straightforward, it can be quite difficult to use that support safely. Therefore, a method for development of concurrent programs can be extremely useful as well.

3 Formal Methods

The importance of formal methods for software development is significantly recognized nowadays. This is mainly justified by the high level of reliability achieved by complex systems developed using languages having a clear mathematical semantics and allowing formal proofs that design steps refine (satisfy) specifications. In particular, formal methods can have a great impact on the development of concurrent systems, where the efficiency of testing for assuring correctness is substantially reduced.

Perhaps the most fortunate consequence of the smooth integration of concurrency and traditional object-oriented techniques is that it naturally supports a development method that allows a programmer to start with a sequential object-oriented implementation (specification) and then progressively introduce concurrency. Initially abstracting concurrency details is indeed extremely important for taming software complexity, but that is not enough: we also need sound justification for the progressive introduction of concurrency. Formal methods can certainly give such justification through the use of semantic preserving transformations, refinement rules, and proof techniques that assure the safe introduction of concurrency. Only in this way the programmer can avoid the insertion of subtle details that might compromise correctness.

However, among other factors, the industrial uptake of formal methods depends crucially on adequate tools, refinement theories and associated proof techniques to support the use of those methods in practice [7]. In particular, formal (or even rigorous) software development is not at all practical unless there are theories justifying the *compositional* and *stepwise* refinement of specifications and implementations. Also, tools for automating tedious (refinement) proof tasks are essential for speeding up formal verification and making proofs more reliable and partly automatic.

4 Operational Semantics

Many approaches have been suggested to define the semantics of object-oriented languages. Most approaches define semantics in terms of mathematical models [1,19,8]. An exception is [18], which define semantics in terms of a process calculus [22] based on operational semantics [23].

Because of the lack of a fully abstract mathematical model for interleaving, and the intrinsic details of the semantics of object identification and dynamic object creation and deletion, we think that the framework of operational semantics is a good alternative for specifying the semantics of concurrent object-oriented languages in a practical way. Indeed, as demonstrated in [5] and [6], the operational semantics of such a language can be easily and concisely defined, being still possible to reason about it in a pragmatical way, and use it to derive language implementations.

Also, by using a process calculus to give the semantics of an object-oriented language, one is not able to use the algebra of the calculus, or its notions of equivalence and refinement, in order to reason about the semantics and programs; for that purpose, one actually would have to use the (operational) semantics associated to the process calculus—see [18], for example. So it seems more effective to directly define the operational semantics of the programming language.

However, in order to obtain a clear and concise operational semantics definition it is essential to have a clear model for program states. Work on that direction [4] has modelled states of object-oriented programs using OBJ [12] specifications and associated order-sorted theory presentations [10]. This approach uses the theory of ATDs for defining operations on states and reasoning about them; in particular, the semantics of subtyping, inheritance, and evaluation and dynamic binding of stored attributes is directly provided by

OSA [10]. There is no need for extra, artificial encodings, as usually necessary when states are modelled by mappings from variable names to their respective values.

5 Refinement

Based on the operational semantics of an arbitrary object-oriented language, it is possible to directly define a suitable notion of refinement for concurrent object-oriented programs written in that language [6]. Moreover, this notion (relation) has some basic properties, such as reflexivity and transitivity, and comes up together with an effective proof technique for proving refinement. This technique is based on aspects of data refinement [14] and ideas from (bi)-simulations [21].

The refinement notion discussed in [6] has been explored and proved to be quite suitable as a basis for formal *stepwise* development of concurrent object-oriented programs, provided that the associated operational semantics satisfies some mild and natural conditions. In fact, several aspects relevant for refinement of concurrent object-oriented programs are considered by that refinement notion: data refinement (including dynamic data structures), nondeterminism, concurrency, interference, and refinement of atomic operations.

The unique related approach for refinement of object-*based* concurrent programs is described in [17], where many examples of formal program development are presented. Most examples are simple and elegant, and use a few refinement preserving transformation rules. Assertional "Hoare style" inference rules for reasoning about rely and guarantee conditions [15] are also used for proving refinement. However, no general definition of refinement, as in [6], is proposed.

6 Compositionality

Further work on refinement of concurrent object-oriented programs [3] proved that, under some mild and natural conditions, the refinement notion presented in [6] is a congruence with respect to various standard programming language constructors, including parallel and sequential composition, conditional, and nondeterministic *internal* choice. Choosing parallel composition as example, it was proved that

$$p \sqsubseteq q \quad \text{implies} \quad p \mid\mid o \sqsubseteq q \mid\mid o,$$

for any expression o formed by experiments (*visible* operations) and the programming language constructors mentioned above. It was also established in [3] a weaker compositionality result for the atomic evaluation constructor, which does not preserve refinement; that work also illustrates how novel compositionality properties can be derived from the basic congruence property.

