Panel Discussion

Theory and Practice of Concurrent Systems

Chairman:

Ehud Shapiro
Weizmann Institute of Science, Israel

Panelists:

William Dally
Massachusetts Institute of Technology, U.S.A.

Carl Hewitt
Massachusetts Institute of Technology, U.S.A.

Robin Milner
Edinburgh University, U.K.

Kazunori Ueda
ICOT, Japan

David H.D. Warren
University of Bristol, U.K.

CHAIRMAN: (SHAPIRO) Welcome to the panel on the theory and practice of concurrent systems. My name is Ehud Shapiro. I am from the Weizmann Institute of Science in Israel. I think we all enjoyed the very high quality and stimulating conference and I hope that this panel can match up to whatever we heard until now and contribute and be a proper ending for this conference.

We have here with us several distinguished scientists that represent a wide range of views and backgrounds on this topic of concurrent systems. I hope that we will benefit from hearing their views and hearing them debate and challenge each other's views, as well as answering the questions from the audience.

The way the panel will proceed is, first, we will have a ten-minute presentation from each panelist followed by short questions or comments from the floor or from the panel. And then we will open the discussion. The discussion will be based on questions either from the panelists or the floor, and after each question we will go in some round-robin fashion among the panelists. If people from the audience want to ask questions or make short position statements, not too long please, then you are always welcome and I will try and make specific points in the panel where we will stop and take some comments from the floor.

We have with us William Dally, who got his Ph.D. from CalTech, where he designed, among other things, the Torus routing chip. Prof. Dally is presently at MIT, where he leads a research group that is building the J-machine, a fine grain concurrent computer.

We also have Carl Hewitt who is a professor at MIT, where he also got his Ph.D. Among the major contributions of Prof. Hewitt is the Planner's formalism and Actors formalism for concurrent computation. And we will probably hear about it some more today.

We also have Robin Milner, who is professor at Edinburgh University and also an elected fellow of the Royal Society of London. Among the fundamental contributions of Prof. Milner are the programming language ML and the standard version for standard ML, which he is working on presently, and contributions to the theory of concurrency via the calculus of communicating systems CCS.

We also have Kazunori Ueda, who got his Ph.D. from the University of Tokyo. He is presently a member of ICOT. Dr. Ueda is a designer of the concurrent logic language GHC (Guarded Horn Clauses) which is at the foundation of a lot of research carried out presently at ICOT and in other places around the world.

The last panelist is David Warren, who is a professor at the University of Bristol. Prof. Warren made a significant and fundamental contributions to the techniques for sequential and parallel implementations of Prolog, and he will close the first presentation.

We are missing Prof. Geoffrey Fox from CalTech, who was supposed to represent the application side. And I am sorry about that. I expect as a natural consequence this panel will concentrate on the areas in which the panelists are strong at and they are now interested in, namely concurrent languages, models of concurrency, and parallel architectures.

So, please, Prof. Dally.

DALLY: I would like to start off by making a couple of observations about how people build parallel systems today.

There are two major approaches that I

have seen people take.

The first is to simply take whatever the latest microprocessor is and wire a bunch of them together with some network. This does not work very well for several reasons. The biggest is that the microprocessors represent the evolution of serial computers that over the last 40 years have been highly tuned to the sequential models of computation. They have stacks because you have LIFO activations in sequential programs. They have I/O architectures that are tuned for devices like DISCS, that have latencies in the tens of milliseconds and transfer blocks with many thousands of bytes. So, if you try to tie a communication network on through either an I/O channel or try to method it into the memory interface, neither works well. Because it doesn't match the needs of the communication mechanism.

The other approach is to take whatever your favorite model of computation is and wire it in the hardware in whatever seems the most obvious way, and perhaps you are fairly clever about optimizing and doing good engineering. But this also has problems. Because you've implemented just one model of computation. And a few months later you may decide to change that or you may discover that somebody else has a program that is written in another model of computation.

These two approach reflect people who start from what the hardware is, or people who start from what the software is, and directly implement that rather than deciding what the right primitive mechanisms are, that means middle between hardware and software.

If you look at many different models of computation, the actor model that Carl is going to say some about, the data flow that we have had some talks on here in the architectural track, shared-memory which I think includes logic programming languages that share variables and communicate through them, and data parallel programming is exemplified by Dr. Waltz's talk this morning—they all have three requirements of the underlying hardware. They need communications. Machines are physically distributed whether the model of computation is or not, and so you have to move information from one point in the machine to another.

They need synchronization, you have got to decide when the task should execute based on the desirability of producing its result and availability of the inputs that it requires, and you need to have some naming convention either to name cells in the shared memory, to name synchronization points in a dataflow program, or to name objects in an actor program.

The challenge for architects is to define the right interface. That is what the architects do. Computer designers implement the interface. The interface is coming up with mechanisms for communication synchronization and naming that are both efficient to implement and easily support most of the proposed models of computation that people have. And in fact the early results we have indicate that it is possible to do this with performance within a factor of two or three of a machine that is hard wired for one particular model of computation.

I would like to give you a couple of examples of mechanisms that are implemented in the J-machine that we are currently building at MIT in collaboration with Intel Corporation. One of these is fast message handling. This means more than a fast network, the technology that I developed when I was at CalTech. You can now build a network that will deliver a

message across the diameter of the machine of several thousand nodes in just a few microseconds. That is only part of the problem. Message delivery from the mechanism point of view includes initiating the message SEND, allocating storage for the message when it arrives at a node, and synchronizing tasks for message arrival. So, we have to do the four things indicated here. We have experimented different mechanisms for sending. We have chosen the SEND instruction, because it is proper compromise in our case between expense to implement and efficiency of communication. With it we can dispatch a six word message across the diameter of the machine for the cost of executing three instructions. This takes about 200 nanoseconds. The message then goes through a network which delivers it to the destination. Many proposals have been forwarded for clever networks that will do load balancing in the hardware or adaptively route to avoid contention bottlenecks. After evaluating many proposals we found the most critical problem is to reduce the overhead in the network, to make the network very fast. It became a matter of more throwing features out of the network and concentrating on the engineering, to the point that there are only a few gate delays between the message entering a node and the message leaving.

Once the message arrives at a destination, it has to be put somewhere. This is a storage allocation problem. Sequential architectures have been specialized to allocate storage for activation records and stacks. Concurrent architectures need specialized storage allocation for concurrent activation in message queues, and things of that nature.

Once a message arrives, the problem becomes one of synchronizing the arrival of that message with the computation going on at the node where it arrives. So, if we think about message queue being the box indicated here, from one end we can think of it as taking messages off the network. But from the processor's point of view, it is a ready list. Every message can be thought of as corresponding to a task that is ready to run. Whatever the task is at the head of the ready list needs to be dispatched. This dispatch, if the mechanism is chosen appropriately, can be done in just one clock cycle with one fetch from memory. The dispatch creates thread of control by forcing an instruction pointer and an addressing environment by setting the segment registers. Out of this mechanism, which is as inexpensive as you could expect, you can then build other things like the Snoopy cache protocol suggested by Prof. Warren in his talk in this conference.

The handlers that get dispatched by this process can be from simple functions such as updating the state of a line in a Snoopy cache and dispatching further messages to handle that protocol, or they can be grown into full processes by opening a larger addressing environment. In the J-machine operating system, growing a message into a full process takes less than a microsceond.

The other synchronization mechanisms needed to support these models of computation include some synchronization on data presence which can be accomplished for instance by having a tag that indicates that data is not yet present. An attempt to reference that data will suspend the task.

The final mechanism that I am not going to talk about, since I am limited to ten minutes here, is naming. I will say that many things that you see in a specialized architecture, like the wait and matching store and data flow machine or the set of special caches that Prof. Warren referred to

are special cases of generalized translation functions. What you want to do is to associate some key and produce from that piece of data. You can take for all of these models of computation the requirements for naming particular locations or naming particular elements of data and try to refine from that what the most primitive mechanisms are to implement those models. One set that we have chosen is to have the general translation mechanism to transit the key into data and to have some mechanism for protecting storage. We have chosen segmentation.

The key in picking these mechanisms is not so much to come up with the mechanisms that are most arrogant or most directly represent model or computation or algorithm that is being solved. And it is quite easy to propose a network that specializes for particular algorithm because the network captures the logical interconnect of the algorithm. But it is usually more efficient to pick a general network and map a logical network of the algorithm onto that. And often you get even better performance because you can then share communication resources between different logical links in the logical communication network.

The key challenge in choosing these mechanisms is to pick mechanisms that can be implemented with little overhead. In the J-machine the communication mechanisms I have described formatting a message and sending it, delivering it across the diameter of a 4000-node machine and buffering it at the destination for a six word message in the end, takes about two microseconds. This is about two orders of magnitude better than in the connection machine model 2, and about three orders of magnitude better than Intel IPSC-1 hypercube.

The synchronization mechanism is a single clock cycle to dispatch a primitive handler and less than one microsecond to create, suspend, destroy or resume a task. One aspect of efficiency is this overhead and also the interpretation overhead of implementing the models of computation in this mechanism.

Another aspect of efficiency is how well our hardware makes use of the underlying resources it is built out of. What is the cost performance of our architecture? This is a true measure of efficiency for parallel machine, not processor utilization as is commonly used in the literature. If I am building a machine that has N processors and it takes time T-1 to run on one processor and time T in N processor, the measure of the cost performance of the machine is the ratio of area of 1 to the time of I to the area of N and the time of N, not just the ratio of the speedup divided by the number of processors.

Let me leave you with one final thought, which is that the last formula it often turns out that the efficiency is greater than 1 for building a parallel machine. And the reason for this is that it is not expensive to build a powerful processor, and the technology that we have when we are implementing the J-machine in, the processor takes approximately 1/10 of the area of the chip. It is a fairly powerful 10 MIPS 36 bit CPU. The chip is mostly a memory chip. In fact what a fine grain concurrent computer is is a machine that has a very small amount of memory per processor, not necessarily a machine that has a weak processor, a single bit processor, or something of that nature.

So, if we think of these memory chips as being the jellybean parts of the 90s, then what we can look forward to is from defining the right general purpose

mechanisms that we can implement many models of computation out of, and coming up with efficient implementations of them, then we can build jellybean machines from these jellybean parts. Because many models of computation can be mapped into the mechanisms, we can have transportable parallel programs. People will no longer write a program to run on just one machine. Instead, they will be able to move a program from different implementations that will all support the same mechanisms.

Thank you.

CHAIRMAN: Thank you very much. Are there any comments or questions from the panel or from the floor? We have a couple of minutes.

WARREN: A brief question. What do you feel about the ratio of the amount of memory with (the size of) each processor? Is there some ideal amount of memory that a processor needs in such a machine? Or how do you determine it?

DALLY: Okay, this is a very good question that many people have brought up. If you are familiar with the rules of thumb that Gene Amdahl proposed in the 1960s, he, working on the traditional sequential processor architecture, said that to keep the processor happy you need to have at least one megabyte of memory for each MIPS of performance. If you look at a jellybean component such as this, you may have only about 64,000 bytes of memory in a 20 MIPS processor, which would seem to violate Amdahl's rule quite a bit. The difference is that he was looking at a machine where the amount of memory required was determined by the performance of the I/O system. He needed to have enough memory so that the processor could keep busy while swapping further pages of memory in from a slow disk. The real figure of merit is how much memory is available within a certain time delay when the processor needs it. So, if we implement an efficient communication mechanism and use a naming mechanism to provide a logical virtual address space, even though the memory is necessarily physically distributed, you can keep a processor busy with a very small amount of memory locally. We have been experimenting with amount between 10^4 and 10^5 bytes. And it seems that for many algorithms that is more than sufficient and that the real critical number is not the amount per processor, but the total amount in a machine. You need to have a sufficient amount of memory for the problem fit in the machine at a given point in time so that the problem does not become I/O bound.

QUESTION (Floor): I have a question. What is the ratio of local access time versus global access time of your machine? How fast is the global access in J-machine?

DALLY: Okay, now when I am speaking of our machine, I am speaking of a particular implementation. It is not a fundamental limitation in any way. To do a local access in our machine takes 100 nanoseconds. To do a global access, round trip sending a message having the read or write message handler at the other hand perform the memory read or write and send the message back, takes on the average 2 microseconds. That gives a ratio of 20:1.

CHAIRMAN: Thank you. We can have more questions during the discussion. Please, Carl.

HEWITT: What you have just seen is a

good illustration of something I wanted to talk about, and that is the difference between the sequential computation and concurrent computation. In concurrent computation many things are happening at the same time. Sequential computation doesn't have the problem of having to watch out for somebody else out there. I would like to take my theme here the principles that lie behind some of the work that we done on the actor architecture.

