Panel Discussion

A Springboard for Information Processing in the 21st Century

Chairperson:

Robert A. Kowalski
Professor, Imperial College, U.K.

Panelists:

Hervé Gallaire
G. S. I., France

Ross Overbeek
Argonne National Laboratory, U. S. A.

Peter Wegner
Professor, Brown University, U. S. A.

Koichi Furukawa
ICOT, Japan

Shunichi Uchida
ICOT, Japan

Kowalski: I would like to begin this panel by presenting a brief summary of the Fifth Generation Computer Systems Project, both in terms of its objectives and in terms of the technologies it identified in order to achieve those objectives. In broad terms we can identify three such areas of technology.

The first and, to the world outside the FGCS community, most prominent area is the one concerned with applications, sometimes referred to as "artificial intelligence", but more modestly in this case better described as "knowledge information processing".

The second area concerns the development of parallel computer architectures. This is one of the main areas which the FGCS project has explored.

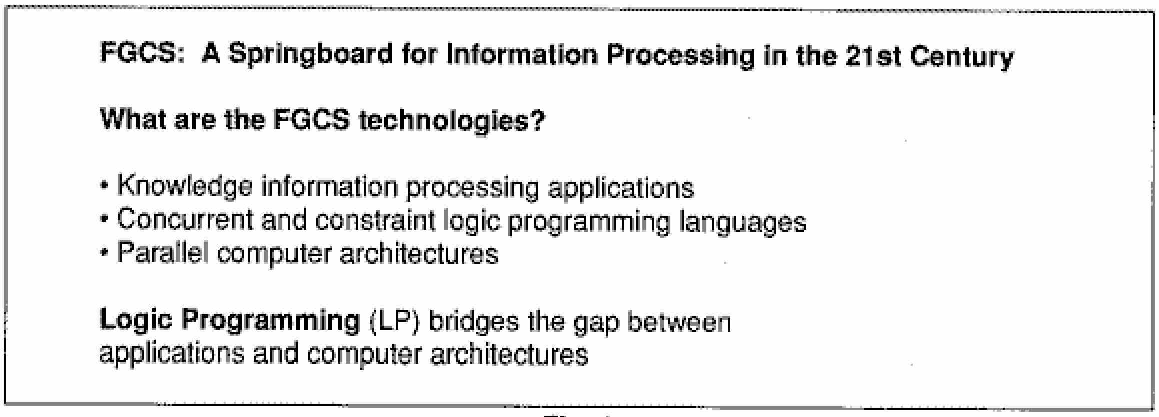
The third area, bridging the gap between applications and computer architectures, is concerned with software; and in the FGCS project this has concentrated on the development of concurrent and constraint logic programming languages. It is perhaps the identification of the importance of this third area of work which is the most original aspect of the FGCS project.

So these areas — advanced applications, parallel computer architectures, and supporting software — were identified as the appropriate targets for the FGCS project; and perhaps they will still be appropriate for the development of computing in the next century. We should consider now what are some of the problems that arose during the course of trying to achieve the goals of FGCS, and whether those problems will continue to affect the achievement of those goals in the future.

In the area of knowledge information prcessing we have the problem of identifying the scope of this class of applications. Does it aim to provide a general model for all computing in the future, so that all computing will be forced into a knowledge-based framework? Or does it address only a niche market, which is still relatively small today. (It has been estimated that the AI market currently amounts to only about 1% of the market for the whole computing industry.) If this is only a niche market, is it a market which will become more important in the future, as it merges more and more with database technology for example? It seems to me that these are important questions to address, if we are to understand the scope and significance of FGCS technologies for the future.

And how shall we judge the FGCS approach to parallel computer architecture? Was it a mistake, and will it equally be a mistake in the future, to develop seemingly specialised computer architectures for apparently niche applications? Or was it an ingenious discovery that such architectures might in fact provide a basis for building more powerful, general-purpose computers in the future?

And what about the choice of logic programming to bridge the gap between applications and parallel computer architectures? There seems to have been a number of new, mini-gaps which have appeared between different forms and different styles of logic programming, between so called "don't know" and "don't care" forms of logic pro-



FGCS: A Springboard for Information Processing in the 21st Century

What are the FGCS technologies?

