Concurrency Issues in Object-Oriented Modeling

Ms. Snehal Chaflekar

Professor, Dept. Of Information Technology, PBCOE, Nagpur, Maharashtra, India

---***---

Abstract - *The work described in this paper is a rst attempt to nd a synthesis of concurrency and the object model. A representative sample of concurrent object-oriented languages has been analyzed to identify issues {dimensions{ peculiar to the conjunction of the two features of interest. The presentation includes sections that review and develop the basic concepts, both in concurrency and the object model, needed for the analysis of the languages. The relevant issues are presented in a structured way, along with a discussion of pros and cons of the possible alternatives. Issues identified include several concurrency and encapsulation features and also communication, migration and transparencies among others. Some preliminary conclusions are ordered, as well as suggestions for future work.*

Key Words: Concurency, Co-routine, Encapsulation, Object, Thread

1.INTRODUCTION

This work is an attempt to nd a synthesis of two important elements in the design and construction of software, as they are re ected in programming languages. We investigate object-based programming languages that o er concurrency (henceforth also called concurrent object-oriented languages.)

1.1 Concurrency

The design and construction of software in which (loosely- or tightly coupled) concurrency is involved has always been a hard task. In the past, it was reserved to the initiated who were responsible to build certain type of software, such as operating systems and some real-time or distributed applications.

The discipline evolved, and new tools and concepts were added to ease the task of the concurrent programmer and improve the quality of her work. Unfortunately, these achievements have not kept pace with the increased complexity and diversity of demands that are put on concurrency. Driven mainly by the desire to take advantage of decreasing costs of hardware, but also for more stringent reliability (and other) constraints, the constant need to tackle more complex problems, and especially problems whose solution is expressed naturally in concurrent terms (thus avoiding overspecification), the discipline of building concurrent programs is now spreading to areas and (uninitiated) people not expected a few years ago. The quest for new design techniques, new paradigms, etc., has increased correspondingly, but we are still far from a panacea. object model Though the object-oriented model has roots in simulation and artificial intelligence, they remained relatively unknown until the eighties, when it sprang to notoriety with the Smalltalk phenomenon. The uniformity of the approach (\everything is an object or a message between objects") captured the interest of a community burdened by the complexity of the problems it was tackling and the tools it was using. To name just two of the expectations raised, it was anticipated that programming would move closer to design, and that reusability of software would be practical. While initial results have not been miraculous, there are signs that indicate that this is a (maybe `the') right approach to building software.

Object-oriented methodologies are having a deep e ect in the culture and practice of software building, and we witness trends to introduce them in all kinds of application domains, at least as a new engineering paradigm, much in the fashion of `structured pro-gramming' in the seventies.

1.2 concurrency + objects

It is not surprising, given the importance of both concerns, that several groups of researchers trying to apply the object-oriented methodology to the harnessing of parallelism. This is done through attempts to extend the object-oriented paradigm to include concurrency. Besides, for reasons similar to those that fuel the use 3of concurrency, there is a great demand for distributed languages and systems, so the initiatives tend to include both concurrency and distribution.

Foundations are considered first, and only then is presented the core of the work, consisting of a set of interesting features identified during our survey of the languages. We analyzed the literature for each language to nd out how (and if) the facilities as-sociated with concurrency and object-oriented are provided.

2. CONCURRENCY

Concurrency corresponds to the everyday experience of carrying out several tasks at the same time, something typical of human activity , we note that humans can simultaneously chew gum and walk in a straight line But things are not so simple in the world of computer programming. When studying programming, the conventional approach is to avoid concurrency as long as possible, trying instead to nd sequential algorithms to solve the problems. This is attributed to the increased complexity inherent in dealing with concurrent algorithms. It is hard to `think concurrent,' to use a loose but evoking expression. Not to mention the di culties associated with specifying, deriving, and proving correct concurrent programs (as compared with the same activities in the case of sequential programs). However, there exist several strong reasons to use concurrent algorithms. One is economic: the more parallelism one has, the more (computational) resources can be poured in to solve (faster) a problem. Trends in the recent past show that, with the declining costs of hardware, the primary e ciency concern for a large family of applications is execution time (correctness provided, of course: more on this below). Since physical limits to computer peed are fast approaching, it is clear that introduction of parallelism is the only way left to speed things up.