I would like to talk about having two objectivies.

The first objective is ultra-concurrency which is implementing concurrent systems whose concurrency is limited only by the laws of physics. This is the kind of thing that Bill Dally has just talked about—that we are actually getting closer and closer to it—the idea of ultra-concurrency. "Well, that is going to be the limit of concurrency?" The limit of concurrency is going to be the laws of physics, and also the size of your pocket-book as well. But the fundamental limit is of interest since some people are willing to pay to go all the way out to the outer envelop of the performance curve. So the limits are the laws of physics and the second objective is robustness. What is robustness? Here I would like to tantalize you by saying that robustness is different from reliability.

Reliability means completely repeatable. The system always does the same thing the same way and gets the same results within some kind of tolerance. Robustness is more general than that, in terms of trying to get something that goes beyond things where you can only use reliability.

Ultra-concurrency imposes stringent requirements. It's performance should be limited only by the laws of physics, which requires good engineering, like the kind Bill Dally talked about. In addition, ultra-concurrency must be maintained. It is not suffcient to set up the system go fast. Ultra-concurrency has to be maintained dynamically as the computation is on-going. Ultra-concurrency requires that the shape of concurrent computation has to be adaptable to the shape of the application, has to dynamically conform.

We have developed Actors as a universal concurrency primitive that has the capabilities that are required for ultra-concurrent computation.

Now, one important idea here is the Actors are really a mathematical model with corresponding implementations. They are not a particular programming language. For example, one of the students in my laboratory prepared an actor implmentation of a full Guarded Horn Clause language on a concurrent machine. So, programming language independence is a very important notion that I think leads to its adaptability and facilitates our ability to avoid certain linguistic conflicts and even provide a basis for addressing linguistic conflicts. We also use Actor to specify and reason about the properties for concurrent systems. Instead of giving mathematical specifications, the usual practice in Computer Science is to give mathematical specifications which have to be proved. For example in a corporation has an auditor suborganization who checks so that the corporation's financial system meets certain requiements and specifications the financial system is supposed to have. The Actor approach is to provide another concurrent system which can check up on the first! Because the only way in the real world that you can get specifications to hold is by creating mechanisms that work at enforcement. So, not only are Actor systems used for implementation, but they are also used for specification as well.

Bill Dally alluded to the need to interface to these things. He was talking about building a multicomputer that has both special purpose processors and general purpose processors. What we need to do is to specify a communications interface. Software systems build down to the interface and are grounded on it. Hardware systems build up to the interface using gates, memories, processors, arbiters, etc. The communication interface in-between, needs to be specified to make that kind of interconnection possible. Actors provide exactly this kind of interface. Such an interface provides a number of advantages to actor systems. One is the machine transparency you gain from this kind of system, as it goes from multi-processor to multi-computer, from local area network to wide area network. The interface provides the capability to have various different kinds of speical processors on the systems. If enables new systems to be connected. A computation is not predetermined as to how big it is going to be. The new and open system principle is the new and later computers can be connected into it. The communications interface facilitates ultra-concurrency by decoupling physical properties from ability to communicate. This provides a kind of mathematical mechanism to specify that it turns into an engineering specification.

Concurrent computation is much more interesting and quite different from sequential computation. One of the most important differences is that concurrent computations are indeterminate. Indeterminacy means is just plain "not determined" by anything about what the outcome is going to be. The asychronous operation of actors result in a certain necessary indeterminacy. No matter how much information is collected ahead of the time we still may know what outcome is going to be. Suppose we have an account in Chicago with $60, and in Tokyo attempting to do an electronic fund transfer to withdraw the money, another party in London attempting to withdraw $50, and someone in Boston is getting that $50. Well, we can say that one of the parties is going to be disappointed. Furthermore no matter how much you know about the structure and initial configuration, the operation of the electronic funds transfer system is indeterminate. Conflict provides a macro analysis difference. Various parties operating concurrently in the world work at cross purposes with incompatible courses of action. In many cases they didn't know ahead of time that there would be a conflict. So, we might have an agreement among the parties that each one agrees not to take out the last 50 dollars in the bank. But, lo and behold, there is the cooperative organization, they do take up the last 50 dollars in the account, because an emergency arose which required use of the funds before the other parties could be notified.

This finally leads me to my definition of robustness. Namely, "robustness" is keeping commitments in the face of indeterminacy and conflict. And this is how it differs from reliability: to be robust a system has to be more resourceful. If the first way doesn't make it, then it must be able to assess the situation and decides what to do. So, robustness is what we need interested in analyzing. Indeterminacy and conflict are always present. But I haven't explained what commitments are. It is one of the fundamental tasks for the theory of concurrent systems to define what commitments are to elucidate their structure and say how they are related.

The first thing I want to do is to talk

about discovery of a way that won't work. And this is bound to be controversial in these parts. I see several of you scribbling away getting ready for a counter attack!

Namely I want to claim that logical deduction has two fundamental problems.

The first fundamental problem is that logical deduction is too strong: an axiomatization of any "real" situation will have inconsistencies. The inconsistencies arise as a result of various conflicting commitments that have been made.

Not only logical deduction it is too strong; it is also too weak. If you think back to the example of the shared checking account where several parties are trying to withdraw money from the same account. One of the parties is not going to withdraw the money that it requests. That conclusion we can arrive at deductively. However, we cannot deduce which will get the money! You cannot deductively concluded the steps of. And in terms of our message passing systems, a concurrent computation has partially ordered events that keep intersecting back in and forth, as opposed to the linealy ordered events of sequential models. The interwining of the partially ordered events makes it impossible to use logical deduction to implement concurrent systems. And, therefore, I would like to claim that concurrent computation is not equal to deduction plus control. No matter what controls are put on the deductions the required decisions cannot be deduced.

The next step is to draw implications for the field of computer science. Computer science is a new field of research which is still in search of foundations. One approach (which I call algorithmics) to deal with this by attempting to ground computation and computers in mathematical logic. The algorithmic approach is to give a precise specification of a task that has to be performed, including properties of the environment in which is operates, and develop the code to perform the task alongside the proof of correctness, and to carry out that proof that the procedure meets the specification. However, precise specification can only be developed for mathematical domains. Rigorous specifications for open systems always have inconsistencies because of conflicting commitments. So, basing computer science purely on algorithms won't work.

With respect to GHC, the good news is that it is not logic programming. If GHC were logic programming then, it would be stuck inside algorithmics. And algorithms, only work for mathematics and not open systems. GHC (like ABACL, A'UM, PARLOG, etc.) is not logic programming. Furthermore, it is a good thing that it isn't because otherwise it wouldn't have the power needed to do their tasks.

So, in closing I would say what we need is new foundations for computer science. I don't have time to talk about this much today because my time is up. However, I would like to mention the kinds of concepts that we are going to need in our new theory of computer science in order to be able to make systems to operate in the real world. We need to know things like trials of strength. We need to know when one particular party is making a commitment or has a responsibility. We need to know about authority, and how authority is power authorized by other authorities. We need to know about things like conflict and cooperation, we need a new notion of representation, where one party of the system needs to represent something to another party of the system. Previously from AI we have had psychological notions of representation which is the correspondence between a structure in the mind of an

intelligent agent and the state of affairs out in the world. Such Artificial Intelligence definitions don't carry over into cooperative, competing social organizations. So, we have to get new definitions and make a new science based on the sociology of science and organizational theory, as a supplement to the previous psychological notions. In this way we can develop the kind of foundation that we need for large scale concurrent computation.

CHAIRMAN: Thank you very much. Carl gave us sufficient materials to discuss for two hours. But I will now just take one or two comments from the panel if there are any and then we will continue. Any comments, questions? Okay, we will proceed right on to Prof. Robin Milner's presentation.

MILNER: I think Carl has said a lot of my piece for me. I am going to take away the sociology and add the algebra.

I think my first thing is that we are looking for a descriptive science. And we are looking for something which doesn't, like sequential computing, cover a part of computation. It is actually covering all of computation and actually a good deal more besides. Now, I want to talk about three aspects of the descriptive science, which we shall probably never find in the ideal. The first is universality, the second is primitives, and the third is structure. (Fig. 1)

The first thing which I believe was first noticed by Carl Petri is that we shouldn't have three levels of description. In some way the von Neumann model forced levels of description upon us, because we learned how to describe the behavior of the machine in a sequential way. That is we learned how to write sequential programs. Programming is just description. It is just

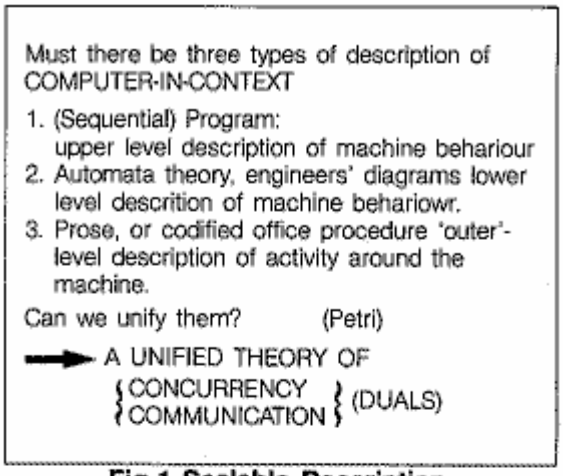


Fig.1 Scalable Description

describing something that doesn't exist yet. And it happened that we had a very limited mode of description and that was the sequential programming. And so it was quite natural to us to have something else going on below it, which was, say, automata theory describing what went on underneath and something which went on above it, which of course would be different again, no surprise. In this case it might be just prose or it might be some more structured kind of narrative, but it would in a different style. Since we are already committed to something special at the programming level, it wasn't really odd to have all these levels of description. So, now how do we unify them? Well, it is a tautologous thing to say that in doing so we seek a unified theory of concurrency. What I want to say also is that communication and concurrency are virtually the same subject. Concurrency without communication is boring, communication without concurrency cannot exist. So, those two things are precisely the same.

Now, to proceed to primitives (Fig. 2), I still don't believe that you can get around this idea, that if you take different primitives for communication, say for example