- Knowledge information processing applications
- Concurrent and constraint logic programming languages
- Parallel computer architectures

Logic Programming (LP) bridges the gap between applications and computer architectures

Fig. 1

gramming, and between algorithmic and specification styles of use.

The gap between the "don't know" form of logic programming, needed for user-level applications, and the "don't care" form, useful for controlling execution on parallel computer architectures, has been experienced not only at ICOT but at many other research centres throughout the world. ICOT has investigated a number of ways to narrow this gap, the most notable is based upon the model generation theorem prover. Will this gap be closed by such work? Or will it open up even wider in the future?

By comparison with the first gap, the second gap, between the algorithmic style needed for efficient programming and the specification style useful for program specification, has received relatively little attention. I shall have more to say about this second gap in my later contribution to this panel.

Finally, we should consider the alternative technologies that have become increasingly important during the past decade, such as object-orientation and connectionism for example. We should try to determine whether these technologies are compatible with, and complementary to, the FGCS technologies, or whether, as some critics argue, they are in conflict. In the second case, might such alternative technologies be a better guide than the FGCS technologies to the future of computing in the next century?

I would now like to introduce the panelists. The first is Hervé Gallaire, who is well-known as the first director of the European Computer Research Centre based in Munich, a research centre supported by the three European computer manufacturers, Bull, ICL and Siemens. Gallaire is now in charge of formulating policy and implementation of software in a large French software house, GSI, which employs over 3,500 people. He is also president of the international Association for Logic Programming.

The next panelist, Peter Wegner, from Brown University in the United States, belongs to a somewhat different school of thought. He has had a great deal of experience in mainstream approaches to software, having made important contributions to ALGOL, ADA, and more recently to object-oriented programming. He is the author or editor of approximately a dozen books on these topics; and has edited, in particular, a collection of papers discussing both logic programming and object-orientation.

The next panelist, Ross Overbeek, is associated with Argonne National Laboratories in the United States. He has had a distinguished career working in many different areas of the FGCS technologies. He has made outstanding contributions to the field of automatic theorem proving; and the group at Argonne in which he has worked has until now been the leading group in the world in this area. He has also worked extensively on the development of parallel computer architectures. Very recently, however, he has dramatically changed the direction of his research and begun to focus on applications in general and on genetic sequencing applications in particular.

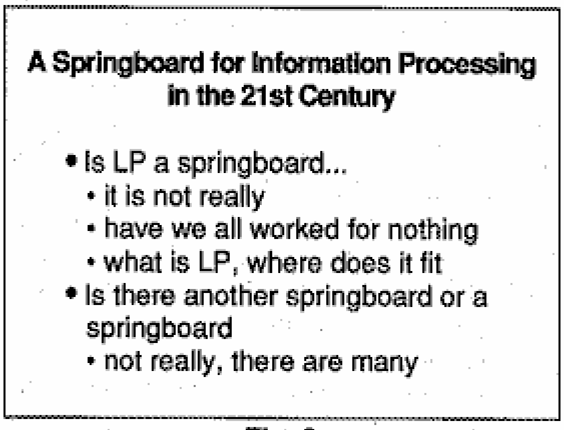
The next panelist, Koichi Furukawa, was one of the researchers at the Electrotechnical Laboratory (ETL) who was responsible for planning and launching the FGCS project. He has worked at ICOT from the beginning of the project, first as head of the second research laboratory responsible for software and the FGCS kernel language and now as a deputy director of the ICOT research centre. In addition to his role in the scientific management of ICOT, he has contributed to numerous research publications concerning all aspects of the FGCS software. His research judgement has had a major influence on the direction of research in the FGCS project.

The following panelist, Shunichi Uchida, like his colleague Furukawa-san, also worked at ETL, where he helped to plan the FGCS project before joining ICOT at the beginning of the project. He began his work at ICOT, first as deputy head of the first research laboratory responsible for computer architecture and now as research manager of the ICOT research centre. He was responsible for developing the PSI machine and the PIM architectures. It is widely rumoured that he will be the director of a likely post-ICOT project.

I have also included myself as the final panelist. It is time now to receive the presentation from our first panelist, Hervé Gallaire.