The other reason is {conceptually{ more interesting: many problems are inherently concurrent in their formulation. The interest for concurrent solutions is obvious: they will be the most natural ones! To understand why this does not contradict what was said above about the complexity of concurrent solutions the reader should realize the gap between the `nature' of a problem and its formalization in a way suitable for the derivation of a computing solution. And here is where programming languages play their part: they are {at some level{ the tools the problem solver uses to express her understanding of the problem, or better, the tools she 2 uses to express a suitable algorithm. It is not within the scope of this paper to discuss the precise role of programming languages in problem solving or even in the formulation or description of algorithms. We nevertheless believe they are important enough to justify their study. Moreover, in dealing with object-oriented concurrent languages, the eld is very far from closed. At this point, we hope to have motivated the reader to carry on, ready to endure the review of concepts related to concurrency. We believe this section to be necessary because of the myriad de nitions, semantics, etc., through which concurrency is introduced in the literature. The following is not a complete survey, but should su ce as a recapitulation and to agree on some basic concepts and terms.

2.1 Fundamental Concepts

We do not start from the de nition of algorithm, program, process (these issues well dealt with in e.g. Ancilotti 88]) but instead review some basic concepts. The (computational) activity is the basic notion of (sequential) execution: it may be conceived as abstracting the notion of one processor executing one program (or as the dynamic counterpart, shown in the trace, of the static speci cation represented by the program.) We say that two activities A and B are concurrent if they do not have a necessary temporal ordering between themselves, i.e. A might occur completely before B , the reverse could also happen, or they could even overlap in time. (An intuitive notion of time succes for our purposes.)

We have parallelism when concurrent activities overlap in time we also say in that case that concurrency is being exploited. Two concurrent activities communicate when they are able to exchange information by some of the means described below.

2.2 Language constructs and tools for concurrency

Various constructs to express concurrency in programming languages have been proposed.

2.2.1 co-routines

The first construct to handle concurrency at a language level is the coroutine, introduced by Conway (1963). A coroutine is like a procedure in that it has local declarations and code. It di ers from a procedure in the way control is handled, and in the fact that its state persists across control transfers.

A procedure is invoked with a call statement and returns control to the caller via a return statement. There exists a hierarchical relation between caller and callee: the callee does not know which unit called it, and cannot choose where the control will ow to on its termination. A coroutine, on the other hand, may suspend its execution via a resume statement, transferring control to another coroutine whose name is indicated as the parameter in the resume call. Coroutines are therefore organized in the same level whereas procedures are 6organized in a hierarchy. When it is resumed later, the coroutine continues execution from the point it did its last resume, its state unaltered.

Notice that the coroutine concept allows the interleaving but not the parallel execution of activities, i.e., no two coroutines can be executing simultaneously. Considering uniprocessor machines, however, they are very useful, for any concurrent program is executed by interleaving, so coroutines are a good basis on which to build an interleaved implementation of concurrency.

2.2 fork and join

The fork and join pair of constructs is also due to Conway. The fork instruction produces two concurrent executions within a program, namely the continuation of the invoker and the spawning of a new one starting from some point in the program (speci ed as a label in the argument), thus splitting in two the control ow. Note the di erence with coroutines: we now have two executions that are rather independent, but are in fact executing (portions of) the same code, and sharing the name space. The join is the symmetric operation. When encountered by an activity, it nishes its independent execution and disappears. Note however that its parent activity (the one that fork ed it) anyway `inherits' whatever data the dead activity might have elaborated for these data is in the shared name space. So an explicit passage of results is not needed. This is a powerful primitive: it can be proved any form of precedence in concurrent actions may be expressed as a composition of forks and joins.