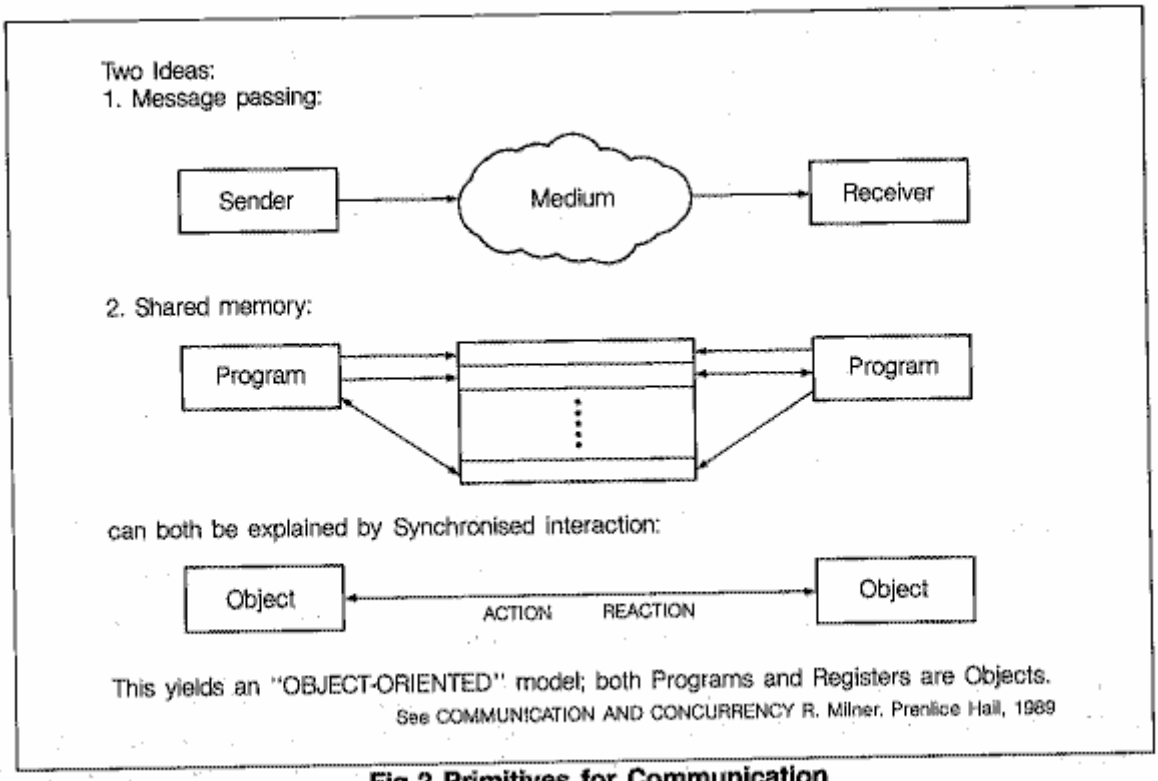


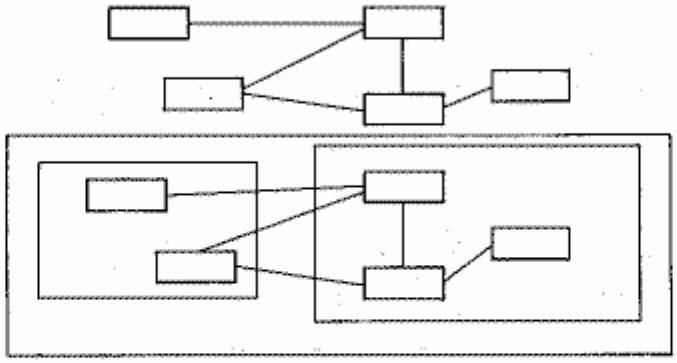
Fig.2 Primitives for Communication

things which we know so well the shared memory model or the message passing model—they are different of course because messages have identity, whereas the values that are set in memory don't have identity, you can read them more than once, you can send them more than once, and so on—but if you take those things, you inevitably have two kinds of objects in your world. You have the sender and the receiver, you have the active object and you have the thing in between which might be the message or it might be a register. And you have these arrows between these heterogeneous kinds of objects. So, if you say let's just have one object in the world, then you cannot, as far as I can see, get anything else but synchronized communication as a primitive. Now, that of course people would immediately want to disagree with, particularly if I say it so forcefully, which is one reason for saying it forcefully. One usually says things forcefully if one isn't quite sure. But I found that that for me goes on working well. And really that is nothing more than saying that we are looking for an object-oriented model. "Object-oriented" is a conveniently loose phrase, but it tends to mean things existing independently and battling each other rather than being parts of one and only one organism.

So programs and registers are all objects. And this makes a surprisingly smooth model. As a matter of fact, I think all I have been doing for many years is trying to provide some of the mathematics for Carl's actor theory.

Thirdly, structure, which I think is an extraordinarily rich subject (Fig. 3). Because we can mean so many things by it. This is where I think we are justified in

We must explain, in one theory,
1. Fiexd (heterarchic) structure of interacting agents (e.g. machine components)
2. Hierarchic description superposed on this:



3. Dynamically growing/shrinking structure (e.g. procedure activiations in a parallel programming language)
4. Mobility among agents (e.g. jobs in an operating system processes among processors)

Fig.3 Hierarchic/Heterachic Structure

going beyond Petri net theory which unquestionably was the first true theory of concurrency, namely looking for all the handles on the structure that we can get hold of. There are really two different kinds of structures which look as though they are the same. One is the physical structure of the system, which is flat, heterarchical, merely says something about how it is interconnected. And that occurs with virtual systems and with real systems. But the other is much more elusive kind of structure. It is the structure we impose on the system in order to understand it. Anyone who is trying to understand a heterogeneous system operating in parallel knows that you get a better decomposition of the system for some kinds of understanding by associating the elements differently from another kind of understanding. So, we impose structure by resolving into subsystems in certain ways. And that is something which we must be able to do in our theory.

There are two further aspects of structure. One is that in virtual systems which must be covered just as much as real systems by a theory of concurrency, they grow and shrink. Procedure activations in a concurrent programming language, which are very, very hard to understand, are an example of that. But if we can understard those things semantically then we release a lot of power.

But more subtle than that—and this is something which Carl has been dealing with in his actor model all these years—is that agents shift their connectivity. And that is different from growing and shrinking. In growing and shrinking they maintain their neighborhood structure, just within themselves developing some more new components with new neighborhood structure, but not changing existing neighborhood structure.

Now, there are two rather subtly different examples of changing connectivity. One is jobs flowing through

an operating system; you can think of those as software. Then you can think of the shifting association between the processes and processors. Now, one of the challenges for a model is not only to be able to describe hardware and software at the same time but within the very same expression to denote a process and a processor and that association between them which represents the fact that the process is running on the processor. And it is not that that is our single aim, but that is such an interesting and challenging aim that, if we find the theory can do that, then it will probably be able to do a lot of other things as well.

So, that is all I want to say.

Just three themes about a good theory of concurrency. One is universality, the second is seeking the right primitives, and the third is the structure. And we don't necessarily find the unique answer to these three things.

That is all I will say at the moment. Thank you.

CHAIRMAN: We have time for a few comments or questions.

HEWITT: Robin and I have been having this discussion for years, but I still worry about synchronous communications. It seems to me the only way you can have synchronous communication is by convention. If you have the synchronous communication then you have to have two different names for the same event. What this means is that if I leave this room here, then in order for my leaving the room to be synchronized with my entering the hall outside, it has to be that entering the hall outside is the same name as the name for me leaving the room, because this is one event there that has two names. But then I worry about, Robin, your use of registers. Like I am the programmer over here and the register is over there. Then I have got two things. One is I push off the message here and there is the second event where it arrives over there, and now it looks like I don't have two names for the same event or that cannot possibly be synchronous.

MILNER: In that case, my answer is that they are not the same event but there is some intermediate medium which takes part in first one event and then the other. That is all, we agree on that, I think.

CHAIRMAN: Any other comments?

(FLOOR): I guess I have a comment on the same line. If you go to a synchronous communication models, the basic problem you have is whether you can dynamically create the agents to provide an unbounded buffer or not. Otherwise you have a problem with things like recursive computation, where you can not guarantee the delivery of communication, unless you have an unbounded buffer or potentially unbounded buffer.

MILNER: Yes, so, you must be able to create new agents during that execution. That is true.

(FLOOR): And an unbounded number to provide the same kind of guarantee.

MILNER: Certainly in the model I have been dealing with, I have always been able to create new agents, and I left out the ability to shift the connectivity of agents, which is a different thing. Because I wanted to deal with the algebra in a way in which I knew how. Now, I think we had slowly developed and I think one can proceed toward the idea of mobility. But I entirely

take your point. But I think it starts...

(FLOOR): Right, I think that is the basic difference between the power of something like CSP and CCS. Once you can create these agents dynamically, then you can define or buffer these synchronous communication in terms of synchronous communication. However, if you are not able to create an unbounded number of agents, fast enough in some sense, then the power of these two systems is quite different.

MILNER: Yes, agree.

CHAIRMAN: Bill, you have a comment?

DALLY: Yes, Robin, you proposed this synchronous interaction as a fundamental descriptive primitive because it captures both message passing and shared memory without hiding state as message passing does or hiding communication as shared memory does. Do you think this is something that is also prescriptive in terms of how machine should be constructed, or do you think the two are quite different?

MILNER: I am very glad for that question. Because one thing that strikes me is that primitives for mental model aren't necessarily the same primitives as the ones you want for construction of a machine. I am sure there is a very good example of that, or many good examples in programming, where the idea of refunctional application is a beautiful primitive for a mental model. But you implement function application and you will find it is far from primitive on any machine which involves change of environmental kinds of things which seem for us in a mental model. We seem to be able to understand it and it may be just that because our mental machines have developed over a long time, so what is primitive for us is really not primitive for a machine.

So, I don't believe it is prescriptive but I believe there may be a happy agreement in this one case. I don't know whether you would agree about that. In other words, the synchronized communication is a good primitive idea for both the mental model and for the machine model. Would you agree it is good for hardware as well as for mental model?

DALLY: I think that in hardware it is difficult for the reasons Carl described to do that. Because you tend to bave hidden state flying our wires and things of that nature. And it is easier to deal with the synchronous model at least at some model, where you have hidden state in messages.

MILNER: Okay, then, this is another example of a mismatch of primitives, I believe. Thank you.

CHAIRMAN: Okay, we can continue on this point perhaps later. So, we will have Kazunori Ueda's presentation now.

UEDA: I would like to talk about the role of kernel languages in the FGCS project.

The outstanding feature of the FGCS project is that it took the middle-out approach, one direction going from the kernel language down to hardware and the other direction going from the kernel language up to application software (Fig.4). This approach was adopted as a hypothesis of the project when it started. I will explain why and how we designed on kernal language.

In the initial stage, we assumed logic progrmmig as the best framework for a

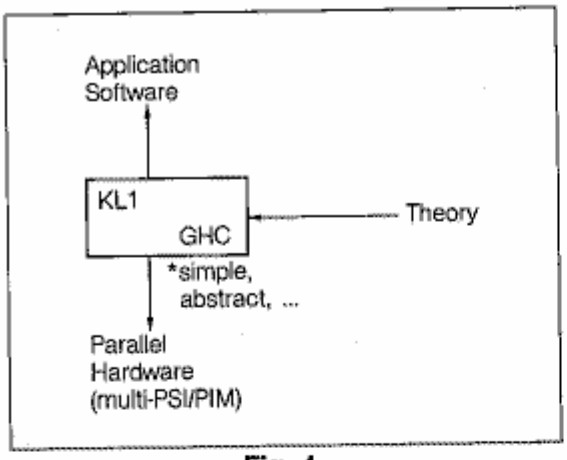


Fig. 4

kernal language and designed KL0 based on Prolog. And now in the intermediate stage, we have a basic language GHC and its practical version KL1. What is the role of the kernal language layer? I think it is not just a programming language. I mean, if it is just a programming language, then we can design any powerful language that is good for parallel implementation and also for application software.

However, a kernel language is the first thing we have in the middle-out approach, so it should stand by itself without relaying on ohter layers. Instead, a kernel language should be supported by sound principles or good theories. This is what I like to stress in may talk. Possible guidelines include simplicity or abstractness.

A kernel language must provide a framework of thought in which to clarify many different concepts and requirements. This layer should also reconcile theory and practice, and should connect applications and hardware. The kernel language layer has all these roles.

We are often asked whether parallel programming is difficult. I think a technically fair answer can be given only after making much more effort to create parallel software. The amount of effort so far is much less than the effort made for creating sequential software. We are too much accustomed to the sequential programming of von-Neumann computers, and this is causing a problem in putting the project into practice. In particular, we suffer from relaying on sequencing to ensure the correctness and the efficiency of a program, and the assumption of constant-time access has kept us from exploiting locality of programs.

So, one possible clue to solve these problems is to provide an easy-to-use parallel language, along with its efficient implementation. And this is exactly what we are trying to do now. We are encouraged by the success of the object-oriented programming, because process oriented programming and object-oriented programming are very similar. We still have to choose how to design the kernel language. There are two alternatives: one is to augment a sequential langauge, and the other is to design an inherently parallel language. This table compares these two alternatives. In the first alternative, control can be over-specific, but in the second alternative on essential sequentiality can be specified. An augmented sequential language lacks flexibility in that language defines the granuality. In constrast, a program written in an inherently parallel language expresses any potential parallelism. From programmers' point of view, the first approach allows programmers to rely on sequentiality, while second approach encourages the change of thought. So in the long run I believe that the latter approach is more promising.

Another concern is whether to choose a language without control or to choose a language with control. Here by "control", I mean control for the correctness of a program rather than control for efficiency.

So I think the difference to these languages is more a matter of formalism rather than a matter of abstractness, because, for example, GHC is a very abstract language with control. I think these two languages will be used at different levels for different purposes.

Let's see how these two kinds of languages should be related. (Fig.5) These figures look very much like the figure given by Carl Hewitt.

The left figure shows the usual case in our computing practice. There may be a good closed language without control and there may be a good theory within this inner framework. But we are living in this outside world and we have to implement somehow this beautiful language using some lower level sequential language. Curently there are almost no principles to guide us in filling in the gap between the inner framework and the outside world. What we are trying to do is to reconstruct the left figure into the right figure. The right figure we should try to find good principles and a language for interfacing between the inner framework and the outside world. In designing a good language for this gap we should carefully reexamine the current practice shown in the left figure. I mean that we have many problems, for example, in the use of Prolog systems. In Prolog, we make heavy use to extralogical features, but such features should be reconsidered and reconstructed by designing a good langauge surrounding the inner layer of pure Prolog.

My principle is that we should separate different concepts first. After that we could reintegrate the inner layer and the outer layer. But we have to do this very carefully.

This shows why we chose to expose parallelism. One reason is that we believe that the development of concurrent systems should be supported by many people working on various layers. I think that parallelism is too tough and too difficult to be considered by a small number of people working on a limited area of computer science.

