How could object-oriented concepts and parallelism cohabit?

moderator :J.P. Bahsoun
panelists : J.P.Briot, D. Caromel, L.Féraud, O. Nierstrasz, P. Wegner

1 Introduction

In recent years, a lot of new languages and new concepts have been conceived in order to promote parallelism in the object-oriented framework. These proposals could be investigated using different concepts related to parallelism and object orientation. Among these concepts, we can find shared variables/message passing, inheritance/delegation, reflection... The degrees of a good cohabitation may be appreciated by combining the above concepts. In order to have significant criteria we have to determine how languages fit some requierments. These requirements should cover the different phases of programs development i.e. specification, design and implementation.

To structure the discussion, we may focuss on some formalisms and languages, in order to specify and to implement concurrent objects. The benefits of the different classes of these formalisms and languages should be discussed from a methodological point of view.

The issues to be addressed in this context could include the following questions: What are the main characteristic design principles and methodologies of concurrent object-oriented programming? How could the current object-oriented methodologies handle concurrent and distributed objects?

2 A Foundation for formalizing Concurrent Objects: Jean Paul Bahsoun - Université Paul Sabatier

The underlying metaphor of object orientation is that of largely autonomous objects with encapsulated state interacting by message passing. This has led many researchers to design concurrent object-oriented programming models. It is well known that concurrent programs require a much more careful analysis to prove them to be correct. We propose to adapt formal methods used for the specification and verification of conventional concurrent programs to an object-oriented framework. The usual definition of an object consists of an identity, a state, and methods which modify the state. A software system is composed of several objects which interact by means of messages. Methods are usually partial, i.e. they may be safely executed only if certain preconditions hold. In a distributed environment, the caller usually cannot guarantee that the precondition of the called method is met. Hence, preconditions are made explicit as conditions of enabledness, and messages may be blocked if the corresponding method is disabled. Our aim is to define a formalism for proving properties of agents and agent systems, reflecting their structural definition by inheritance and parallel composition. It therefore has to be compositional w.r.t. both of these means of software construction. To make the model intuitive and avoid unecessary complexity, parallelism occurs only between *different* agents, whereas each agent alone executes its methods strictly sequentially. All communication among the agents as well as between the environment and the agents occurs by explicit message passing. Therefore, we propose to distinguish between three levels of reasoning about systems built from concurrently executing agents, each focusing on one particular aspect or view of agent systems.

The *action level* is concerned with the local effects of single methods or actions offered by an agent. The execution of a method or action transforms the agent's local state. Therefore, a simple formalism based on pre- and postconditions of methods and actions is sufficient at this level.

At the *agent level*, we reason about behaviors of individual agents of a given class, dealing with both safety and liveness properties. Proof rules take advantage of the encapsulation of an agent's private state which may only change by executing methods or actions defined for the agent. In contrast to the action level, initialization and fairness conditions are taken into account. Aspects of communication and interaction with other agents are not dealt with except in the form of environment assumptions.

Finally, the *system level* models the top-level view of the entire system by an external observer. In particular, it cannot refer to the internal state of any object. Only the execution of actions by agents and the trans-

1074-8970/94 $3.00 © 1994 IEEE

mission of messages are visible. Typical properties of interest include synchronization and liveness involving several agents.

We will define three different formal languages and proof rules, one for each aspect of reasoning. These languages will be related by "interface rules". Care will be taken to obtain simple and intuitive transitions from one level to the next.The conflict between inheritance and synchronization constraints is nothing else than a particular case of the more general problem that is raised when method constraints and inheritance are put together. Indeed, a synchronisation constraint is a method constraint with a different semantics. Whereas a non-satisfied precondition raises an exception in the sequential case, it requires waiting in the parallel case.

In our approach [1], the problems of synchronization constraints caused by parallelism between methods disappears, since the agent itself executes the methods. Inheritance is treated at the sequential level, and all the aspects related to parallelism are treated at the composition level. The three levels of logic proposed correspond well to our intuition. The properties expressed with the logic of actions are automatically inherited (if, of course, the concerned method is not redefined or extended in the subclass). Therefore, only the proofs of agent properties have to be checked. But, under some conditions, we can reuse the existing proof of a property in a subclass.