Gallaire: Good afternoon. I would like to thank the program committee for inviting me this afternoon. It's a great pleasure and a great honor to be here to discuss the future of logic programming. In fact, I remember, in '79, being invited in a much smaller audience by ETL since ICOT at the time did not exist, but in fact there were already Fuchi-san and Furukawa-san who pulled the strings, and we were discussing and comparing merits of logic programming and LISP, at the time. They made a decision later --- I think they made the right decision. It's certainly much more difficult to stand for me here today than it was back in '79 in this very small audience; but nevertheless, I think I'll try to address some of the issues that Bob has been mentioning.

We are here to discuss logic programming as a springboard for information processing into the next century. Now, you see on this slide (Fig. 2) really the answer I'd like to give to this question. Let me comment very rapidly this statement.



A Springboard for Information Processing in the 21st Century

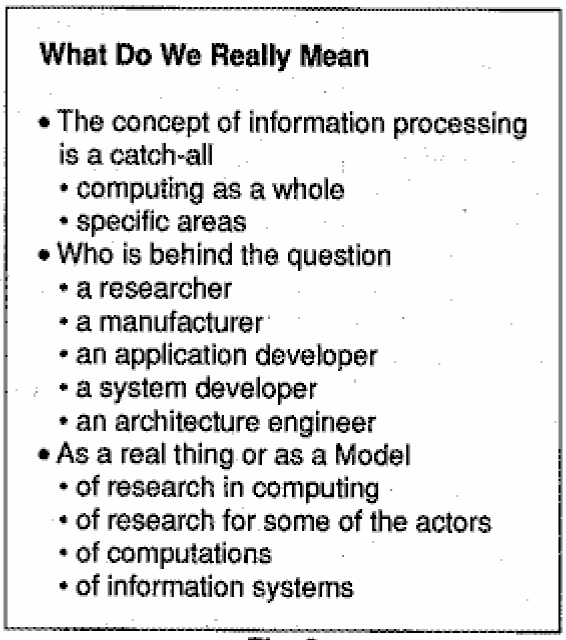
- Is LP a springboard...
  - it is not really
  - have we all worked for nothing
  - what is LP, where does it fit
- Is there another springboard or a springboard
  - not really, there are many

Fig. 2

When I say that LP will not be a springboard for information processing, what do I mean? I think the question, no matter how hard Bob is trying to make it precise, is really very general, and that there cannot be a simple answer to such a question. In short I think what we should be trying to do is find where logic programming can be helping us for processing information and knowledge rather than seeing it as a general springboard for information processing in the next century.

So, when I ask, "Do we all work for nothing?", clearly the answer is "No", and what you have seen this week, and what is going on elsewhere is clear proof that the answer is "No". But, if the answer is no, where do we fit, where should we be, and where will we be in the future? This is what I will essentially talk about, but I'd like to make clear that, in my mind, there is no other springboard either.

So, what do we really mean when we discuss the role of logic programming as a springboard? I will just use one slide (Fig. 3) here, to explain why I'm really lost when I look at information processing as a whole. I don't think we can really address that. And, if I take the time to discuss this, it's because it's really by going into more detail that we can give conclusive answers rather than too general answers.



What Do We Really Mean

- The concept of information processing is a catch-all
  - computing as a whole
  - specific areas
- Who is behind the question
  - a researcher
  - a manufacturer
  - an application developer
  - a system developer
  - an architecture engineer
- As a real thing or as a Model
  - of research in computing
  - of research for some of the actors
  - of computations
  - of information systems

Fig. 3

You see all these different viewpoints of a system: the manufacturer's, the researcher's, the user's one, etc. There is no way to ignore that they express different requirements and that logic is not going to fit them all. Further, even if we settle for the use of logic, it can be used to model the world and reason about it, or it can be the world itself. This adds complexity to the question. More, today we build systems in layers; it could be worth studying at which layers is LP apparent, for each of the views we discuss. In analysing such a complex situation, I'm saying that neither logic programming nor any programming paradigm can be the underlying foundation, except in the sense that Alan Robinson used on his talk on Monday to discuss an abstract computing device: the Turing machine. In this sense, yes, logic is the underlying paradigm, but as soon as you leave this --- I would say --- secure world, then, you are forced to see other things (Fig. 4).