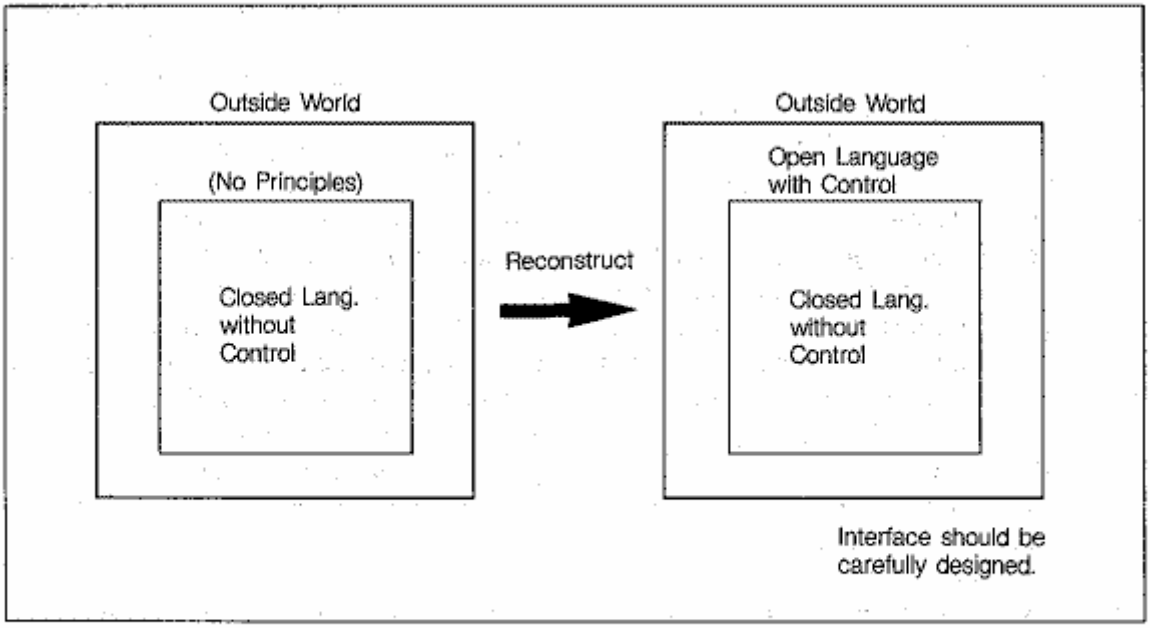


Fig. 5

Another reason is that the effective use of parallel computers require the accumulation of good parallel algorithms. I think this is clear when we see the current culture of sequential computation, which is well supported by the accumulation of sequential algorithms. I believe that the applications programers should have access to parallelism. I am not saying that all applications programmers should program parallelism. I am saying that they should be able to program parallelism if they want. Our approach is to encourage exploiting parallelism by exposing parallelism in a simple and abstract form rather than to confine parallelism in some layers.

The last two slides show the overview of GHC and KL1. These two languages are actually our theoretical and practical version of concurret kernel languages. Our first kernel language in the initial stage was KL0 which was based on Prolog. It was a sequential language, and was good for describing closed systems. But we had to make many ad hoc extensions to Prolog to write an operating system SIMPOS. We are now at the intermediate stage, and GHC was designed to be inherently parallel and good for describing open systems that may interact with the outside world. But still we have to step up to the next stage, because the theoretical version of our kernel language, namely GHC, excluded meta-level operations necessary for observing and controling parallel execution. Accordingly the practical version of the kernel language, KL1, had to feature meta-level operations in a rather ad hoc way to write an operating system PIMOS. In the next stage, we plan to reconstruct meta-level operations and reflection capabilities in a more elegant way.

Why did we choose GHC and KL1 as the kernel langauge? The reason is that GHC is a very primitive language that allows multiple views. First of all, it can be viewed as a process description language, but GHC is so primitive that even communication channels are not primitive constructs of the language but are programmed in the language. This language can be viewed also as a dataflow langauge and as a parallel assembly language. The design goal of GHC has not so much to do with logic programming, but GHC can still be viewed as a logic programming language amenable to declarative reading.

I am very interested in keeping the core part of the kernel languge as simple as possible, because simplicity means firmness or robustness. Simplicity reveals the essence of parallel computation. Simplicity also helps us to tell one thing from another. As Dr. Shapiro said, GHC may be too simple for some applications like systems programming, but it serves as the reference point for comparing different concurrent logic languages. We have too many things to consider at once, so now we are concentrating on basic things. As I said, discrepancy between GHC and KL1 still exist, but are much smaller compared to the discrepancy between pure Prolog and KL0.

Another point on simplicity is that it enables us to have useful, formal semantics, which is what I am now working on. Simplicity also encourages the development of formal methods for program anaylsis and transformation and optimization. These are the reasons why we have still two versions of our kernel language: a theoretical one and a practical one.

CHAIRMAN: Any comments or questions from the panel or the audience? Please step to the microphone, if you have.

TAKAYAMA (ICOT): I would say I am

too acustomed to mathematical thinking rather than the von Neumann style of thinking. As you know, mathematical thinking has little to do with parallelism. Do you believe GHC gives a good impact of mathematical thinking, for example to get the parallelism into mathematical thinking?

UEDA: What is the last point? Does mathematical thinking help in constructing parallel programs?

TAKAYAMA: Not really. Mathematical thinking is very limited. It does not have any concurrency. Maybe I hope it can be extended to a larger area of thinking which contains a sort of concurrency. Do you believe GHC gives a good impact or gives us a good hint to extend mathematical thinking to the theory that has concurrency?

UEDA: The only comment I have now is that anyway the concurrency issue is very new to mathematical thinking.

CHAIRMAN: Maybe we can continue this in the general discussion. Any other comment from the speaker? David, are you ready to give your talk?

WARREN: In listening to various position, I think thre are two concepts which get confused in my mind at least. I will call these concepts, for want of better word "parallelism" and "concurrency" (Fig.6).

When I think of parallelism, I think of the computer itself carrying out activities simultaneously, with the goal of making computation actually run faster. And then there is the other position representing an interest in "concurrency", which is looking at applications where the notion of communicating objects or communicating processes is intrinsic to the application, for example designing an operating system, simulating some complex process, an airline reservation system, et cetera. So, it is a question of whether we will be talking about concurrency in application or concurrency down in the computer. And those are really quite different things. So, for example, we can have a concurrent application running on a sequential computer. In fact most concurrent application, for example airline reservations systems, are actually run presently on large

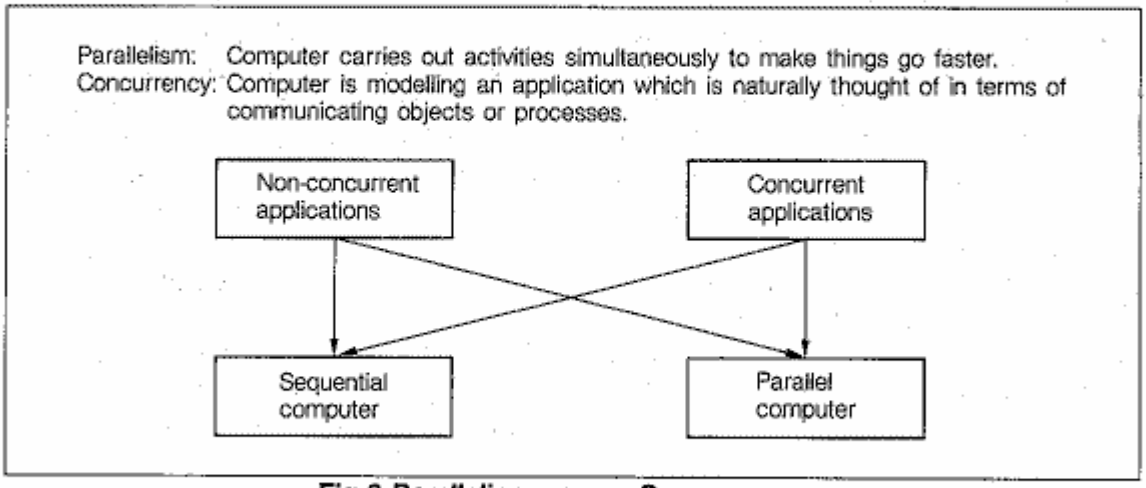


Fig.6 Parallelism versus Concurrency

sequential machines. One could also, in principle, run concurrent applications on parallel computers. But ordinary non-concurrent applications can also equally well be run on sequential computers or parallel ones. So, I think it is important to distinguish these two different motivations for what we are discussing.

My own principle interest is more in exploiting "parallelism" (rather than "concurrency"), with the goal of making computation go faster on parallel computers. But what I believe is important—and this is rather in contrast to Ueda-san's position—is that we should try to be making Parallelism invisible to the normal programmer. Programming is already, as we know, a difficult enough task without making it even more complicated by forcing the programming not only to have to think about his complex application but also to have to worry about parallelism going on inside the computer.

So, I think ideally we would like the parallelism to be invisible to the programmer. In fact what we normally think of today as sequential machines are often really parallel at the lowest level. For example, with pipelining in a sequential machine, at the very lowest level we have some degree of parallelism that typically only affects the microprogrammer who develops the microprogram to interpret the lowest level instruction set. And everybody else who sees the computer from that level upwards does not need to be aware of the parallelism within the machine.

I think we want to achieve that degree of separation in any parallel machine, even if we are talking about large scale parallel multiprocessors. So what we would like is that the programmer should be rather like an architect designing a building. The architect specifies the size and shape of the building, has a general idea of how much the building is going to cost, but he isn't concerned with the details of how the building is going to be built. That job rests with the construction company which is like the underlying computer system. That underlying system should be solely responsible of determining how many resources to operate in parallel to get the job done as quickly as possible.

Now, a mention has been made of different languages, whether we need sequential languages or parallel languages or what. I think we should classify the languages into, sort of, three broad categories—sequential, parallel, and declarative.

Sequential languages are the conventional von Neumann languages, and the parallel languages are more recent—I would rather call them conventional languages designed for programming parallel machines. In both these kinds of languages, we think about what is happening down below in the machines. They have built into them the notion of state and the notion of changes of state through assignment opertions. And the difference between the sequential and parallel languages is that, in the sequential language, the state is changed one step at a time, while in the parallel languages we may be changing state at many places at once. But fundamentally we are thinking of a program here as to how to specify what should be going on underneath inside the machine.

In contrast, the declarative language is looking more upward toward the application. What is it that we are really trying to do? What is our problem? How can we best describe that problem, and get it solved on a computer? And the examples of declarative languages are Prolog and GHC and in fact ohter committed choice

languages. All these languages are describing problems in declarative ways. We may be looking at concurrent applications, for example operating systems, and for these applications languages such as Guarded Horn Clauses have proved very appropriate. On the other heand, we may be looking at applications where concurrency is not intrinsic to the application, and there Prolog, I think, is more suitable.

The Chairman, in asking us to prepare for this session, one of the things he asked the panelists to discuss was how they envisage future "concurrent*" computer systems. So, let me conclude by giving you my view of what I would like the machines of the future to look like (Fig.7).

