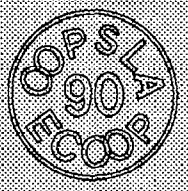

21-25 October 1991
Ottawa, Canada

Panel: *Foundations of Object-Based Concurrent Programming*

Chair:

Gul Agha, University of Illinois at Urbana-Champaign

Panelists:

Akinori Yonezawa, University of Tokyo
Peter Wegner, Brown University
Samson Abramsky, Imperial College, London University

Professor Gul Agha:

I am at the University of Illinois at Urbana-Champaign. Professor Aki Yonezawa from the University of Tokyo is a leader in the field of Object-Oriented Concurrent Programming. He has pioneered ABCL/1 and a number of other object-based concurrent systems, and is the Program Chair for this year's OOPSLA conference. Professor Peter Wegner from Brown University was the first chairman of SIGPLAN, was responsible in the early 1970s for the idea of ACM specialized conferences including the POPL conferences, and was last year's OOPSLA keynote speaker. Professor Samson Abramsky from London University (Imperial College) has contributed to the foundations of concurrent languages and will discuss the relation between foundational work on concurrency and functional programming and object-based concurrent programming. All the speakers, including myself, attended a two-day workshop on object-based concurrent programming held immediately prior to this conference and will give you their personal perspective on the workshop. Below is a summary of their remarks.

Akinori Yonezawa

I will describe four areas for further work which emerged from the two-day workshop on object-based concurrent programming held before the main OOPSLA conference.

- *Type theory for concurrent objects.* A type is a constraint on the behavior of an object and a subtype is a finer or stronger constraint. The ideas of type and subtype have been very useful for structuring programs, detecting compiling errors, and optimizing the run-time code. In the sequential world, ideas such as abstract data types, parameterized types, and polymorphic types have been implemented and are well-understood. Deep mathematical theories for types and subtypes have been developed. In Object-Oriented Programming, class and subclass are being used for code sharing and code reusing. The distinction between types and classes is useful as a means of classifying and characterizing objects. The notion of types in the Concurrent or in the Parallel world is ambiguous since the generalization of types to the richer constraints associated with concurrency is not well understood. We still do not have a good notion of type that captures synchronization constraints, interrupt behavior, and multi-party interaction.
- *Extension of delegation and inheritance to Concurrency.* Inheritance and delegation may be viewed as two ways of code sharing. Kafura and,

subsequently, Tomlison and Singh have examined concurrent delegation and inheritance. We cannot simply extend the sequential idea to the concurrent world since sharing is much more complex. I am pessimistic about finding an easy solution. The object world is moving toward finer grain concurrency. Do we really need inheritance and delegation in concurrent systems?

- *Grouping mechanisms.* Grouping is useful in identifying and naming the objects, in resource management, and in computation and debugging. What are the language mechanisms or constraints to capture grouping of concurrent objects? Connected with the issue of grouping is compositionality of the concurrent objects—in the sense of being able to compose concurrent objects to get a composite object. In most computation models for concurrent objects compositionality has not been achieved.
- *Concurrent computational reflection.* Computational reflection was introduced in a sequential context by Brian Smith. In the sequential world, the CLOS group has been working with metaobject protocols. Pierre Cointe has been working with reflective objects in LISP. Mary Shaw, in 1987, gave us an object oriented version of computational reflection. Computational reflection means representing an execution scheme so that it is manipulable at a higher level. That implies making the execution scheme dynamic, changeable, adaptable, and extensible. So there have been several proposals for using this mechanism but we have to be very careful what we are talking about since concurrency is more subtle and complex than sequential programming.

Peter Wegner

Object based concurrent programming combines the object based and concurrent paradigms: it combines encapsulation with communication and synchronization. We combine the modeling power of sequential object-oriented programming with the computational and expressive power of concurrency. The essence of object-based programming is encapsulation of local data. Object-based programming is a form of component-based software technology. One research area I would like to talk about is the extension of component-based software technology to new kinds of software components with more powerful encapsulation mechanisms.

Let me introduce the term programming in the very large (PIVL) to denote software development of programs that are not only textually large but also temporally extensive (persistent), large in the number of people that create and use them, and large in the educational and equipment infrastructure needed to support them.

One approach to programming in the very large is *megaprogramming,* a new term introduced by DARPA for an as yet undefined component-based technology for very large programs. The term expresses a wish rather than a reality: the wish for a technology to manage systems an order of magnitude larger than those we can handle today. One approach, proposed by me and Gio Wiederhold, is to view encapsulation as the key notion and to supplement data abstraction that encapsulates data at the level of objects with more powerful encapsulation mechanisms that encapsulate behavior and knowledge. We call such large abstractions with powerful encapsulation facilities megamodules. Megamodules can model large abstractions like banks and city transportation systems rather than merely bank accounts and vehicles. Megamodules may have many internal concurrent activities and their interfaces are determined by the megamodule software community rather than by global interface policing.