But the point is that, when you really want to understand the picture I was showing before, for example to see where logic fits, you are forced to choose one direction and to push the paradigm you have chosen to the limits. Actually, by taking the decision to see logic programming as a unifying paradigm in some sense, people have pushed the walls around, and pushing the walls around, learned not just about that paradigm, but also about the other ones. It's when you try to cross the border and to push the walls further, that you really understand and contribute to progress significantly; just to give a concrete example, I think that the way the concurrency problem has been attacked in logic programming, tells us a lot about concurrency in general, and that we make progress in understanding concurrency no matter which paradigm, in the end, people will use to implement it.

Now, let us look at the question from the model and research perspective (Fig. 5); my answer, as I already said in the previous slide, is that, yes, clearly logic programming remains the major candidate to provide a coherent view of the world. I don't think there is any serious competition from that viewpoint. Saying that, does not imply that everything is easy and simple to solve, but never-

Global Computing Perspective

- hierarchy of views of the world
- multiple views at each level
- different tools needed
- different abstractions needed
- neither LP nor any programming paradigm as the underlying foundation
- but, keep trying pushing to the limits (eg, develop Hardware was a good idea)

Fig. 4

Model and Research Perspective

- the major candidate providing a coherent view
- seamless integration of the world parts
- major contributions: rule-based deductive DBMS, problem solving and AI, specific logics for time and belief, handling negation, concurrent programming, constraint programming
- in some cases, achieve dominant position
- but it will not be the unique encompassing model, adopted by everyone
- because there is too much human intelligence
- and there are too many abstraction viewpoints

Fig. 5

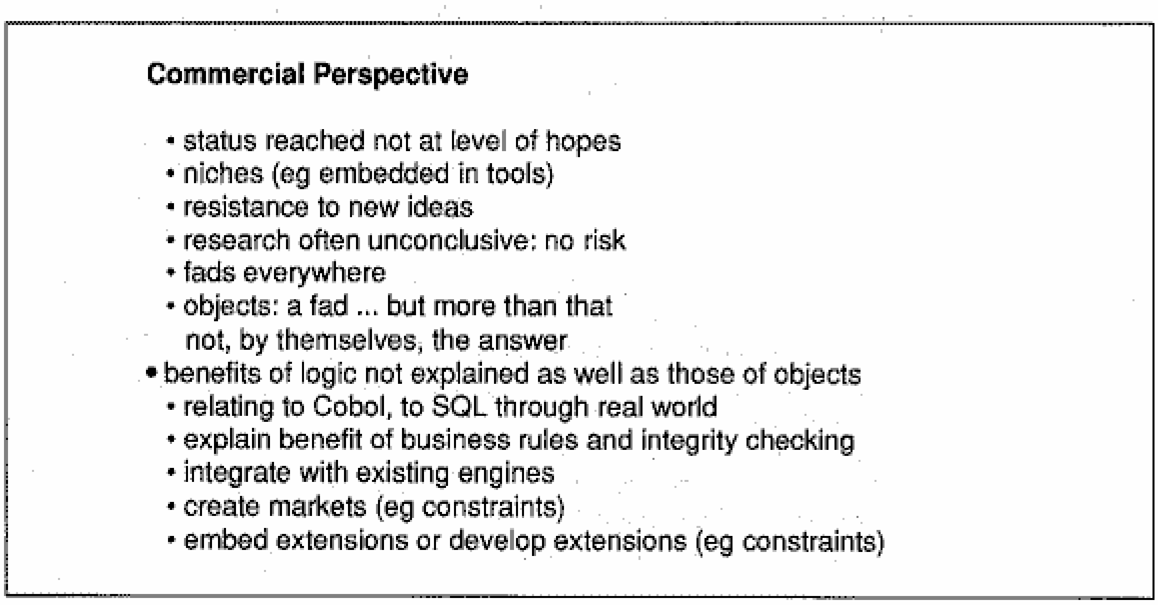
theless, I think progress has been made in a number of areas; and I've listed here just some examples, by no means trying to be exhaustive; I stress the contribution of logic to the world of deductive databases which perhaps are not as well known as some of the other contributions in the field of concurrency or of languages; the future of databases for me is no doubt connected to the future of logic, among other things, as we've learned over these years.

Similarly, the future of problem solving is linked to that of logic programming; the constraint languages are certainly a good example of the type of solutions that we have been able to find, and that will certainly flourish over the years; this will be the way to go for problem solving not just in our field, not just for symbolic computing but as well for combining numerical and symbolic computations, something required in many hard problems that will become feasible through this unique approach. Here are other examples I already mentioned: concurrent, parallel programming, and so on.