Starting from the bottom, the actual hardware, in the talk yesterday I described something called the data diffusion machine; I will just summarize briefly what that was. The aim was to have a general purpose multiprocessor architecture, which should be scalable to an arbitrary size, and which should support a notion of global address phase or shared virtual memory. But the machine is like a message passing machine in that it has distributed physical memory. And the machine is a hierarchical machine in concord with what Herb Simon was saying he though hardware should look like. This machine is completely general purpose, although it is motivated by looking at executing logic programs. In

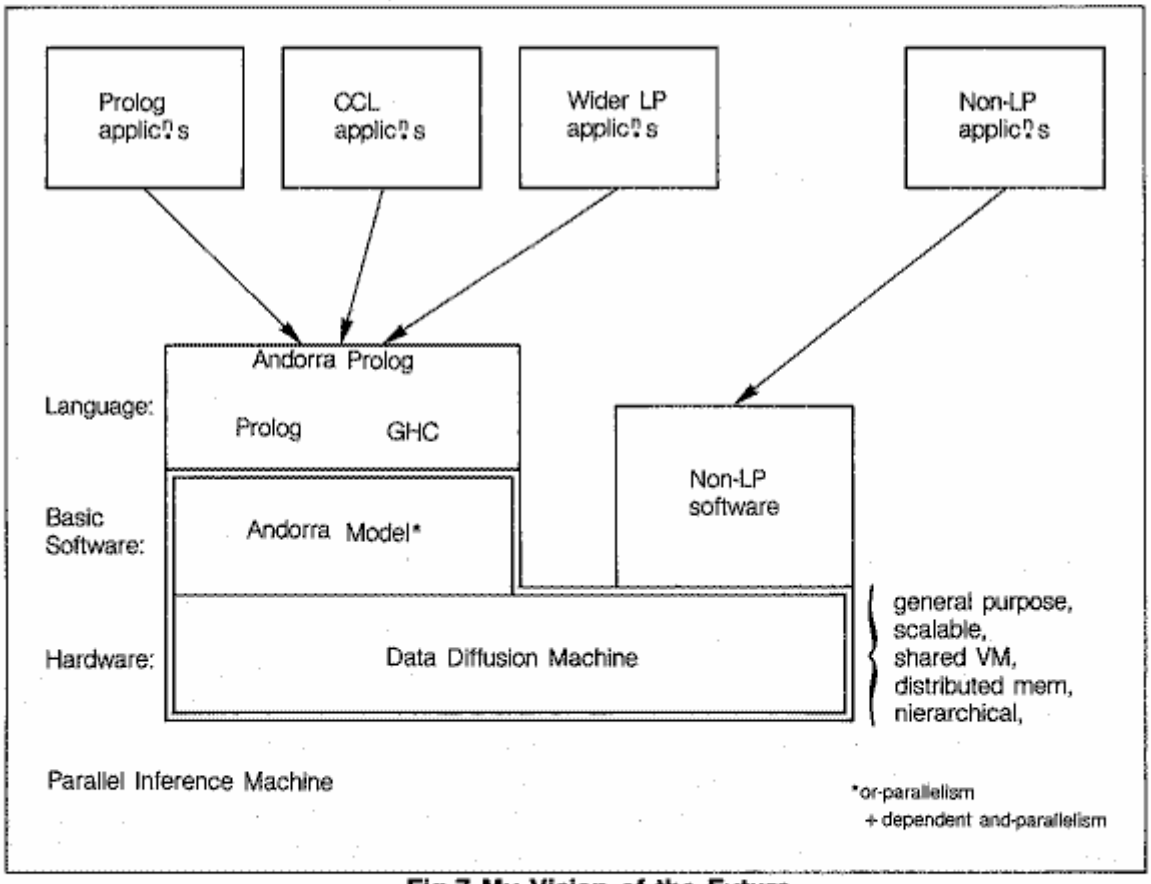


Fig.7 My Vision of the Future

*which I here take to include "parallel" computer systems

fact, it equally well could be used for any kind of software applications. I have drawn in this picture examples of non-logic programming applications passing down into software which is run on the data diffusion machine. In fact, today conventional shared memory machine such as the Sequent and Encore have software which could run equally well on the data diffusion machine. So, that is why it's a general purpose machine.

But my own interest is particularly in developing parallel inference machines. We turn the data diffusion machine into a parallel inference machine by supplying some appropriate basic software. The current thinking of what that basic softwre should be is an idea called the Andorra model, which was referred to briefly in the talk by Seif Haridi this morning.

The idea of the Andorra model is to combine or-parallelism with dependent (or stream-) and-parallelism. This Andorra model infact is an idea for extending some existing work which we have done on the Aurora system, which is an or-parallel implementation of Prolog. It is an implementation which allows the standard language Prolog to run on a parallel machine, with the same semantics as we get with running Prolog on a sequential machine.

With the Andorra model running on top of hardwre like the data diffusion machine, or indeed existing (parallel) hardware, we have in principle a parallel inference machine. What languae is to be the one to program applications on top of that parallel inference machine? I should stress that, as far as the user is concerned, the parallel inference machine is a black box and he is unaware of the fact there is parallelism inside the machine. So, we want a language which is appropriate for applications but which hides the underlying parallelism from the user. And such a languge, which I will call Andorra Prolog, is an idea that is still in a state of evolution. But it is a language which will exploit the Andorra model, the underlying computational mechanism. It is a langauge which will subsume both Prolog and committed choice languages such as Guarded Horn Clauses.

So, we envisage this language being suitable for running applications which now are written in Prolog. We can map these directly into Andorra Prolog and run them as before. Equally, applications written today in committed choice languages such as Guarded Horn Clauses could also be mapped into Andorra Prolog and run equally well on sequential or parallel machines. The user is unaware of whether the underlying machine has only one processor (and is therefore a sequetial) or whether it is a parallel macine.

But Andorra Proog is more general than either Prolog or Guarded Horn Clauses alone. And the interesting thing is going to be what Seif Haridi was talking about in his talk this morning—applications which can't easily be mapped into either Prolog or committed choice languages, but which Andorra Prolog makes possible.

I think my time is up.

CHAIRMAN: I think the kind of intellectual battle that is going on here, is getting clear. Unlike Sumo where everyone wants to end up on top, here everyone wants to end up in the bottom level, and wants the rival to be staying on top. Any questions?

DALLY: I have two questions, for you, David. The first is about what is best described in terms of a common saying,

about LISP, in that the LISP programmer knows the value of everything but cost of nothing. And in parallel machines most of the cost is in communications, and performance of an application is greatly determined by how well an algorithm exploit parallelism. It doesn't make sense to make tha parallelism invisible from the programmer.

WARREN: I think that (making parallelism invisible) is the ideal that one wants in order to run large applications. If one is going to develop some complicated natural language system, for example, I think the larger such a system is, the more the developer is concerned about correctly modeling the application. If he can run it faster on a parallel machine he will be very happy. But he is most concerned about solving his problem by correctly representing his problem. So, for that kind of application clearly I think he does not want to be involved in parallelism, if that is going to make his task more difficult.

Now, I think you are right in saying—and in what Ueda-san was saying—that we currently don't know how to map problems into parallel algorithms. So, I think, yes, in the short term probably we do want to be able to control how the parallelism is exploited inside the machine. But I think this is a sort of short-term goal rather than a long-term goal. The way I would see that happening is possibly by a variety of annotation to the language which would specify directly how the parallelism underneath is going to be exploited. That is what ICOT is proposing with the languages based on GHC, to actually control physically where this bit of computation happens. I think that is not ideal. Hopefully when we learn better how to exploit parallelism we can embed all that knowledge in the underlying system, and therefore get the system to take care of it for us and not to have to worry the applications programmer with these concerns.

DALLY: The second question is about primitives. I think Robin said it very nicely a function call assuming which is a nice primitive first to think about, but no compiler-writer today wants a function call instruction in a machine. By the same notion, a shared memory reference is a nice thing for us to think about. There is a lot of work going on underneath to implement that. Is there any reason why you think a shared memory reference needs to be hardware primitive?

WARREN: Whether it needs to be a hardware primitive or a firmware primitive, I think for basic application software we want it to be primitive, because I think the natural way to think of parallel computation is in terms of different processing agents, sharing data, working on that data, each agent working one step at a time, the problem data overall being manipulated several steps at a time. The notion of sharing data is really, sort of, the same as the notion of shared virtual memory, where you just view the virtual address as being a name for a piece of data. That is why I think the shared virtual memory concept is very important for writing nice parallel software.

So, whether that is implemented directly in hardware or in firmware I am not so arguing about. But I do think one needs an interface supporting shared virtual memory.

HEWITT: I wanted to ask David in what sense do you see GHC as being declarative?

WARREN: I think the notion of declarative language is that one can think of what the result of the program is in terms of the declarative meaning of the statements of which the program is made up. With GHC and the committed choice languages, programs tend to be perpetual processes, so they don't really produce results. They develop through time. But one can think of what they are doing through time in terms of producing conclusions which would follow from some intermediate set of assumptions. In that sense one can view them as correctly reflecting the declarative content of the clauses which make them up. But I think it is important to realize in logic programming that the logic programming language is not solely the declarative part, it is also the control part, it is how we use it. So, simply understanding the declarative content of your program will not tell you all you need to know about it. So we also have to reason about the control part of the program. It would be nice if we could get rid of the control part and just have to reason about the declarative part. But I think in reality that is not feasible. So we really want a nice form of control which we can also reason about well.

CHAIRMAN: Any other comments?

MILNER: Could I question the assumption that, so to speak, to see the results of a program clearly from the presentation of the fprogram, that should necessarily mean that it is inferred logically. There are many kinds of mathematics and I can't see any qualitative difference between seeing what comes out of the program as a result of mathematical analysis of something which isn't logical on the one hand, and seeing what comes out of the program as a result of logical inference. And these two things seem to be exactly on a par. And I don't think that the case has been made that logic programming has any priority here.

WARREN: All I can say is that, for me, looking at the logic program, it is easy to understand the program piece by piece using the declarative content of the program. It is easier to look at one little clause and to know what that clause means just from the declarative content of the program rather than trying to reason about the totality of some mathematical object. So, this is what I find in logic programming, which I cannot find in other formalisms. One can look at the program in small parts, see that each individual piece is correct, and therefore, conclude that the totality must be correct.

MILNER: Just to come back briefly on that, I believe that could be, because concurrency is something very new to us. And it may be that working with the right mathematics of concurrency, then we can develop the same feeling of naturalness as you now have for logic, and so we get a programming idea which in some sense perspicuous but isn't logical. Maybe a matter of training for us.

WARREN: I certainly think, with the use of logic programming for concurrent applications, the transparency of what one will say purely from the declarative point of view is a less all embracing part of the understanding of the program than it is for languages like Prolog. So one has to think more about control when one is thinking about concurrent applications than when one is thinking about non-concurrent applications.

UEDA: Professor Hewitt and myself maintain that Prolog is a good langauge for describing microtheories in his terminology, and GHC is a good language for describing open systems. Prolog has good, simple semantics. GHC also has fairly simple semantics. However, I don't think these languages and semantics can be combined in a straightforward way. In the case of Andorra Prolog, how are the semantics combined? Do you have any semantics for the integrated language?

WARREN: How would the semantics combine in the Andorra Prolog? I am trying to say that Andorra Prolog is still not totally defined. But from the declarative point of view the clauses are clauses. We all know what clauses mean. From the operational point of view, I think it is rather difficult to explain without expalining at great length what Andorra Prolog is. But if you want to write applications which are in the Prolog vein, then you will find the operational semantics is much as Prolog. If you want applications in concurrent style you will find the reasoning you will need to do is very much what you will do for GHC. Then there are the applications which involve both non-determinancy and co-routining, and then parallelism will really be on both. Sorry I have not answered your question very well.

HEWITT: Just a second. If you look at FGHC the thing I find mysterious is how to account for shared objects with a changing local state. Suppose I have a shared account in FGHC. How do I deal with it? I would like to make a withdrawal. I have a message for the account. So I assert that the first element of the stream is a WITHDRAW message with a write only variable that he can use to record success or failure. That is a mechanism like invoking a procedure in procedural language. The result comes back, either I got the money or I didn't, I can't logically deduce which was going to happen, and neither can anybody else. So that looks very, very operational. If I think about how I use this account, I assert that WITHDRAW mesages is the first element of the message stream for the account response variable. I wait on the response and get back, "Yes you have got the money," or "No, you didn't." The response didn't follow logically from what I did. This is not simply a matter of controlling deductions: it's problem of not beng able to make enough deductions.

WARREN: I think my view is that that is because the logic is not all you need to know about to find out about whether someting follows from what you did. Because a logic program in general is made up of logic plus control. In this case, where the control is controlling the concurrency, I think it is important to know what that control language is doing as well. But I think we had better let either Ueda-san or Shapiro discuss that question.

CHAIRMAN: So, I wil just continue with this. There is still a commment to David even though he is sitting. I think the classification of language into sequential, concurrent, an declarative is very useful. But I think you have misclassified GHC and the other concurrent logic languages. I think they are in the same family as OCCAM, CSP, et cetera. They also happen to have declarative reading of a much more complicated and different sort than declarative languages. And the declarative reading of concurrent logic languages is something like: "if the following is the

sequence of possible histories of a process, then this is also another possible history of a process." But the declarative reasoning in concurrent logic programs are about histories of processes rather than about logical relations between values like in a simple logic program. So, even though the classification is useful, I think GHC should be put in the concurrence side rather than in the declarative side, because you must reason about the control aspects to understand it. And I think this also answers Carl Hewitt's point that if you do not want to think about these languages declaratively, no one forces you. There is a precise operational semantics of these languages. You can just think about it and ignore the logical programming aspect.

The point is that many of us believe it is useful to think about the logic programming aspect in addition to the operational aspect. If you want no one to force you. But you have the option to do that.

There was a comment from the audience.