[1] Jean Paul Bahsoun, Stephan Merz, Corinne Servieres. A Framework for Formalizing Concurrent Objects Technical Report-IRIT November 1993

3 Abstract Control Types for Concurrency: Denis Caromel- Université de Nice

After the initial breakthrough of concurrent Object-Oriented Languages (OOL) —the paradigm of active objects or agents, the unification between routine call and inter-process communication— we are forced to admit that none of them has succeeded in addressing all the issues of parallel programming.

First of all, even the underlying basic model of concurrency of OOL differs in several fundamental features. Regarding the definition of parallel activities, we still discern at least the three categories of processes (sequential, quasi-concurrent, concurrent), and regarding the semantics of communication, the variety is even more important (asynchronous, semi-synchronous, asynchronous with interruption and rendezvous, synchronous, ...). We believe that successful languages will offer several kinds of communication: because specific properties can be proved for each communication type, and within one application domain, and even one system, there are heterogeneous requirements. As to the nature of processes, we conceive of no other choice but sequential processes for OOL with some Software Engineering concerns. Probably OOL featuring concurrent processes will be restricted to languages dedicated to object-oriented operating systems.

While objects have succeeded in bringing some reuse in concurrent programming, so far they have failed to give a comprehensive answer to a complex problem we would like to focus on: the reuse of synchronization constraints. Researchers have been trying to design a universal framework for expressing concurrency control (path expressions, synchronization counters, activation conditions, enabled-sets, behavior abstractions, pre-post ambles, synchronizers, ...), but we believe there is no such absolute abstraction; indeed, most of them present interesting properties and are useful in specific contexts. Therefore, at the programming level, we must focus on designing languages which permit to program and reuse abstractions: to build Abstract Control Types (ACT) as we once recognized the need to build user-defined Abstract Data Types, not only to use a few predefined data-structures.

In order to achieve this goal, some characteristics are needed at the language level. Regarding the concurrent aspects, in our view, the critical features are: explicit control, routines and requests as first class objects, access to the list of pending requests. But there are some other language issues which are not specific to concurrency. One of them is syntactical: many abstractions use specialized notation and if we want an ACT approach to be successful, we should be able to keep at least some flavor of the original syntax. Other application domains of OOL face the same problem, namely expressing the syntactic constructs of one language within another. We believe we will have to develop better solutions than just overloading and infix functions. Another issue concerns static type checking: since ACT are programmed and no longer built-in within a language, static controls are sometimes difficult to establish. However, since concurrent applications are shifting towards Open Distributed Systems, the way we enforce type coherence in software systems will have to be reconsidered anyway.

Finally, and beyond the language aspects, ACT need to be made widely available into structured libraries of components. We ought to define criteria for their classification, to formally study the complexity of their implementation, to precisely document their properties regarding expressiveness and reuse of synchronization in different application domains. In the long term, some ACT should be enhanced with their specific formal system in order to prove properties and to achieve formal derivations within each particular framework.

[1] Caromel, D. "Programming Abstractions for Concurrent Programming", Technology of Object-Oriented Languages and Systems (TOOLS Pacific'90), November 1990, Sydney, Australia, pp. 245–253.

[2] Caromel, D. "Towards a Method of Object-Oriented Concurrent Programming", Communications of the ACM, September 1993, Volume 36, Number 9.