In some cases, logic has really reached a dominant position. It's not the case everywhere, needless to say; certainly we can improve the position of logic in a number of these domains, but it will never be dominating in every field. There's just too much human intelligence, I believe, for accepting that one particular way of looking at things dominates everywhere. Universal computation formalisms are a good illustration of this principle. And there is also a point I want to bring: when we try to solve a problem, we are very good at devising different abstraction levels for that problem space; people like to work at the appropriate abstraction level. Sometimes it is useful to map the solution expressed at one level to a different one, sometimes it is dangerous to do so; and it can be dangerous for logic programming, by the way, to try to map everything down to it; reasoning at the appropriate level is what we need.

Let me turn now to the commercial perspective? (Fig. 6) The answer to our question regarding the future of logic programming is simpler from that perspective than from the others. No; we've not reached a status that we were hoping to reach; whether we will do so in the future, depends very much on how we are going to tackle this problem in the near future, not just in the next century. I think we have to worry about that before we reach the next century. I'm afraid it will be too late if we wait until that time.

There is always resistance to new ideas; this could explain why we've not succeeded very well in the commercial world. I think it's not just that;



Commercial Perspective

- status reached not at level of hopes
- niches (eg embedded in tools)
- resistance to new ideas
- research often unconclusive: no risk
- fads everywhere
- objects: a fad ... but more than that
  not, by themselves, the answer

- benefits of logic not explained as well as those of objects
  - relating to Cobol, to SQL through real world
  - explain benefit of business rules and integrity checking
  - integrate with existing engines
  - create markets (eg constraints)
  - embed extensions or develop extensions (eg constraints)

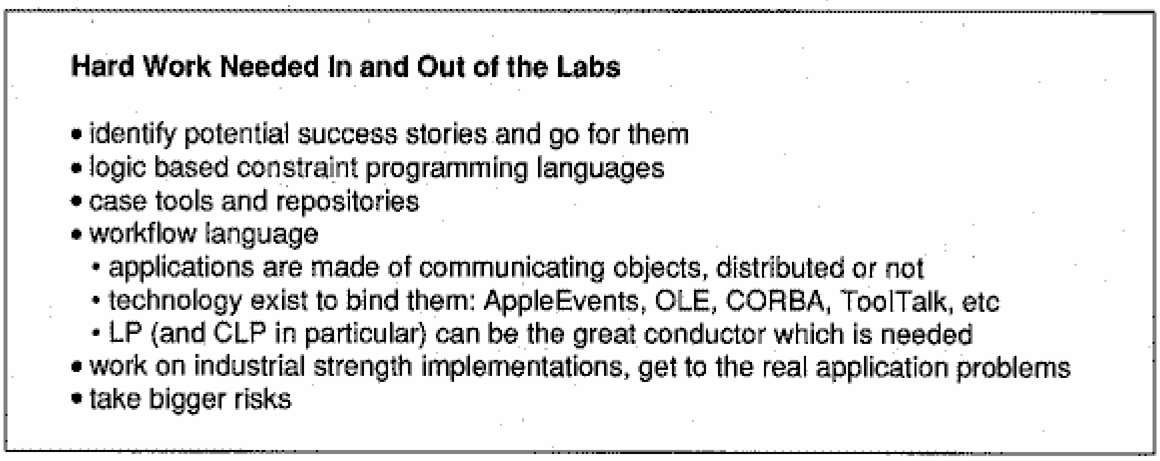
Fig. 6

it's probably that we have not made all the efforts we could have made to explain and to really demonstrate the benefits of logic programming. The world, especially the commercial world is dominated by fads. Object programming, for example, is certainly a fad in the commecial world; they understand however that there is more to it than just a fad; it brings something which they can understand; in contrast we have not explained really exactly what we could bring with logic programming; for example, we have not related deductive databases to the SQL world or COBOL well enough. I think we have to do more, and I'd like just to give an example of things we could do. Take a payroll application. I think it would be a great success if you wrote it as a logic programming package; it would be written as a concurrent logic application running parallel streams corresponding to different employees; it would also use constraint programming on the computing side, if you wanted to be able to state declaratively regulations - this needs dataflow computing and would be provided through constraint programming; it would also benefit from constraint programming if you'd connect it to a human resource package supporting scheduling or job allocation, and so on. I think that's kind of results that the commercial world could understand. We have to do that, but it will take time before we can be convincing on this type of examples.