How does megaprogramming differ from object oriented programming? One difference is that they have heterogeneous interfaces that cannot always directly understand each other (megamodules have their own type systems and local terminology). In contrast, components in object-oriented systems generally have homogeneous interfaces. Classes (which determine object interfaces) are generally global since data abstractions generally encapsulate data but not other classes.

Open questions in the design of megamodules include: what should megamodules encapsulate?, how should they communicate?, what data conversion facilities are needed for communication between megamodules?, how should megamodule libraries be organized? We have addressed these and other issues in a joint paper by me, Gio Wiederhold at Stanford, and Stefano Ceri of the University of Milano. It is available from the computer science departments of Stanford or Brown.

Let me switch to another subject: the question of what determines module boundaries. Module boundaries are determined in part by data abstraction, in part by notions of distribution and in part by notions of synchronizations. An abstraction boundary is the information hiding boundary when you try to look in to a module. A distribution boundary is determined by how far out you can see from within a module. In block structure languages, the distribution boundary is coarser than the abstraction boundary, since block and procedure

boundaries permit looking outward but not inward. The abstraction boundary is coarser than the distribution boundary for megamodules whose abstract interface hides many internal distributed modules. The synchronization boundary is the boundary at which incoming threads are held up for purposes of synchronization. It can coincide with the abstraction boundary, as in Ada tasks, or can be more finely grained, allowing threads to synchronize within a module as in the case of monitors. It is interesting to explore the interplay of these three factors in determining module boundaries, both to examine trade-offs in module design and to determine more precisely what we mean by modularity.

My personal research agenda in the area of object-oriented programming focuses on the study of software components. What is the glue for composing components? Is the glue specifiable as components or in some other way? If the glue is components we have second order components. A second part of my research agenda, related to one of Aki's topics, is concerned with methods of incremental modification and reusability of types and classes. Inheritance is a form of incremental modification that supports both subtype and subclass modification. Inheritance is a form of glue for composing classes by composing subclass components (it supports both behavior modification and system evolution). But is it the right kind of glue? Is inheritance the right kind of notion for composing software components or is it something that is just proved useful in a special context and maybe isn't extendable to say concurrency and distributed systems.

Samson Abramsky

Theoreticians pick things apart and try and find the elementary particles, the fundamental theories, to permit their formal analysis. Things are analyzed by considering their simplest and purest forms. In contrast, object-oriented programming is a rather rich stew of things with many ingredients bubbling merrily on the pot. It combines things that have been studied separately under various guises, for example, abstract data types, subtyping, message passing, etc. It is the combination that makes the Object Oriented paradigms attractive. Theoreticians have been understanding things at a fundamental level often in separation and not in this particular combination.

To bridge the gap between theory and object-oriented programming, we must combine the ingredients that have been separately studied in a way that is relevant to practitioners. There are two distinct paradigms which have received a lot of attention that between them more or less cover the main elements of the object oriented paradigm. The first is the concurrent paradigm as formulated by Robin Milner in terms of communicating processes. Unfortunately he is not here so I'll say a bit about his work on the calculus of communicating systems (CCS) which you can find described in his recent book on communication and concurrency. This gives a notation for describing concurrent systems built up by uniform operators, combinators of communicating processes, and encapsulation, and a mathematical theory which allows you to reason about these things to show that one description of a system is equivalent to another or that a description of a system satisfies certain properties.

I certainly believe this work is very relevant to more practical considerations as well and indeed Robin's book contains many examples and how it can be used for modeling realistic situations. It covers the most of the computational aspects of the object oriented paradigm. Objects can be viewed as processes, the interaction with objects by invoking methods can be seen as a particular case of communication with an environment, and the idea of encapsulation can be modeled by restricting the visibility of operators.

The second paradigm is that of typed functional programming languages. Functional programming languages are declarative languages based on theoretical foundations from the lambda calculus and logic. They embody work on the foundations of type theory, type inference systems, and subtyping. One of the motivations has been that subtyping should provide a model for one aspect of inheritance in object oriented languages.

Over the last couple of years some rather encouraging progress has been made in this area. Let me mention two developments here, one by Robin Milner and his colleagues. This is work on the pi calculus or the calculus of mobile processes. There is a long paper by Robin and some of his colleagues and also a more recent paper on functions as processes which appeared in this year's ICALP conference.The basic step is to extend the CCS paradigm to be able to pass the names of ports or channels in communications. This gives you a way of formalizing systems or networks of processes which can dynamically reconfigure themselves. One process can pass the access to another process to a third party.