4 Reuse and concurrency: Louis Féraud-Université Paul Sabatier

Object oriented programming is a traditional way to achieve reuse,however putting it in practice leads to peculiar questions induced by the existence of concurrent activities: parallelism adds supplementary requirements to solve the potential conflicts occuring when an object is used.The inclusion of concurrency in object oriented programming can be reached through various approaches. Among them,we find object oriented concepts aimed to program parallelism.Here, synchronization plays an important part in the affair.As a matter of fact, a request to parallel objects is usually confronted to constraints due to concurrency before serviced. Several answers to address this problem have been proposed through programming language constructs.Some approaches include explicit code in each object to treat the requests. Because of the possibility of scattering this code in methods,reusability of parallel objects is difficult to achieve. In other proposale,[1] synchronization is defined as sets of constraints bound to objects, not as primitives appearing within parallel activities. It seems more workable to organize reuse by considering the latter approach, i.e. encapsulating synchronization. Another way to cope with concurrency is to extend a sequential object oriented language such as C++ with new mechanisms devoted to implement parallelism . An extension mode makes it possible to build a parallel program using two class hierarchies [2].The first one is devoted to traditional objects features such as attributes and methods while the second one concerns the behaviour of objects running concurrently. Then parallel objects can be obtained using a multi-inheritance mechanism with two lines of ancestors.The major benefit of this approach is to make it possible to reuse separately pieces of both hierarchies, the one induced by the sequential classes maintains the reusability potentialities of the extended language.In this approach, parallel programming is mainly viewed as the hierarchical composition of reusable software components [3].In contrast with some other models, we think that behaviour classes are not to be predefined because it appears difficult to give an exhaustive taxinomy of concurrent behaviours.The behaviours are to be built up by developers according to the requirements of the parallel applications that they are dealing with.Note that the above extension mechanism is rather related to a family of sequential languages than to a particular one,behaviour classes can be viewed as a kind of coordination language for sequential software components. Let us now consider the activity of software production for a distributed environment.If we assume that the developers are aware of the environment they use, they may want to control how the objects implementing their application are distributed.In such a case, they have to program explicit distribution.To solve this point, several approaches[4] make it possible to split objects into distribution fragments. If we again look into the strategy of extending a sequential language to distribution, we have now to define object oriented concepts suiting these new requirements. Considering that objects can be broken down into distributed pieces, it appears now pertinent to provide the programmer with concepts describing how the objects are decomposed in fragments and how these fragments are evolving during the execution of the program.Denoting the constitution and the evolution of fragments, may be reached by the means of the concept of a virtual configuration. Configurations can be specialized and reused as usual software components. Generally, distributed objects communicate using protocols.Defining protocols in communication classes, outside the implementation of the features and the configuration description allows to reuse them.From that viewpoint,it appears that communication supports can also be considered as reusable software pieces. Software components reuse in a distributed environment can be obtained by extending sequential object oriented concepts to new specificities. In addition to traditional constructs implementing the objects features, new mechanisms are needed,configuration classes or communication classes appear as relevant tools. The major advantage of this approach seems to rest in the preservation of the reusability power of the sequential underlying language.

[1] S. Frolund,G.Agha "A language framework for multi-object coordination". Proc. ECOOP'93 pp 346-360.

[2] J.P.Bahsoun,L.Féraud,C.Betourné A two degrees of freedom approach for parallel programming Proc. IEEE ICCL'90 March 1990 New Orleans.pp261-270.

[3] J.P. Bahsoun, L.Féraud A model to design reusable parallel software components Proc. PARLE'92 LNCS Springer Verlag 605 pp 245-260.

[4] G. Kaiser, B.Hailpern" An object based programming model for shared data" ACM TOPLAS April 1992 Vol.14 no2 pp 201-264

5 A Foundation for Composing Concurrent Objects : Oscar Nierstrasz-Université de Genève

Modern software applications can best be described as Open Distributed Systems (ODS). These applications are often not only reactive systems – that is, inherently concurrent and long-lived – but they are typically susceptible to constantly changing requirements. For these reasons, traditional software development methods and programming languages are poorly equipped to address these needs.

Robustness with respect to changing requirements can be achieved through strong encapsulation (i.e., by localizing the effects of changes), by raising the level of abstraction (i.e., achieving a high degree of configurability), and by systematic software reuse (i.e., developing generic solutions that can be applied to a class of problems).

Object-oriented languages go a long way to addressing the needs of ODS (1) by providing objects as an organizing principle for applications, and (2) by providing various mechanisms (such as inheritance) to support systematic software reuse. But object-oriented languages suffer from both computational and compositional weaknesses: First, there is no generally accepted model of concurrent communicating objects. Second, the notion of a "software component" is typically supported in a strictly limited and ad hoc fashion, and may be poorly integrated with other, computational features of the language.

Many of these problems can be traced to an overemphasis on inheritance as not only a mechanism for classifying objects, but as the principle mechanism supporting systematic code reuse. Though classes, viewed as software components, actually have two different kinds of clients – client objects and subclasses – the compositional interface to subclasses is typically defined using ad hoc rules rather than through the type system. As a consequence, inheritance can violate encapsulation in a variety of ways (as has been well documented). Furthermore, inheritance of synchronisation constraints for concurrent and communicating objects has been shown to be a difficult problem. Finally, classes provide a strictly limited granularity of software component, and are often extended with other concepts (mixins, generics, modules, etc.) to truly support systematic software reuse.