I'd like also to observe that we are working by building everything from scratch even in the software area; for example, we are building constraint programming out of logic based computation engines. A question could be, should we do it instead by connecting to existing engines, such as pure C and forgetting about, or trying to forget about the whole world of logic programming. I don't think we can do that completely and still reap all the benefits of logic programming, but we should not ignore mix and match solutions.

So, in conclusion, I'd like to say that we need to identify potential success stories and work very hard for them (Fig. 7). I've indicated here only three areas where everything is somehow starting, where we can impact the world because the world is not really stable. I've listed constraint programming languages --- I've already given examples. I've listed case tools and repositories; this is a very immature world; the knowledge representation requirement is essential, as well as deduction techniques and thus there is a lot of potential for the type of modeling that logic programming can achieve. The third example listed here is that of "workflow languages".

Some of the people call them coordination laguages. Let me explain what I have in mind. The computing world is going to be distributed. It's going to be distributed more than being decision-based or being compute-intensive-based. Computing as communication is starting to happen, and it will be more and more so; lots of technologies are taking off the ground around the client server architecture or the cooperative architec-



Hard Work Needed In and Out of the Labs

- identify potential success stories and go for them
- logic based constraint programming languages
- case tools and repositories
- workflow language
  - applications are made of communicating objects, distributed or not
  - technology exist to bind them: AppleEvents, OLE, CORBA, ToolTalk, etc
  - LP (and CLP in particular) can be the great conductor which is needed
- work on industrial strength implementations, get to the real application problems
- take bigger risks

Fig. 7

ture, but these technologies, today, do not really address the problem of synchronization of communications and processes. They are low level, underlying technologies, that can be found in a number of products listed here. What we need is something more powerful, such as the master of ceremony. We need the conductor to organize this distributed world. I think this is really one area where with all the results we have on CLP, which in this case, is either constraint logic programming or concurrent logic programming in some form or another, that we have more than what is needed to do a good job. And I'd like to conclude on that. Thank you.

Kowalski: I propose that we continue now with our next panelist, Peter Wegner, and that we delay the discussion until after all the panelists have made their presentations.

Wegner: Good afternoon. I feel greatly honored to be a member of this panel, and especially to be invited as the only outsider, since my main field is not logic programming. As an outsider, my role is to be provocative. I will take a controversial position concerning the role of logic programming in problem solving, arguing that the object-oriented modeling paradigm will dominate the logic-based reasoning paradigm for application programming in the 21st century.

Before I get into the details, I would like to say a few things about the impact of ICOT. ICOT has made many scientific contributions to logic programming, concurrent logic programming, constraint logic programming, parallel machine architecture, parallel machine system software, and applications like genetic applications. It has also contributed to building the Japanese research infrastructure, to making logic programming research fashionable throughout the world and to creating a new model for cooperative research that has influenced the Esprit research project, and provided a stimulus for the European Economic Community. It has provided a vibrant research environment for a new generation of young researchers and some infrastructure that can be transferred to the real-world computing project. ICOT has been very successful in terms of its impact with great scientific, social, and political effects.

Are modeling and reasoning paradigms complementary, or is there an inevitable tradeoff between them? My main points are that modeling and reasoning paradigms are incompatible in the sense that we have to trade off one against the other, and that the modeling paradigm will dominate the reasoning paradigm in the 21st century. Logic programming is a paradigmatic example of the reasoning paradigm, while object-oriented programming is a paradigmatic example of the modeling paradigm. The relation between reasoning and modeling parallels that between ICOT's fifth-generation computing project and the real world computing project proposed as its successor. Fifth-generation computing is problem solving by reasoning, whereas real-world computing is problem solving by modeling.