PER BRAND (SICS, Sweden): I would like to make a comment on Andorra Prolog. Just now we have quite a precise operational model for a fragment of Andorra Prolog. And this fragement is defined in the paper for the conference and as David said our system is still in the situation where we might change the semantics. So, today we have an exact operational semantics, we don't have a denotational semantics I would say.

But in this operational semantics you can really write anything you would like to write in GHC and you can also write things which combine Prolog and GHC very easily. There is a very smooth combination between things you would write in Prolog and write in GHC.

CHAIRMAN: Any other comments to David's presentation?

HENRY LIBERMAN (MIT, U.S.A.): Yes, I would like to talk about the goal of having parallelism being invisible. I would like to say that I find that totally incredible as a goal. And the reason is that in many applications the whole goal of the application is to make the concurrency visible. A simple example of that would be something like a flight simulator, where you have on screen a display of some train, and a bunch of gauges that are measuring fuel and altitude. You definitely like all these things to operate in parallel in the natural way of writing the program for gauges, measure the current amount of fuel and display it on the screen, measure the current amount of fuel some X time period and display it on the screen and so forth. If you cannot say in your languages that these gauges are all running in parallel, then I don't see how you can possibly write a program like that. So, I just want to say that I would like to challenge the notion that parallelism is invisible.

WARREN: Okay, obviously my first slide wasn't very clear. What you are describing in my terminology is "concurrency" not "parallelism." If your application could run on a sequential machine, I think you would be quite happy to do so. So, what I am saying is not "concurrency" should be invisible but "parallelism" of the underlying hardware should be invisible. Do you see what I mean?—No. For example, as I have said in a pipeline machine, the parallelism is indeed invisible to an ordinary programmer.

MILNER: Could I just add something to this? I think it is possible to have both of

these things. I certainly believe when a person asks a question one must be able to express concurrency if that is part of the structure of the problem. I equally agree with a lot of other people that one would like to decouple that from the parallelism that is in the machine and I don't in the least see why either those things mean that you shouldn't also be able in a language to express, if you wish to, that a certain process that you have in your program should be run on a certain processor. I don't see why programmers should be denied that. I think we should make sure that the programmer who doesn't want to do that still has the expressive power of all kinds including concurrency, but I don't see why one shouldn't added to that the ability to express what is going on on a particular machine, if that is very much his concern, what he actually wants to get the job run and run fast and that may be the way to do it sometimes.

WARREN: Just to sum up then, I think "concurrency" should be visible, "parallelism" should be invisible. Perhaps that makes it a bit clearer?

HEWITT: I would like to address this issue about control, where basically logic wasn't able to do it by itself, but maybe control would be the knight in shiny armor that would come and rescue us from the situation with respect to the Guarded Horn Clauses.

It seems to me that when Kowalsky introduced this notion of control, the idea was to be able to prune the branch of the tree. You had all these possible deduction; if one prone that branch of the tree, that will be okay. But it seems to me that is not the situation with respect to Guarded Horn Clauses. In the example of the shared account, because I have my stream and I assert the first element that is withdrawn and send it down. I get the result back. It is not like I has too many deductions that need to be proved okay. Instead I got this conclusion that didn't follow deductively at all. So there may be some problem here.

CHAIRMAN (SHAPIRO): I think Carl was referring to the role of logic programming and its relations to concurrent logic programming. With the permission of the other panelists I have also prepared a presentation, and I think I will answer the question.

Even though I was all ready to be the elderly statesman who just sits there as the manger of the discussion, I was told by other panelists that I am not yet old enough for that and I should also say what I am thinking, so that is what I am doing.

So, I would like to propose an image, a possible hypothetical image of future computer systems. As you will see I will also have a certain idea who should be on the bottom. This image consists of two concepts: one is logic programming and the other is partial evaluation. And I will explain both of them, how they relate to what was mentioned.

Logic programming started with its practical realization in Prolog which is still the main workhorse of logic programming, but in the last few years we see major and very improtant developments in logic programming, which cause it to diversify and be applied to many different areas. One of them is application to logic databases and there is the LDL system from MCC, the ECRC database system, etc. There are the very exciting and very promising developments in the area of constraint logic programming, Prolog 3, CLP (R), CHIP, etc, which seem to be

extremely powerful application programming languages. There are also perhaps less well known but I believe very important and significant developments in the direction of higher order logic programming and I refer here Lambda Prolog, the work of Dale Miller and his colleagues. Also developments which we heard a lot today during the conference in the area of concurrent logic languages, GHC, Parlog, concurrent Prolog, etc. I believe that each of these directions which logic programming is developing is very important and has its own stance which is independent of the other developments. So, it is not that it is threatening other developments or depends on successes of other developments but it is a well defined and quite successful avenue of research which seems to be leading to very fruitful results.

The question is that we all started asking several years ago and we are still asking, is how can we use logic programming as foundation of our future computer systems. And how do we integrate or apply all these areas? And how will we apply them in the future? Perhaps the most conventional answer is that we will have a network of workstations running UNIX and all these will be applications on this network.

But I believe this direction will not be very successful, because for every level of abstraction that you start there is a certain degree of complexity of a system you can build on top and after that the system collapses under its own weight. So, without building layers and layers of abstraction, you cannot build the highly complex systems that we need to build in the future. And I think every panelist here was (or most of them as least were) suggesting different layers of abstraction to build their systems on, and I am going to discuss one of them now.

Closely related to the concept of layers of abstraction is the idea of partial evaluation. What are implications of the idea of partial evaluation? We heard an excellent talk by Dr. Futamura—I will not repeat the ideas, just the implications. Perhaps the most important implication is that there is no intrinsic overhead associated with high level formalisms. So, it is not the case that if you use a high level formalism to specify a solution your specification must run slowly, and if you use a low level formalism it will run faster. If you use a high level formalism just to specify a solution, then you can partially evaluate the formalism plus the solution on any particular strata, and in principle there should not be any additional overhead besides what you specified in the solution. So, therefore, the expressive power of formalisms that we use is of fundamental importance, perhaps more than the "overhead" associated with it whether it is too high level or not.

Another implication is that overhead of layers of abstraction can be eliminated. So we should not be too scared of building layers and layers of abstraction, because in principle we have the concept of how to eliminate them—this is partial evaluation. So, jsut as an illustration, in principle we can take the high level language application program which specifies some solution and a compiler for this high level language, which compiles some to architecture, and the hardware description of the architecture in some hardware description language, then partially evaluate the whole thing into a specialized hardware description, which can then, using VLSI compilers, generate hardware. So partial evaluation knows no limits in some sense. You can partially evaluate through arbitrary layers of

abstraction with enough ingenuity.

So, to conclude these implications, I think that an abstract formalism which embeds naturally and efficiently all other useful formalisms can be employed as a foundation of a general purpose computer system.

And my personal conclusion is that concurrent logic languages offer such a formalism. So, this may explain a little bit why my research is so obsessed with embedding every formalism that someone invents in concurrent logic languages. Because I want to be sure that the concurrent logic languages are expressive enough to support natural and efficient embedding of this new formalism or some old formalism.

And if this is possible, then we can have this possible view of the future computer system. At the bottom we have the hardware, which has work stations, parallel computers probably (in some foreseeable future) three dimensional meshes, data base machines, and on top of this layer of abstraction that I described based on the concurrent logic languages, the operating system, and on top of these all other languages. I think are applications in which you want to progrm in a concurrent logic language, but also there are applications that you want to program in a constraint language which does not show parallelism, and you want applications to program in a function language. So, you would like to support all formalisms or useful formalisms in this computer system.

What do you do with the overhead? The answer is to partially evaluate it. So, with appropriate partial evaluation techniques, the applications can be mapped directly to the hardware and eliminate all these layers and layers of abstraction which we build in software.

So, if there are any comments from the panelists I will take them.

WARREN: What do you do to have a precise notion of how efficient an embedding is? How efficient is it and how do you characterize it?

Because my feeling is that embeddings we have seen from high level languages into concurrent language aren't efficient enough. They are not constant time embeddings, whereby one operation in a high level language maps down into a single constant time operation in the concurrent logic language, and therefore they are not adequate for the task.

CHAIRMAN: The answer is: Yes, there are constant time embeddings. Actually there is a paper in the concurrent prolog book which talks about a test for the adequacy of a language for an architecture, which proposes a criterion of having constant time and constant space overhead embeddings as a criterion for suggesting such a language. And I showed it for the particular model of FCP, or FGHC is the same in this case. So, the answer is yes, you can have constant time embeddings. And it can be proved formally. Whether you can find for a particular high level language an efficient embedding, that is another matter. Because the former proof goes by simulation of a von Neumann machine. So you can take your Prolog compiler which compiles into a von Neumann instruction set, and then simulate the von Neumann machine in FGHC. This will give you a constant time overhead in the proof. However, to get practical embeddings, you need more ingenious schemes. For example, there is another paper there on OR-parallel Prolog compilation into flat concurrent Prolog, which shows a more

practical embedding of OR-parallel Prolog into a concurrent logic language. This is arguable but it seems to give practical results.

So my answer is that using today's implementation techniques, maybe we can get reasonable but not competitive results, but certainly we have a proof of principle. And the concept of partial evaluation has no limitations. So whether we can do it today or not, that maybe depends on our understanding of how to apply partial evaluation techniques. But in principle there is no intrinsic overhead associated with high level formalisms, provided that can prove this constant time overhead theory.

WARREN: Are you saying you think you can do that for OR-parallel Prolog down into committed choice language?

CHAIRMAN: Yes, concurrent logic languages, yes. I believe this can be done, and I have given some evidence to this.

WARREN: It can be done or it has been done? I haven't seen any where it appears to me there is evidence that it has been done.

CHAIRMAN: This has been done for Prolog and the theoretical analysis shows that for the balanced trees you have a logarithmic overhead compared to sequential implementation, for the particular implementation techniques are used.

WARREN: I mean, that's showing that for a subset of programs there is a logarithmic, not constant-time, overhead. That isn't quite what we are looking for.

CHAIRMAN: Well, we all are doing research, and we are at different stages of having half baked ideas and half baked results. But I think this result is at least for me convincing enough to try and apply it to Andorra. And actually that is one of the computational models I am going to rush to embed in FCP after this conference, and we can discuss the results maybe in some other conference. Then we can say if it is realistic or not. We can also compare it to, for example, how much effort it would be for me to write an Andorra interpreter in FCP, versus how much slower it will take to run, versus how long it will take for you to write an Andorra interpreter and see how long it will run.

DALLY: From a system's point of view, you did not talk so much whether you have constant time embedding or not, but what in fact the constant is. And perhaps I am a minority here and I am not having a religious fervor about logic programming. But I question whether a language that is as powerful as providing unification as the primitive construct is primitive enough to be able to model languages that use communication in a much more basic way. When you say partial evaluation, are you thinking of compiling things down well below the level of FCP?

CHAIRMAN: Well, in general, yes. But the answer holds even for languages like FCP, FGHC or Prolog, I think the success of the Prolog precisely depended on the fact that we compile special cases of unification to constant time von Neumann operations. The practice of Prolog shows that in 90% of the time or 95% of the time programs do not use general unification but use unification to simulate the simple operation of load, store, cons, car, cdr, etc. And they are compiled efficiently enough

so that the overhead of using them for these purposes are small. So the advantage of having such (some people say) wide spectrum formalism is that if you stick with the subset which does not exploit the full power of it, it can be compiled without the overhead associated with the full power. But in the X percent case where you use the full power, then you pay for it, but you know what you are getting.

So my answer is yes on the fact that even without further compiling, just the languages as they are, are already implemented, the operations which simulte simple operations take constant time, and a small constant time. I also believe that techniques of partial evaluation can be used to cross this layer of abstraction if needed. But that is a cojecture.

Let's go round robin again.

MILNER: Yesterday we have heard in Dr. Futamura's talk that partial evaluation is of course perhaps a restricted kind of program transformation, and so is compilation. I believe that perhaps it is a little bit restricted to think of just the method of partially evaluating programs down through a logic programming language, in order to get efficient implementation. I think we should think much more broadly about the business of transforming programs from a very pleasant and natural way for its particular application into another languages which may be one close to a particular machine. And I think that there is a lot of unanswered question here, a sort of presumption that somewhere along this line there has to be a logic programming language. And I don't think that has been justified. I think in this community, it may be presumed that that is necessary. But I believe strongly that the important concepts of programming have very often come from some kind of restricted science. For example, matrices are an important concept which are very useful in programming. Logic is a very important concept, very useful in programming. What we are looking for (and it happens not to have been discovered as early as logic) is the mathematics of concurrency which when found and properly analyzed can then be imported into programming. I think we are in danger of following a path where we get stuck on one particular mathematically elegant form of analysis, namely logic.