Another development in this area I would like to mention is Ben Thompson who has just written his Ph.D. with me. He's taken a somewhat different approach, unifying the Lambda Calculus and CCS in a more direct way in his calculus of higher order communicating systems. CHOCS, which allows sending processes as messages. One of his examples is

an object oriented language and he translates the problem in a rather nice way into his CHOCS formalism.

This recent work gives encouragement for progress. However, it addresses the computational aspect of functional programs without really talking about types. The type aspect of functional programming is a key ingredient in properly understanding inheritance. In my contribution to the workshop, I addressed the question of finding a synthesis of type theory and concurrent process theory. I will mention some of the ideas. The particular work I described is written up in a paper on computation and interpretation of linear logic based on a recent system called linear logic developed by the French logician, Jean Girard.

What's exciting about linear logic is that it carries the scope of logic right into the heart of some of the key engineering issues which deal with efficient structuring of computations. It takes resources and the way that these are used in the processes of logical inference seriously, making them visible and part of the logic. Computationally significant operations should be made visible and not just taken for granted. For example, the lambda calculus bought has an underlying theory of substitution which is appealed to for free. But anyone who has implemented a functional language will know that substitution is anything but free from a computational point of view. Making things like substitution part of the logic is one of the main thrusts of linear logic.

To conclude, I'll mention some things that came up in the course of the workshop. I came away with a desire to understand better the relation between actor formalisms and the communicating processes. We now have tools developed both by the people in the actor community and on the communicating process side to get a much better understanding of what those relationships are. This will help to build a bridge between some of that work and the object-oriented concurrent work. There are some interesting questions do with identity. Identities of objects are a main topic and a known issue in object oriented languages and they are much harder to make visible in the communicating process world where the things that have identities are in fact the ports or channels of the system. In actor formalism ports are uniquely correlated with actors, so that actors have unique identities.

Another point is to do with inheritance and concurrency. This can be formulated in terms of meshing subtyping with the idea of attaching computational interpretation to logic in a way that I mentioned. Another point concerns what theory and practice might have to say to each other. Practitioners want to have formalisms which are very expressive. While that is at one level desirable, it is also undesirable because it lets you do things that you don't really want to do. On the other hand, there's a tendency in formalisms to prescribe how you express things, this is particularly so for typed formalism: they winnow out the wheat from the chaff, keeping typed things and throwing away the things that you can't type. Locating the right point for that trade off so you have sufficient expressive power but also sufficient structure and discipline is a rather interesting issue.

Another example of the mileage one can get from a good semantic understanding of formalism is static program analysis: the kinds of analyses that can be done at compile time and fed into the efficient compilation of programs. We've seen this in the functional programming community where the semantics of functional programming has been put to use in analysis, such as strict analysis, which has been used in optimizing compiled code and execution. It would be interesting to see how well those ideas could be applied to actor formalisms and more generally within the object oriented world.

Professor Gul Agha

Agha discussed the trade-off between structure and flexibility and the problems it raised for formal models of object-based concurrency. Here is a summary of his remarks:

I will coin the new buzzword COOP to denote Concurrent Object-Oriented Programming. Let me proceed to give a flavor of some of the fundamental issues in modeling COOP systems. I will distinguish three kinds of concurrency: divide and conquer concurrency (involving unrelated subcomputations), pipeline concurrency, and cooperative networks. Divide and conquer concurrency has a functional form, and furthermore, if the components may not have side effects, it allows unrestricted concurrent execution. Pipelined concurrency can be expressed in terms of streams which feed one process or object to another through a pipeline. Cooperative networks of processes can have arbitrary connections and constraints on the concurrent execution between objects.

Now let us consider a very simple example of divide and conquer concurrency, namely, the problem is multiplying a list of numbers represented as a tree. The leaves of the tree are numbers. The product of all the numbers in the tree may be found by multiplying the products of the left and the right subtrees. If a subtree is a number, we can just return it. Evaluation of the left and right tree products can be concurrent

because of the so-called Church Rosser property. The actor formalism can express this concurrency, but the concurrency cannot be explicitly expressed by functional programming because the needed continuation is history sensitive. Let us see why this is so.