The debate between modeling and reasoning is a continuation of the 2000-year philosophical debate between rationalism and empiricism in the domain of computing. Descartes' statement "cogito ergo sum --- I think, therefore, I am" is the essence of the rationalist credo that thinking is the basis for knowledge. His development of Cartesian geometry, which reduces geometry to algebra, encouraged the belief in the possibility of reducing physics to mathematics and of computation to logic. Hume's argument that deductive logic is an insufficient basis for inductive and causal laws and therefore for empirical sciences like physics is the basis for empiricism. Hume caused philosophers like Kant to reexamine the rationalist belief in pure reason, but did not deter Hegel from adopting an extreme form of rationalism based on a flawed form of reasoning (dialectical reasoning) that strongly influenced mathematical philosophers like Russell.

The early 20th century was a period of mathematical rationalism, with Russell's attempt to reduce mathematics to logic in Principia Mathematica and Hilbert's program to solve all mathematical problems by mechanized mathematical rules. However, in the late 1920s and 1930s, several results showed such rationalism was not feasible, including Godel incompleteness, Turing non-computability, and Heisenberg's uncertainty principle in the domain of quantum mechanics. This caused a return to empiricism, with operationalism in physics, behaviorism in psychology, and a trend away from formalism in mathematics that continues to this day.

Object programming is an empiricist endeavor, while logic programming is both motivated by and limited by rationalism. Though logic programming does not deny the need for observed data as input to computation, it does imply that all computation once the data has been gathered be performed according to the laws of deductive logic. We argue that logic programming is a restricted form of computation that limits our ability to directly model applications and that the empiricist influences represented by object-oriented programming free us from these rationalist restrictions.

Let's consider three paradigms for computing: the state-transition paradigm, the object-oriented modeling paradigm, and the Prolog-based reasoning paradigm. The state-transition paradigm is the procedure-oriented paradigm; programs here are sequences of action, and the computing step is a state transition. With the modeling paradigm, the paradigmatic example is the object-oriented paradigm; a program is a collection of interacting objects, and the computing step is message communication. The paradigmatic reasoning paradigm is the logic paradigm; a program is a set of rules of inference and the database of facts, and the computing step is logical inference.

We understand the logic paradigm fairly well at this meeting, but I would like to say a little about the modeling paradigm. The object-oriented paradigm mimics scientific modeling, since it models entities by their observable behavior. Objects model things by their observable attributes, namely by the set of procedures that determine their interactions and by the set of messages that are meaningful for each object. In doing so, they specify "what" by all potential "hows", providing a declarative specification by all potential behaviors. The object-oriented paradigm therefore captures the interdisciplinary essence of modeling in the domain of computing.

Let's modify Bob Kowalski's divide-and-conquer aphorism "algorithm = logic + control" to:

program = declarativeness + control
= "what" + "how"

This change is insignificant if we equate declarativeness with logic as in logic programming. However, it becomes significant if we broaden the definition of declarativeness to include other ways of specifying what is computed independently of how it is computed. Object-oriented programming specifies objects (things) independently of specific computations (actions) and therefore has a claim to be called declarative. It specifies objects in terms of all their possible effects without commitment to a specific effect. The declarativeness of objects is very different from the mathematical declarativeness of logic programming, but is in some respects more useful because of its close similarity with the way "things" like tables are specified in scientific modeling.

The control mechanisms for objects is also very different from that of logic programming. Control at the object interface can be specified by a select statement of the form:

select (op1, op2, ..., opN)

where op1, op2, ..., opN are the operations or methods of the object. This select statement is implicit in objects but becomes explicit in concurrent objects or concurrent processes such as those of Ada or CSP, where it is a version of Dijkstra's guarded commands:

select (G1|op1, G2|op2, ..., GN|opN)

The select statement is essentially a read-eval-print loop which waits for a message and then executes it. Objects are locally non-deterministic. They do not know what they will do next. They sit there waiting for an input; and when an input arrives, they commit to that input; so this is a committed choice form of non-determinism. The notion that control at the object interface is realized by a committed-choice select statement is usually thought of as a property of concurrent objects, but it applies also to sequential objects.

Next, I will consider the role of open systems in modeling. The term "open systems" suggests incremental, scalable, modifiable, adaptable, evolving systems. Openness is a property of modeling that is not as captured by logic. There are two kinds of incremental change: reactiveness by dynamic response to stimuli, and encapsulation that facilitates static extensibility of modular systems by programmers. Reactiveness captures temporal evolution of organisms as in biological evolution, while encapsulation captures external modification as in software engineering:

open system = encapsulated + reactive
= spatial extensibility + temporal evolution

A system will be called strongly open if it is both statically or dynamically modifiable. Objects are quintessentially strongly open; they are reactive in that they can react to their environment and they are modular and easily extensible. Logic programs are not strongly open; they are not reactive in a technical sense discussed below. Although they can be statically extended by adding clauses or adding facts to the database, they do not model the application domain as directly as object-oriented systems so that a small change of the application domain does not necessarily translate to a small change of the program.