Indeed, the results presented in [3] justify *compositional* development of concurrent object-oriented programs. That is essential for development of complex systems, where separate development is essential and reasoning has to be local. However, it was only considered compositionality *in the small*; properties of compositionality *in the large* are related to the semantics of module systems, which was not discussed in that work.

The proof of the congruence result justified and provided a deep insight into some of the technical decisions adopted in [6]. For instance, by analysing the proof it was concluded that the congruence result can only be obtained if experiments have an atomic and terminating nature. It could also be concluded that the congruence property is only valid for *contexts* formed by *visible* operations. In fact, this should be expected for any notion of refinement based on the observational behaviour of states with respect to a restricted group of *visible* operations.

7 Mechanical Reasoning

In order to mechanically support formal development of concurrent object-oriented programs in FOOPS [9], some tools were implemented [2]. Several techniques were proposed for integrating those tools and using them for simplifying refinement proofs and making them efficient and partly automatic.

A *symbolic simulator* is used to automate most of the routine work of checking the transitions from a given state, according to FOOPS operational semantics presented in [5]. In particular, the simulator is very useful for checking possible transitions from a set of states having common properties, since this set can be represented by a state having attributes instantiate with symbolic values. It is also clearly described in which circumstances the simulator conforms to FOOPS operational semantics and, therefore, can be used for proving refinement.

A *proof assistant* supports mechanical formal reasoning based on FOOPS refinement theory defined in [6], which provides notions of refinement of FOOPS states, expressions, and programs. This includes the encoding of first order predicate logic with equality presented in [24]. In addition to the encoding of the concepts of the refinement theory, the proof assistant also provides special inference rules corresponding to properties that are quite useful for mechanically proving refinement. For instance, a special rule establishes that the proof assistant can access the "model checking" capabilities of the symbolic simulator.

Some proofs can be automatically done using the proof assistant. However, in most cases the assistant helps to reduce object level theorems to functional level theorems, which result from proofs that a configuration pair belongs to a given simulation, and proofs that transitions from one state can happen whenever transitions from another state can happen. In general, those functional level theorems are related to non trivial functional theories, so that they cannot be automatically proved.

The symbolic simulator is an OBJ [12] program, whereas the proof assistant is implemented by an encoding of FOOPS refinement theory in the 2OBJ meta-logical framework [11]. The computational facilities supported by OBJ and 2OBJ have proved to be quite useful for mechanically proving refinement of concurrent object-oriented programs: OBJ's term rewriting facilities, which efficiently implement equational reasoning; and 2OBJ's engine for applying inference rules and tactics. However, the simulator and assistant are just prototypes whose performance has to be improved a lot to be useful for practical applications.

8 Systematic Development

The results discussed in this paper are just a basis for supporting a method for development of high quality concurrent programs. There is much more work to be done.

It is clear that our notion of refinement can be used to justify semantic preserving transformations and refinement rules, so that programmers won't always have to directly reason about the definition of refinement and details of the operational semantics of a particular language. Towards that direction, it would be specially interesting to investigate domain specific transformations, refinement rules, proof techniques and refinement properties. We think that will be more appropriate for formal development of concurrent programs.

We also believe that our theoretical results can be very helpful for the systematic development of practical concurrent Java programs. We intend to apply our results to make it easier to bridge the gap between sequential and concurrent Java programming. In particular, we want to define practical guidelines for safely introducing and removing synchronization in Java programs, without leading to deadlock or to undesirable interference.

References

[1] America, Pierre, *Issues in the design of a parallel object-oriented language,* *Formal Aspects of Computing* **1** (1989), pp. 366–411.

[2] Borba, Paulo, "Semantics and Refinement for a Concurrent Object Oriented Language," PhD thesis, Oxford University, Computing Laboratory, Programming Research Group, July 1995.

[3] Borba, Paulo, *A refinement theory for concurrent object oriented languages,* in: *X Brazilian Symposium on Software Engineering*, São Carlos, Brazil, 15-18th October 1996, pages 39–55.

[4] Borba, Paulo, *States as specifications,* in: *I Brazilian Symposium on Programming Languages*, Belo Horizonte, Brazil, 4-6th September 1996, pages 223–236.