Attempts to consider concurrency as an "add-on" to sequential languages have yielded numerous anomalies in addition to the problems of inheritance. We propose to adopt an approach in which objects are certain kinds of (well-behaved) communicating agents, and software components are simply abstractions (i.e., functions) over the object space. In this view a class is just a first-order abstraction over objects. A "wrapper" is a higher order abstraction that yields classes or subclasses. Semantically, an integration of functions, objects and agents is needed at the lowest level. We propose that recent progress in the study of process calculi can provide us with many of the need ingredients for an abstract machine, and that we should build concurrent object-based languages on top of such a foundation. The test of this approach will lie in whether we can succeed in specifying frameworks of software components for constructing flexible ODS.

6 Interaction Power, Persistence, and Concurrency: Peter Wegner-Brown University

The observable behavior of objects cannot be expressed by computable functions because objects in software systems have a physical existence in time, called persistence, that causes them to have time-dependent physical properties. Functions capture the transformation power of obejcts at an instant of time but not their interaction power over a period of time. Objects determine a continuing marriage contract for interactive services over time that cannot be captured by a pattern of one-time sales contracts.

The recognition that functions are too weak to express the observable behavior of objects over time has far-reaching consequences. The gulf between algorithmic programming in the small and interactive programming in the large becomes one of expressive power rather than merely of scale. Functional and logic programming languages are seen to express the observable behavior of functions but not of objects. The Turing Test is interpreted as an attempt to express intelligence in terms of the computing power of objects rather than of computable functions. Church's Thesis that Turing machines capture the intuitive notion of effectively computable functions loses its force because functions do not model software systems.

Abstractions simplify concepts by focusing on their relevant properties and ignoring irrelevant ones. Object abstractions focus on observable attributes of objects and ignore unobservable attributes. Because functions ignore time-dependent observable attributes of objects they do not capture their "fully-abstract" observable behavior. Functional semantics views interface procedures as instantaneous events in the lifetime of an object and cannot capture noninstantaneous procedures with real-time constraints or nonserializable concurrent execution.

Software objects exist in the same time dimension as physical objects and can capture the passage of physical time merely by their existence. The richer expressive power of objects does not provide greater transformation power, such as the ability to solve the halting problem. It provides greater interaction power in an orthogonal temporal dimension. The orthogonality between transformation power and interaction power facilitates the independent study of transformational and interactive properties of objects.

Intractable abstractions are a key to extending modeling power in both software engineering and nonconstructive mathematics, because useful behavior can be recovered by further abstraction just as one dimensional distances can be recovered from a two dimensional map. Physicists find no problem in creating useful abstract theories about an inherently unobservable world of "real" (Platonic) objects, while mathematicians create integers and rational numbers by abstration from the inherently unrepresentable "real" numbers. Software engineers likewise capture useful properties of "real" interactive objects in software systems by focusing on purely functional properties. By embedding algorithmic computation in an interactive modeling framework, software engineers can realize greater modeling power without giving up effectiveness.

We agree with Milner (CACM, January 1993) that concurrency is a necessary element of interaction. However, the breakdown of functional semantics in concurrent systems is a consequence of their persistence. Sequential software systems constrain concurrent activity but cannot eliminate concurrent existence or nondeterministic interaction with their environment. Even shared variables of a Von Neumann computer, though protected against explicit concurrency, must handle nondeterministic access by concurrently existing persistent clients. Concurrent functional and logic languages that capture parallelism without persistence have a more tractable notion of time and a more tractable semantics than imperative objects.

Concurrency is needed to handle extreme situations in continuously operating software utilities even when normal system operation is sequential. This situation is somewhat similar to the need for non Newtonian physics in extreme situations of very high speed or very small size. Noncomputable behavior is thus a necessary extreme feature of most interactive systems that must be properly handled if misbehavior in extreme situations can cause unacceptable harm (hard real time constraints). Noncomputability is a form of computational turbulence that mimics the turbulence of physical phenomena. The expressive power of interaction is examined in greater detail in [1].

[1] Peter Wegner, The Expressive Power of Interaction, Brown Technical Report, Dept of Computer Science, December 1993.