CHAIRMAN: Ueda-san.

UEDA: I am afraid that the partial evaluation technique is not almighty. For some higher level languages it may be easy to write meta-interpreters and partially evaluate them but for other higher level languages it may be more straightforward to write compilers rather than interpreters. Three years, I tried to implement pure Prolog on top of GHC and found that it is much more straightforward to write a compiler directly. The reason is that the compilation involves data flow analysis. Maybe the meta-interpreter approach can be used only when such data flow analysis is not necessary.

SHAPIRO: I would like to respond to these two points. First, when I said partial evaluation, I did not mean strictly in the narrow sense. I meant that there was a general conceptual framework which shows a high level description, you do not need to pay for high level description more tha you need for a low level description, because they are equivalent, and there are conceptual methods of mapping one to the other. How this mapping is done that depends on

the technology.

For example in the Logix system we do a lot of the mappings manually because we do not have a good partial evaluator.

Also I do not suggest that someone is going to go and rewrite LDL in FGHC, and partially evaluate it back to its original form. However, a concurrent logic language is expressive enough to specify the functionality of LDL, and therefore, in a retrospect construction of such a homogeneous system, you can reconstruct LDL, so that its interface looks as if it is a result of partially evaluating FGHC prgram and then it talks to the rest to the FGHC system.

So, I suggest partial evaluation is a unifying concept and the technology in which we have partial evaluators today or tomorrow should not stop us. The same way it did not stop us in Logix to use the concept for implementing the meta level functions manually, since we did not have partial evaluators. But the unifying framework is that the functionality of everything should be as if it were specified by the formalism and then partially evaluated, whether it was done so in practice or not is another matter.

To the question of the logic layer, this is a working hypothesis, and I certainly agree that we have not proved it. However there are two ways of doing science. And I believe in the Popperian way of doing science which is making bold conjectures and then either they are refuted or not. But the bolder the conjecture is, the more you learn, when it is refuted. Also the more you win if it is successful. So, I certainly agree that at this stage, it is a bold conjecture and other people are welcome to try other bold conjectures. But I think it is fair to say that the amount of intellectual effort invested in exploring this bold conjecture to its full spectrum is much larger compared to other models of concurrency from the application side. I am not talking about the theory side. I am looking at how to compile other languages, how to write applications, etc.

I think we can be proud in the concurrent logic progrmming community that we have done a fairly thorough job until now and we have at least a good confidence that if we have got this layer of abstraction implemented efficiently, then it will be a very useful one.

Let's continue with the answer.

DALLY: There is no question that it is useful. I think in looking at what everybody's slides had here in pictrues of things that were skinny in the middle and fatter on the top and fatter on the bottom, there were a lots of models or programming that we wanted to encompass, either various forms of logic programming or perhaps extending out to people who want to write data parallel, data flow or actor programs as well, and down in the middle was whatever our favorite kernel language was. Toward the bottom were perhaps many implementation vehicles. The real question becomes what are the proper criteria for picking the small kernel languages in the middle. I would argue that there are two criteria. One is looking up from the bottom. It is in fact something that you can implement effectively. The other is looking down from the top and from as broad a view from the top as possible. Is it one where you couldn't do much better, if you were to specialize in some way. In many ways I find logic programming lacking as the bottom-most abstraction level for this. And the reason is to implement object-oriented or actor language or to implement a more synchronous parallel language like CSP or

OCCAM. In both cases the process of sending the message to an actor of falling off computation or having a rendezvous can be simply implemented in hardware by a primitive communication operation, the synchronization operation in response to that. But if I have to go through logic programming, I wind up having to cons up the stream. I am allocating storage in some sense, and have to resolve names of the stream variables. And so I am putting much more machinery underneath what should be a very primitive operation. Because it can map directly into the hardware. It may be that the logic programming or kernel logic programming language forms a nice layer, a sort of at one branch up the tree, to say here is the whole bunch of logic programming paradigms that map into kernel logic programming language. But it seems there is a more fundamental set of primitives that lies somewhere below that, that needs to be a compilation target for not just the logic programming languages but a broader set of abstractions.

CHAIRMAN: So let me ask you. Let us consider two implementation techniques of an actual language, one which goes directly to hardware and one which goes via concurrent logic programming language. Below what overhead will you say it is practical to use the programming enviornment operating system and etc. of logic programming language, to use an actual language rather than to compile a directory? What is you threshold?

DALLY: I am not sure I understand the threshold. But I am not suggesting to compile directly in the hardware. I think the notion of an abstract machine is quite important to save investment, if nothing else, so that as machines will come and go, as people discover more about the nature of parallel architecture, we are just beginning to do that. What you want to do is to save the investment in programming both applications and systems programs as you move different things around and underneath.

CHAIRMAN: Okay, that is the point of one possible area of abstraction such as the concurrent logic language. So, I would like to repeat my question to you.

DALLY: Right, the threshold in terms of overhead?

CHAIRMAN: Yes, if let's say 5, 10, 20, 100 times lower, what is the threshold below which you will say, "Yes, I am willing to use a concurrent logic language and intermediate language to get programming environment, operating system, portability, etc. and lose X in performance." What is your....

DALLY: Well, the real question is what I get in return for acts, if I get long return....

CHAIRMAN: All the software that could be writtern in this language.

HEWITT: I would just like to agree with Robin, and see if i can't sharpen this up a little bit. It seems to me what we need is a foundation and a model of these concurrent object systems on which to build, and not any given particular programming language, because you like the support of a whole variety different kinds of programming languages of this kind and that kind and functional or what ever you have got. So, what can we say about layer? I think for sure, as Robin pointed out, we can say

it is going to be fundamentally based on communication. So that is fundamentally the notion of communication and the notion of some kind individual, either concurrent object or an actor, that is going to define the layer between hardware and software.

And it seems to me that many of these languages that don't use the actor approach arrive with excess baggage that ought not to be part of our foundation. And in terms of the excess baggage for these Guarded Horn Clauses Languages, I would propose two kinds of excess baggage. I think that unification is excess baggage and I believe that guards are excess baggage that they sort of got in there because of historical accidents etc., and they are not really fundamental to our notion of communication and actors in this kind of layer that we are trying to create.

CHAIRMAN: Would you like to respond to this, Ueda-san? Okay, so, maybe I will respond.

Unification, I agree, is a very powerful operation. And the point I tried to make before is that if you do not use it in its full generality it costs no more than car, cdr or cons costs. The point of guards, the guards are also in CCS, CSP, OCCAM, all concurrent logic languages have guards in one form or another.

HEWITT: The Guarded Horn Clauses Languages have guards. Not all concurrent programming languages have them.

CHAIRMAN: In CCS, CSP-guards are hidden in one way or another before non-deterministic choices, maybe except Actors.

HEWITT: Bill Dally's language and a lots of others—there are many, many object-oriented concurrent languages that do not have guards.

CHAIRMAN: Chikayama-san, please.

T. CHIKAYAMA (ICOT, Japan): Just a small comment about the communication by using less structure. I have in my mind now just a few minutes ago a sort of optimization that you can communicate between two processors without really allocating a cons cell. But still in the language level whether using cons cell there. This kind of optimization is already there. Now, you did say most of the part I wanted to say. So just for a comment—that a sort of optimization is also possible. Even we can eliminate the overhead by allocating a cons cell well.

HEWITT: It seems to me when we are talking excess baggage, there are two parts of it. One is what kind of optimizations you can do on the machine in trying to make it run as fast as if you didn't have it. That is part of the overhead excess baggage which we can do something about the kind of technologies mentioned earlier for optimizing special cases. The other part of the excess baggage is the mental excess baggage of your having to think about all this stuff all the time. Even if in most cases only special cases are used, we still have to have in reserve explanations for all these guards and for this complicated unification.

CHAIRMAN: Leon:

L. STERLING (Case Western Reserve Univ., U.S.A.): I would like to change the topic a little bit and move it back to the central theme of the Fifth Generation project. Could the panel comment on the implications of their views on concurrency

to the prospects of machine intelligence?

CHAIRMAN: I think you may have hit an area where the panel is not an expert. But maybe I will give the panelists the try.

HEWITT: I will give it a try.

It seems to me that there is a challenge here in terms of going beyond the previous conceptions. Because with these massively concurrent machines, you are unable to deal with them or progrm the in the ways that are traditional in artificial intelligence. I think that somehow we have to bring in some of the technology dealing with the large scale organizations, and human teams, and that kind of thing as the only way we can keep track of and keep going on this particular route. Because our old notion of artificial intelligence which we originally got from Turing was very much a psychological notion, where you wanted to make somebody who was intelligent. With all these actors and all these billions of messages travelling back and forth, we can support new kinds of organizations that operate in open systems.

CHAIRMAN: David?

WARREN: Yes, I think building parallel inference machines doesn't directly solve the problems of machine intelligence, not even at all. But it does provide a more powerful tool with which we might be able to tackle those problems.

CHAIRMAN: Robin?

MILNER; I think what I will say is pretty much the same sort of thing as David. It is just that it is more of the nature that artificial intelligence is almost by definition the hardest problem that there is, and therefore we should try to understand our tools. And the tools will actually be largely concurrent in the future.

QUESTION (Floor): Let me perhaps be a bit more focused and push a point a little bit more.

Something which I perceived as common among all of the presentations was the breakdown of the world into individual objects on quite discrete events. There are commitments as well in terms of linguistic abstractions. Can I push that point a little bit and have people make a stand? Do you think that is a fundamental correct way of viewing the world? Certainly Carl Hewitt's view could be pushed saying there were something potentially problematic with those views, when you didn't know what the result with the individual transactions were. Somehow we are making commitment with the design of these machines and building them into indivual object in saying something about the ultimate prospects or intelligence or not of these machines.

DALLY: Well, the world at some level is inherently discrete and the computer world is even more discrete than that. Because if we do limit ourselves to building digital computers, we fundamentally have storage cells that remember bits, and all we can do with the computer system is to decide how we are going to connect these storage cells together and what stage transformational function we are going to decide, define on that. So, it's a very fundamental level of computer design, where you have a finite state alternative. They are communicating in some sense as all of the models you have seen need to build up from that level. And the question of linguistic abstraction is that people don't want to think about every bit

in machine at once, what are the right models, what are the right abstractions that we can give them, and they can harness all of these bits and make them do something useful.

MILNER: I would like to add that. It does seem possible to get breakdowns of the world which could be expressed as concurrent programs, but in which separate parts are not necessarily corresponding to the physical parts of the system. I am thinking of some of the very interesting work by Gerard Berry and his team at Sophia Antipolis in France, where they model real time concurrent systems in a kind of process algebra, but there it turns out that the synchronization between separate events which we perhaps normally think of as a comunication, turns out to be virtual parts or different views of the same system. So that it is rather like finding decompositions of, say, finite state automata in which each component is not a physical part of the real machine but a way of viewing parts of the state structure of a machine. He even gave a model, for example the wrist watch or something of that kind as a composition of such things. And I think the state charts work of Harel was also like that. So, in this world of process algebras or event based models, you don't have to think of the parts as physical parts, although it is open to you to do that.

CHAIRMAN: Ueda-san.

UEDA: I think this project has a very far reaching goal as regards applications such as machine intelligence. Ten years ago we hoped that the semantic gap between applications and hardware would be smaller and smaller. But now it seems to me that the semantic gap between hardware and applications is getting bigger and bigger. Computer architecture is now becoming simpler and simpler, while applications are getting more and more complex. At the hardware level we now have to consider the physical distribution of objects. What we should do in the face of this fact is to find many good concepts to connect these two ends, but that is a very difficult thing. What we are trying to establish by working on our kernel language is one of these concepts which can help connecting the two ends of our project.

CHIRMAN: Furukawa-san.

K. FURUKAWA (ICOT, Japan): First of all, our initial goal was not to create a very artificially intelligent brain but to achieve some foundation for the research of knowledge information processing system. I think the concurrency and parallelism is very much related in several aspects. One is of course to increase the processing power and the other is to enhance what we deal in the sense of intelligence. We have to deal with concurrency objects in that scope also. One other approach is as we have said that we are preparing a lot of layers bridging from hardware and application. Since I agree with these partial evaluation approach very much, though I don't think it is very easy, what I think we need is to develop various kinds of tricks to enable such transformation from a higher level description to a lower level.

DALLY: I would like to follow up on Ueda-san's comment by saying that if the semantic gap is getting larger between hardware and application, we should look upon that as a good thing rather than a bad

thing, because it reflects a greater understanding in two respects.