[5] Borba, Paulo and Joseph Goguen, *An operational semantics for FOOPS,* in: Roel Wieringa and Remco Feenstra, editors, *International Workshop on Information Systems—Correctness and Reusability, IS-CORE'94*, Vrije Universiteit, Amsterdam, September 1994, pages 271–285. A longer version appeared as Technical Monograph PRG-115, Oxford University, Computing Laboratory, Programming Research Group, November 1994.

[6] Borba, Paulo and Joseph Goguen, *Refinement of concurrent object oriented programs,* in: Stephen Goldsack and Stuart Kent, editors, *Formal Methods and Object Technology,* Chapter 11. Springer-Verlag, 1996. Also appeared as Technical Report TR-18-95, Oxford University, Computing Laboratory, Programming Research Group, November 1995.

[7] Cleland, George and Donald MacKenzie, *Inhibiting factors, market structure and the industrial uptake of formal methods,* in: *International Workshop on Industrial Strength Formal Specification Techniques*, IEEE Computer Society Press, Boca Raton, Florida, April 1995.

[8] Goguen, Joseph and Răzvan Diaconescu, *Towards an algebraic semantics for the object paradigm,* in: *Proceedings, Tenth Workshop on Abstract Data Types*, Springer, to appear 1993.

[9] Goguen, Joseph and José Meseguer, *Unifying functional, object-oriented and relational programming, with logical semantics,* in: Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming.* MIT, 1987, pages 417–477.

[10] Goguen, Joseph and José Meseguer, *Order-sorted algebra I: equational deduction for multiple inheritance, overloading, exceptions and partial operations, Theoretical Computer Science* **2**(105) (1992).

[11] Goguen, Joseph, Andrew Stevens, Keith Hobley, and Hendrik Hilberdink, *2OBJ, a metalogical framework based on equational logic, Philosophical Transactions of the Royal Society, Series A*, **339** (1992), pp. 69–86. Also in *Mechanized Reasoning and Hardware Design*, edited by C.A.R. Hoare and M.J.C. Gordon, Prentice-Hall, 1992, pages 69–86.

[12] Goguen, Joseph and Timothy Winkler, *Introducing OBJ3,* Technical Report SRI-CSL-88-9, SRI International, Computer Science Lab, August 1988. Revised version to appear with additional authors José Meseguer, Kokichi Futatsugi and Jean-Pierre Jouannaud, in *Applications of Algebraic Specification using OBJ*, edited by Joseph Goguen.

[13] Gosling, James, Bill Joy, and Guy Steele, "The Java Language Specification," Addison-Wesley, 1996.

[14] Hoare, C. A. R., *Proof of correctness of data representation, Acta Informatica* **1** (1972), pp. 271–281.

[15] Jones, Cliff, *Specification and design of (parallel) programs,* In: R. E. A. Mason, editor, *Information Processing'83*, North-Holland, 1983, pages 321–332.

[16] Jones, Cliff, *Does the O-O community need formal methods?*, in: *TOOLS4: Technology of O-O Languages and Systems* Prentice Hall, 1991, pages 15–22.

[17] Jones, Cliff, *An object-based design method for concurrent programs,* Technical Report UMCS-92-12-1, Department of Computer Science, University of Manchester, 1992.

[18] Jones, Cliff, *Process-algebraic foundations for an object-based design notation,* Technical Report UMCS-93-10-1, Department of Computer Science, University of Manchester, 1993.

[19] Meseguer, José, *A logical theory of concurrent objects,* in: *Proceedings of ECOOP-OOPSLA90 Conference on Object Oriented Programming* ACM, 1990, , pages 101–115.

[20] Meyer, Bertrand, "Object-Oriented Software Construction," Prentice Hall, second edition, 1997.

[21] Milner, Robin, "Communication and Concurrency," Prentice Hall, 1989.

[22] Milner, Robin, Joachim Parrow, and David Walker, *A calculus of mobile processes,* Technical Report ECS-LFCS-89-85, 86, Laboratory for Foundations of Computer Science, Edinburgh University, 1989.

[23] Plotkin, Gordon, *A structural approach to operational semantics,* Technical Report DAIMI FN–19, Computer Science Department, Aarhus University, September 1981.

[24] Stevens, Andrew, Keith Hobley, and Joseph Goguen, *Mechanised theorem proving with 2OBJ: a tutorial introduction,* To appear, 1996.

[25] Waldén, Kim and Jean-Marc Nerson, "Seamless Object-Oriented Software Architectures—Analysis and Design of Reliable Systems," Prentice Hall, 1995.