2.2.3 parbegin . . . parend

This parallel construct was introduced by Dijkstra. It is analogous to an Algol block in its shape. But in this case the statements enclosed in the brackets are all executed concurrently. It is more structured than the fork and join, but less powerful. There are precedence constraints for concurrent programs which are not representable with parbegin . . . parend that are representable with fork . . . join (see example in Peterson 85]). But the power is equated if the par- pair is augmented with suitable synchronization primitives (e.g. semaphores).

2.2.4 processes

Processes are another way of structuring concurrency. The idea is to allow several sequential programs to execute concurrently, each having its own program counter, name space, etc. Interaction among them may take essentially two shapes: by message passing or by sharing of variables Lauer 78]. Much is gained in this schema, for encapsulation allows certain con icts (e.g. interference on shared data) to be managed more easily. Also, the design and programming are easier {at least in principle{ because the smaller units are sequential: concurrency only occurrs among full-sized processes (i.e. processes the size of stand alone programs).

2.2.5 synchronization

There exist several ways of synchronizing activities: they are roughly divided according to whether there is any memory shared among the activities. In the case of shared memory we have the semaphore due to Dijkstra, which is a variable that may be accessed only through two special operations, P and V . The semantics of the operation is such that under certain conditions an activity executing a P on a semaphore may be delayed. It will only allowed to carry on when another activity executes a V on the same semaphore. So its functioning reminds us of the system for crossing a narrow (one-lane) bridge, in which the driver of the last car of a group allowed to proceed is given some token (e.g. a colored stick) on one end of the bridge, and only when she reaches the other side is it possible to allow the crossing in the other direction.

It is well known that semaphores are elemental and powerful, but very unstructured {thus prone to error. Another, more structured synchronizing primitive is the monitor construct introduced by Hoare. It encapsulates data so that it is manipulated only through exported operations, and its semantics guarantees that there is at most one active operation at a time, thus serializing access and preventing interferences in the shared data. In the case of no shared memory, synchronization is done via synchronous message passing. Sending (or receiving) a message is viewed as calling a special procedure, in which the activity is stuck until the partner of the synchronization does the symmetric operation. Then both return normally, resuming their respective executions.

2.3 Threads

A controversial concept Consider a conventional operating system process. It is a dynamic concept, in that it is the executing counterpart of a static speci cation of behavior given in its corresponding program. It is basically an executing entity, with associated resources and data. It comprises 8 several parts. It has a name space that delimits the entities it owns, as well as permissions to access and use resources in the system. It is protected by boundaries from other processes.

It has an execution state (i.e., ready, suspended, etc.) It may have an associated priority. But, again, it also has a dynamic part, i.e., a thread of control, which leaves as a trail the `trace' representing the path of execution. This thread is like an abstraction of the processor, whereas all the environment corresponds to the physical resources, e.g. memory, peripherals, etc., the processor controls (analogous to the `owning' of objects in the name space of the process.)

The other is its thread , the basic unit of cpu utilization , or abstraction of the executing agent, which corresponds to our intuitive concept of activity, outlined above. But consider now having more than one thread in the same environment: now we have concurrency `internal' to the process, sharing the same name space. In our machine analogy, it is as though we had a multiprocessor with a shared central memory. We now have a notion of a process which may have internal concurrency through the existence of several threads of control sharing an address space (multithreading). This of course requires adopting precautions to prevent the interference typical of unchecked concurrency, i.e., the problems derived of sharing entities under parallelism. These ideas have been introduced and studied exhaustively in the operating systems literature. A process is de ned there as a set of threads of control executing within a single virtual address space.

3. OBJECTS

This section reviews the concept of object-orientedness in the domain of programming languages. It rst introduces the notion of object in an intuitive way, through an informal discussion. Then a classi cation scheme is o ered, aiming to identify interesting features of object-oriented languages. Concurrent features are examined using concepts from the preceding section.