To compute the tree product, two computations for the left and right parts are created. The answer to the one of these computation, which happens to be completed first, is sent to the join continuation. It must 'wait' there till the second answer arrives. When the second answer arrives the two answers are multiplied and the result is sent on its way. Thus the join continuation is a history sensitive object. The need for history sensitivity creates the necessity of state: a merge essentially amounts to having state. Once you see this operational structure you may modify it to return a zero if the first result obtained happens to be zero. Now suppose that the list is supposed to consist entirely of positive numbers. While multiplying the list you see a negative number which represents an error. You may wish to send an error message or perform an exception procedure. Now the join continuation may be programmed to wait until the datum has been cleaned up, or it may check the other numbers it receives. Things are no longer referentially transparent because the form of the action depends on the value (say seeing a zero or seeing a negative number). The referential transparency is lost to gain flexibility and efficiency. Representing the behavior of such a system requires a lower-level framework in terms of arrival order of messages, a behavior for processing the current message, a replacement behavior, and the creation of other Actors. This is described in my Actors book (MIT Press) or in the September 1990 CACM article.

Now the primitive actors provide flexibility and efficiency because we can explicitly talk about join continuations, communications, state, etc. However, representing even a simple recursive functions like factorial in terms of pure actors is messy—it involves many low level actors and continuations. Although the actor formalism is modular and does not require tracking the state, it is still too low level for practical programming in general. One response is to say that this is simply a price you have to pay for being able to optimize things like multiplication by zero. A more reasonable position is that you should not pay the price of having to specify low level programming details unless you really need it. Thus we should be able to use abstractions, in this case functional ones, in other cases, constraints, multi-party interactions, etc. for ease of high-level programming.

Thus, we can use abstractions to permit us to hide the details of a system but if we need to, we can reify the underlying implementation in terms of primitive actors, expose and manipulate its structure. For example, we can make the control structure explicit by bringing the join continuation up to the level of the programmer, and allowing her to manipulate it. Conversely we can reflect the manipulated structure, installing it in the underlying implementation in order to dynamically 'subvert' the meaning of the original program so that it means something different. Now, as you can observe, this creates some interesting problem for statically determining types for the system types.

Let me go back to a comment made by Abramsky about having enough structure so that flexibility doesn't drive you insane from the chaos that it creates. My response to that is that we need a flexible structure combined with structuring tools. We do not want a language to impose a discipline because there are many possible disciplines, each appropriate in a different context. A low level language provides flexibility to build different kinds of structures, while preserving freedom of action to conform to the discipline of any world that may be chosen. Furthermore, a reflective semantics allows a language to represent and manipulate its own semantics—thus adding greater flexibility. This is in contrast to the point of view which suggests that programming language designers should find the single right balance between structure and expressiveness? So let me begin by asking Samson what his reaction to this is?

Discussion

The discussion focused on a number of issues including: rigidity versus flexibility in language design, inheritance and synchronization, megamodules, methods for combining components, implementation mechanisms, the use of graph theory, and the implications of fine grained concurrency. An edited transcript of the discussion can be found in the Proceedings of the Workshop being published separately. Finally, Professor Kristen Nygaard of the University of Oslo was invited to make some closing remarks.

Professor Kristen Nygaard

I found this panel confusing—because of the way people talk about object oriented programming. I'm all for theory, it's extremely useful, but Abramsky's comments are a little like the tail wagging the dog. Object-oriented programming started as a phenomenon of computing processes. It

started with what happened in the computers. How could we could use it to model, simulate, and realize a vision of computing processes? How could we understand the phenomenon, including what people do and the processes of interaction between computers and people? We have a rich field of phenomena to understand and relate to. When people talk about what comes after the object oriented programming, for me it's similar to saying what comes after multiplication.

Our goal is to have a way of understanding concurrency and components. Components have properties that determine actions. There are no single actions, they are part of objects. Inheritance was introduced to model generalization and specialization. To blame inheritance because it deals with generalization/specialization is strange because it is generalization specialization. Of course there is a tree structure because generalization/specialization is a tree structure. It can be used for other things, and in the US it has been used extensively in order to achieve standard things such as code reuse. This has blurred the essential use of it. The problems of using inheritance in concurrency are not because generalization/specialization is inappropriate, but perhaps because we are trying to use it for additional kinds of composition for which it was not intended.

Another thing which also makes me a little uneasy is the talk about reflection, with systems changing structure as they go along. I think this is dangerous because we must make certain that we comprehend what we are talking about. Structure is something we impose on reality in order to grasp it and to deal with it.

In the last act of Hamlet, you have the play within the play. If you really look into that, you can see that you have two levels. To have the murder in the play within the play for directly the murder of the stage, namely the King. This is what we call absurd theater because it is meaningless.

Basic phenomena of information processing have structure, we have the process of making the structure, and of restructuring by composition. But we have certain limits on our ability and should be careful in going beyond this. The wrong thing about megaprogramming is not that we have large things and that we have to work because we have large things. It is the way many things are mixed together and dealt with as a large stew that suggests the wrong kind of largeness. Programming is programming and other things are other things and they all have to go together in order to make big things.