The first is by moving hardware down to the lower level, means that we have increased our understanding of what are the right things to build into the hardware. There was a point in time when hardware was burdened down with many complexed features that it didn't need. By stripping these off, we built much more efficient hardware. At the same time we can move the level of porgramming up which enables people to write much more powerful applications than they could if programming was at the level very close to the hardware. So, the large semantic gap is something to strive for, not something try to close.

CHAIRMAN: Please state your name.

J. WOOD (Australia): James Wood from Australia. When I return home I will be asked by people who probably heard of Prolog I just read in the paper about the connection machine with 64,000 processors. So, what is it about the PIM or the Multi-PSI with 64 processor, that is getting such enormous attention? Could the panel give me a one sentence or one paragraph answer that I could take home with me, so that I can travel overseas again?

DALLY: A processor is a very poor measure of anything. Well, that's one sentence. I mean people very often pull one number out of something and then start comparing numbers that look like are similar. And it is not the case. The connection machine, I am sure Dr. Waltz can say more about it than I can, has a lot of one bit processors. You can think of them as two 8 to 1 multiplexers. Because that is in fact exactly what they are. They take 3 bits in and you put in an arbitrary 2 table in and they generate 2 bits out. And that is a very small primitive to start building from. So, it is hard to compare that with the processor in a PSI, which is a powerful 36 bit machine, has a lot of mechanism built in for handling the logic and list manipulation languages.

CHAIRMAN: Anyone from the panel who wants to respond? Ueda-san? Okay, please.

D. WALTZ (Thinking Machine Corp., U.S.A.): I have a question about general purposeness and also about concurrency purposes of view.

It seems to me the most interesting source of concurrency is really the real world, the real visual world as a good example. A lot of what humans do I think is to understand concurrency of various events. It is how we understand the causality and that various regions are actually joined to each other. If that is true, then it ought to be the case if you have a good concurrent language that you could deal with things like vision. Now, it seems to me unless I missed the point completely that no one has ever proposed a logic programming or any of the descendants of it that have any relevance to vision at all. It seems to me that for the most part logic programming languages and their descendants and relatives have very little to say about things like visual processing. On the other hand for other kinds of machines such as connection machine, those are natural problems and they have relatively less to say about logic abstraction and what I may call the ragged reasoning. In some sense, what you really might want is the general theory, which seems to me in some sense not really being looked up by either of those kinds of general views. And I

wonder if you could pick my comment on what would really be a good general concurrent theory that would be satisfactory for tasks for vision as well as reasoning.

CHAIRMAN: I would like to respond and maybe others as well.

One way of viewing the relationship between the simple and highly regular processing in vision and logic programming is that it is fairly easy to write systolic like programs in concurrent logic programs which specify algorithms that you are implementing in the connection machine. So, for formalizing and specifying these algorithms concurrent logic languages are very good tools and there are good examples. It is true that given today's technology of implementing these languages even on the connection machine probably this is a long way to go. So, the answer is maybe today we partially evaluate a mental or a formal representation and write down manually connection machine programs in C. But maybe in some day in the future there will be a compiler or partial evaluator that maps these high level descriptions into highly specialized machines like the connection machine. So, the view that I proposed does not exclude the high concurrent algorithm on specialized machines like the connection machine, although I agree that the software technology and an understanding of how to do it automatically is still far from being realized today.

WARREN: Yes, I think vision is an interesting problem to look at with logic programming. Particularly when we do have parallel systems which will actually be able to deliver big performance. Vision, of course, is a great spectrum from the high level aspects of vision down to the low level aspects. But certainly, consider a system with and-parallelism and or-parallelism for example looking for a face in a crowd. The or-parallelism is searching in the whole scene to find a possible face and the and-parallelism consists of analyzing the particular face at that position to see whether it corresponds to the particular face you are looking for. So, it does seem to me that it is an application which might well be of value look at.

CHAIRMAN: Robin?

MILNER: Could I say briefly something about mathematical models? I am very impressed with the fact that some of the fundamental models in computing don't really help you at all when you come to something as sophisticated and specialized as vision—so we are really struck by the enormous contrast between a general model of computation, which may say something about basic concepts of concurrency but certainly doesn't have any ability in itself to take advantage of regularity of structure, and the kind of machine structure which is well adapted to a particular problem. There is an enormous gap between those models and specialized models. I think that as model builders we should feel pretty humiliated by problems like vision. You can presumably represent them in a general purpose programming language. You might find it much easier to represent them in a specialized programming language. So, I believe the vision is one of those things which might well benefit from special programming methods or special programming language even.

DALLY: As a machine builder rather than a model building, I don't feel too humiliated by vision. If you look at what the connection machine for instance

provides to handle vision, it is a certain amount of data type specialization in being able to handle one bit values easily. But in fact even for early vision people are more interested in handling things in the range of 6-12 bit depending on the resolution of the camera systems. And so there is a certain amount of building up from there.

And the other thing is a certain amount of communication specialization being able to take an advantage of the locality of the machine. If you build the appropriate priitive mechanisms you can get both of those out of the more general purpose machine. You can implement data type operations on small integers quite efficiently and you can communicate with nearest neighbors making very good use of the available wiring bandwidth, and you can do all of this without sacrificing the generality of being able to support asynchronous models of computation, and those that require global shared memory.

I think that it is perhaps difficult to do it in logic programming. But I think there exists a machine that will provide very good support for vision. Perhaps you can't compete with a specialized vision machine that has been hard wired for few early vision functions. But I think it can offer a good alternative for prototyping.

CHAIRMAN: Would you like to continue?

WALTZ: Yes, I think I hear the answers and I think there is some truth in them all. It is a little hard to express, but it seems to me that the fundamental thrust of a lot of the work in the family of languages that you are most interested in is toward the selection of just those things that you should look at and the things that you should specifically combine. Efficient use of resouces apply to just the places where they ought to be applied.

Leaving aside the current computers entirely, it seems to be the flavor of real intelligence may well have a quite different character which is a profligate is use of many kind of processors most of which have no relevance most of the time in the interest of getting extremely fast latency when it is needed. Or there are a lot of processors not attempted for real use. A small number of processors constructs very efficiently but we are interested in the large number of processors in order to get a quick turn around?

CHAIRMAN: Ueda-san, do you like to respond?

UEDA: I have one comment. Until now we have been using concurrent logic languages for programming parallel computaion, but now we have a plan to use concurrent logic languages for programming storage, for database and other applications. Thus, a primitive language can cover a wide spectrum of applications.

CHAIRMAN: I would like to comment as well that concurrent logic languages are in some sense a low level algorithmic languages and they take no stand as to what style of algorithm you wish to implement. At least that is the point that we are trying to investigate.

Any further comments?

T. ITOH (Tohoku Univ., Japan): I think one of the most important and difficult concepts in the phenomenal concurrency is existence of deadlock. Proving the existence of deadlock and recovery from deadlock will be very important in theory and practice of concurrent systems. But in the discussion I have not heard any mention of

deadlock. Are you all free from deadlock?

CHAIRMAN: Anyone?

MILNER: We have a piece of software built around CCS which recently discovered a deadlock in the VAX BMS mailing system. It was a very short activity and it is being mended. So, we can build software which helps, based on particular theories, which does help in discovery and mending of deadlocks. So at least I can say that that is one of the easiest things. It is quite hard to prove equivalence, or particular properties, but it is very easy in the calculus of communicating systems to discover deadlock.

CHAIRMAN: Any comments on deadlock?

HEWITT: With respect to deadlocks I think they can basically make the cost go down to the point where there are basically transactions sitting around among the actors that aren't going anywhere. They eventually can clean up via time out. But I think if you get organizational structure down, the way you paste your concurrent systems together, it would never have any deadlocks at all. You have those transactions that are never going to be completed, but they can eventually be swept out of the system.

ITOH: I have one more question about deadlock. Some people stress the importance of partial evaluation. Partial evaluation brought new difficult problems when we discussed recovery from deadlock or the provable existence of deadlock. Are there any comments or any idea on that aspect?

CHAIRMAN: I would like to respond to this. First of all the possibility of deadlock is inherent in any concurrent formalism or programming language. In some sense the language that cannot specify a deadlock is not a concurrent programming language.

With respect to detecting or proving the absence of deadlock, there is a paper by John Gallagher, Codish and myself in the Meta 88 Conference, which shows how using techniques of abstract interpretation you can analyze the deadlock behavior of concurrent logic programs. So, it is not partial evaluation but abstract interpretation, although they are closely related techniques. And I am sure that there are some relationships as well to the CCS techniques that have been mentioned.

Well, we are nearing the end of this panel. So, we will take one final question, and then we will have the final words from the panelists.

QUESTION (Floor): May I speak from the AI and knowledge representation point of view? I address this question to Prof. Milner.

Historically speaking the present logic programming effort has evolved from the assumption that the first chart of logic is a very good knowledge representation scheme and can represent most of the knowedge that machine intelligence requires. Therefore, Prolog and logic programming evolved. Prof. Milner was saying that if we could attack concurrency directly with a new kind of mathematics. Do you think this new kind of mathematics could provide a different knowledge representation scheme which will diverge from this network and logic and various kinds of schemes?

MILNER: I would hope so. I just would like to keep the door open for it, and I

would also like to mention that modal logics are very good for even systems which concurrency tends to be concerned with. So, one might think of perhaps modal logic programming as a refreshing means for bringing together two apparently warring factions here. I mean, modal logics perhaps have some of the assertive advantages of logic programming. They are also very close to process algebras.

So, I wouldn't give a complete answer to what you asked, but I do believe that there are possibilities, there are sort of increments towards a solution.

CHAIRMAN: I would like the panelists to think about one or two sentences which will symbolize the end of the panel and summarizes the best they can their impressions or thoughts, and we will close with this.

DALLY: I would like to close this with some comments on architecture in that from what everybody was saying they agree that there are a sort of abstract machine at some level. And I would like to suggest the challenge for architects is not either to try to wire together what is available today in the best way or to hard wire their favorite linguistics model, but to identify some general efficient mechanisms. And just as the processor in a PIM or Multi-PSI not equivalent to a processor in connection machine, they need to be quite careful about what the costs are in the machine, and then it is not how many processors are being used at a particular point in time or the fraction of processors that are used, but the cost of building the machine versus how fast the problem is solved.

Another comment on mechanisms is that the right mechanisms to build as an abstract machine model from an architectural point of view, are unlikely to be the same ones that you would choose a linguistic point of view or from the descriptive point of view.

CHAIRMAN: Carl.

HEWITT: I actually think that these are very exciting times particularly here with new generation concurrent systems. From the beginning, ICOT correctly concentrated on the concurrency as the absolutely fundamental issue and at the same time coupled it with advanced information systems. I think that really is right at the heart of the matter. In fact, this marks the transition from artificial intelligence based on psychology with sequential streams of consciousness thoughts and proof ways of doing business to a more sociological approach on commitments, negotiations, responsibilities, etc. It is fundamentally concurrent instead of being sequential and it is fundamentally based on commitments and negotiations instead of being based on arguments and proof.

MILNER: I would just like to say that I do believe in bold conjectures and the more well delineated the conjectures, the better. And let's have many contrasting conjectures. I would also like to say that nobody has persuaded me that inference is more fundamental than action. I still think possibly that action is more fundamental than inference.

That is all I want to say.

UEDA: Let me suggest two directions for our kernel language research. There are two problems with the current kernel language. One is that it is too weak because it does not yet provide an elgant framework for systems programming. So, we have to

consider reflection capabilities of parallel computation.

The second problem is that the current kernel language is too strong in the sense that the basic mechanism, namely unification, is hard to implement efficiently on a parallel machine. So now we plan to design a very efficient subset of our kernel language to encourage parallel programming.

WARREN: I would like to go back to my original point distinguishing between "concurrency" and "parallelism". If we are looking at applications, in many applications, concurrency is a relevant issue but I think in the majority it is not really a central issue to the problem we are trying to solve. So from that point of view, what we have more been discussing today is a rather small part of overall problem to be solved.

If we are looking at parallelism, trying to make computations run faster, then I think that is a small but important area. I think we should try and keep the parallelism as low down and as invisible to the average user as possible.

CHAIRMAN: I think this panel as well as the conference symbolizes the maturity of our field of research as well as our ability at least to interact intelligently people from other approaches, if not to agree with them. I think it is a very good sign that we come to terms and understand each other's annotations and concepts to a degree that we can argue in an intelligent way, and perhaps even exchange fruitful ideas with each other.

I would like to thank the panelists for their participation and the audience for questions and participation.

Thank you very much.