3.1 What is an object?

In the real world, an object is \anything with a crisply de ned boundary" Cox 86]. While this is certainly not enough to work with the object concept, it re ects the important things about objects in the computer science `world.' One of the essential reasons why people want to use object oriented methodologies is because they want to map things they see in the real world into computer representations. This approach is feasible not only in simulation, where its usefulness is obvious, 4 but also in many situations in which a problem, its solution, or both can be thought of in terms of objects. An Object is essentially an encapsulation which encloses some data

The behavior of data objects is expressed most naturally in terms of operations that are meaningful for those objects. This set includes operations to create objects, to obtain information from them, and possibly to modify them. For example, push and pop are among the meaningful operations for stacks, while integers need the usual arithmetic operations. Thus a data abstraction (or data type) consists of a set of objects and a set of operations characterizing the behavior of those objects.

3.2.1 Non-concurrent features

The analysis identi es six orthogonal dimensions of object-oriented language design. Concurrency is dealt with in the next subsubsection the rest are explained here. class classes serve to classify objects in sets with uniform behavior. They specify operations common to all instances and serve as a template from which objects may be created. inheritance class inheritance is a mechanism for sharing operations de ned in a superclass by a number of subclasses. Inheritance schemes di er in the way an invoked operation of an object is matched to a defi nition. It is an object whose state is accessible only through its operations. Its state is generally represented by instance variables. strong typing a language is strongly typed if type compatibility of all expressions representing values can be determined from the static program representation at compile time.

3.2.2 Concurrent features

Before examining the possible forms of concurrency in object-based languages, let us introduce the central entity with which we will deal in the rest of the work, namely the processes (which we will call active objects .) active objects active objects have an object-like interface of operations and one or more threads of control that may be active or suspended. (Threads were de ned in the section on concurrency.) There are two aspects in this classi cation. One is how is the internal concurrency is provided the other concerns ways of synchronizing the threads.

4] CONCURRENCY, DISTRIBUTION AND OBJECT

In this section we examine relevant issues in concurrent distributed object oriented languages. We will use the background laid out in the preceding sections to understand the features presented here. We believe the following features constitute what characterizes a language of the class we are considering. We have based our analysis in three main sources: a set of recent languages that have been branded `concurrent object-oriented' in the literature.

Before going into the enumeration and analysis of important aspects of languages, it is important to stress the limitations of the analysis: we use an informal method. A central idea is the concept of consistence, but no claim will be made that the proposed set of language features is consistent. Moreover, this is not true of the set of all proposed features some of them are mutually con ictive. (In that case a tradeo is obtained using more speci c criteria derived for example from the application domain.) We are not be too concerned with this because we are not trying to design a language, so we do not need to choose which features to include in it, and at what price. Note that claiming consistency of a set of features would entail exhibiting a language which contains all of them.

We nevertheless believe the analysis to be very useful, both to our present purposes of gaining a better understanding of concurrent object oriented language and in the quest for a formal treatment of them. We think that a rst approximation to this problem should be informal and somewhat experimental. While we do not claim to have de ned a design space in a rigorous sense, we have isolated su cient but maybe more than necessary concepts to start trying an identi cation of `elemental' ideas. We expect criticism to our `shopping list' of features. On the other hand, the notion of design space has the drawback that all dimensions are born equal, i.e., we are forced to adopt a at hierarchy of essential features. While this is desirable from a formal (and maybe aesthetic) point of view, we think that it does not re ect our current knowledge of the area. Only when we identify the very basic concepts on top of Taking the languages and their features as phenomena to observe and try to explain should ultimately lead to building a useful model.

4.1 Three concurrency issues

One natural aspect to look at in these languages is that of concurrency. We are interested in the way in which concurrency is introduced (or o ered) in the language. There are three aspects to concurrency here. The rst is whether the semantics of the language prescribe an interleaved execution (nevertheless called `concurrent') or allow true concurrency. The second is peculiar to concurrent object-oriented languages and is about the relation of objects to concurrency: is concurrency o ered between objects (i.e. by concurrent execution of di erent objects), within objects (i.e. by allowing several threads inside an object), or both? Finally, we have an issue that also arises outside the object-oriented domain 8 but is strongly coupled to the previous, namely the `grain' and `weight' of concurrency o ered by the language.

4.1.1 Concurrency: `true' versus interleaving

It has already been pointed out that some constructs (e.g. coroutines) allow (quasi) concurrency but not parallelism. In particular some models of (active) objects can service only one request at a time, i.e. can have at most one active thread.]. These languages are nonetheless termed concurrent because they allow interobject concurrency. Quasi concurrency is useful because it reduces the time that zero threads are executing Wegner 87] (when a thread suspends itself to wait for an event, it allows another thread to enter the object). Other languages, in contrast, o er real parallelism. Note that this device conceptually eliminates queues (although queueing does happen whenever the number of requests exceed that of available processors {but this is and implementation issue and is taken care of by the runtime system). I.e., in an `axiomatic' sense

4.1.2 Intra- and inter-object concurrency

Concurrency may be obtained in an object environment in two di erent ways, namely: intra-object (or intranode) it is when inside an object several threads are allowed to execute concurrently (this may turn up to be quasi-concurrent, like e.g. in ABCL/1) inter-object it is simply due to the independence of di erent objects, which may be ex- ecuting simultaneously. This case is simpler because by the very nature of these objects (i.e. because they are units of distribution) there is a very controlled interac-tion among them because they do not share any space and communicate through a well-de ned interface.

Some authors Power 88] argue that there is a third form of concurrency, namely that obtained by creating an object. We consider this to be a special case of internode concurrency, because concurrency is increased by independent and loosely coupled execution of new threads in a di erent name space. Power's view does not a ect essentially our subsequent discussion: the reader may take either view and transform the statements suitably and our discussion will retain its meaning.

4.2 Communication, synchronization

The types of communication mechanisms available to a language of the kind we are considering are essentially those usually associated with distributed systems. Though message passing is used in conventional object-oriented systems, it usually has the conventional procedure call semantics. (It is worth noting that in object-oriented languages message passing may be present without there being any distribution.

5. CONCLUSIONS

The main conclusion is one that somehow prevents other conclusions. It is the recognition that the area we are trying to probe is very new and so still wide open and uncharted. This view is shared by others (Moss 88], Hogg 88], Nierstrasz 88]). This is the reason for the lack of uniformity in concepts and terminology. Another conclusion is that it is essential to formulate unifying models at least we should strive to agree on what are the basic concepts that matter here. We believe our work is a step in that direction as it helps identify important issues. An intuition that was present at the beginning {and that partially prompted this research{ emerges stronger after the work: the marriage of the object model and concurrency is very interesting in at least two counts. One, the promises it holds in terms of easing the task of designing and building concurrent systems. And the other {last but not at all least{ is the challenge of making it possible, conceiving and really building the abstract models that allow a deeper understanding and the ulterior formalization of the features pointed out in this work. So, the road to follow is clear: to devise a model that adds concurrency to a more precise object model. In the short term, the rst step is to isolate a set of elemental mechanisms (in an axiomatic sense) able to embody both the notions of concurrency and the object model.

REFERENCES

[1] Jacobsen, Ivar; Magnus Christerson; Patrik Jonsson; Gunnar Overgaard (1992). Object Oriented Software Engineering. Addison-Wesley ACM Press. pp. 15,199. ISBN 0-201-54435-0.

[2] Meyer, Bertrand (1988). Object-Oriented Software Construction. Cambridge: Prentise Hall International Series in Computer Science. p. 23. ISBN 0-13-629049-3.

[3] Kindler, E.; Krivy, I. (2011). "Object-Oriented Simulation of systems with sophisticated control". International Journal of General Systems: 313–343.

[4] Lewis, John; Loftus, William (2008). Java Software Solutions Foundations of Programming Design 6th ed. Pearson Education Inc. ISBN 0-321-53205-8., section 1.6 "Object-Oriented Programming"

[5] Ben-Ari, Mordechai (2006). Principles of Concurrent and Distributed Programming (2nd ed.). Addison-Wesley. ISBN 978-0-321-31283-9.

[6] Pierre America. "Pool-t: A parallel object-oriented language". In Akinori Yonezawa and M. Tokoro, editors, Object-Oriented Concurrent Programming, pages 199–220. MIT Press, 1987.

[7]James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen, "Object-Oriented Modeling and Design", Prentice-Hall International.