Let me talk next about why logic programs cannot be reactive. Reactive systems compute by irrevocable side effects on their environments, since their reaction to stimuli cannot be taken back. Pure logic programs cannot react till a theorem is proved: they cannot make an irrevocable commitment until they are sure the result is true and cannot therefore be reactive. The property of being able to take things back as though they had never happened, called retractiveness, is incompatible with reactiveness.

Concurrent logic programs sacrifice retractiveness, essential to logical completeness, in favor of committed choice non-determinism. In giving up logical completeness to realize reactiveness they adopt a control structure very similar to that of object-oriented programming. To see this, consider the set of all clauses of a concurrent logic program with the same predicate P in the clause head:

```
P (E1):- G1 | B1 - head P (Ei), guard Gi, body Bi
P (E2):- G2 | B2 - reducible if unifies (E, Ei)
        and Gi
...
P (En):- Gn | Bn - irrevocable committed choice
```

This corresponds to the set of all clauses that could potentially be invoked when a goal P (E) is to be proved. In a pure logic program, all clauses would be tried until either the goal is proved or all alternatives are exhausted. We call such exhaustive search through all alternatives don't-know nondeterminism because the system need not know which alternative leads to a correct solution, since all will eventually be tried. Don't-know nondeterminism is often implemented by backtracking and conflicts with reactiveness. Concurrent logic programming systems cannot handle don't-know nondeterminism and adopt committed-choice (don't-care) nondeterminism that permits reactiveness because there is no longer any requirement to revoke the effects of a commitment. The resulting control structure for choosing among alternatives when satisfying a given goal P (E) is exactly like that for choosing among methods of an object. The set of clauses for a given predicate P are analogous to the set of methods of an object.

Let me conclude by considering the design space of reasoning versus modeling in terms of two dimensions: a declarative or "what" dimension corresponding to reasoning versus modeling and an imperative or "control" dimension corresponding to retractive versus reactive. The pure logic paradigm is a retractive reasoning paradigm, while the object-oriented paradigm is a reactive modeling paradigm:

Pure logic programming (LP) $\rightarrow$ reasoning + retractive
object-oriented programming (00P) $\rightarrow$ modeling + reactive

Since "what" and "how" specifications determine a two-by-two grid, two other possibilities must be considered (see Fig. 8):

concurrent logic programming (CLP) → reasoning + reactive
distributed logic components (DLC) → modeling + retractive

How / What	Retractive	Reactive	
Reasoning	Pure LP	CPL	Shared Constraints
Modeling	DLC	OOP	Distributed Components
	Complete	Flexible	

Fig. 8 Design Space for What and How Specifications

The combination "reasoning + reactive" corresponds to concurrent logic programming. The combination "modeling + retractive" does not currently correspond to any well-established language, but is being investigated in terms of distributed logic components that encapsulate local don't-know nondeterminism but have committed-choice interfaces.

The combinations of retractive reasoning and reactive modeling appear more stable than the other two combinations. Reactive reasoning and retractive modeling systems are hybrid design alternatives. Concurrent logic programmers claim that reactive reasoning captures the best of both worlds, while researchers in other language paradigms feel that this approach falls between two stools, being neither logical nor reactive. Distributed logic components are even less well explored than reactive reasoning systems. Here again it could be claimed that combining distributed locality and completeness is an advantage, but it is more likely that local completeness will be found to be an inadequate hybrid because it is neither reactive nor a reasoning system.

To conclude, I am suggesting that modeling paradigms are fundamentally different from reasoning paradigms in that they are based on a different notion of declarativeness and are reactive. There is a trade-off between object-oriented reactiveness and logical completeness such that we must choose one or the other. For programming in the large, reactiveness is more important than logical completeness, and because of this the object-oriented rather than the logic paradigm will become the dominant paradigm for programming in the large in the 21st century.

Kowalski: Thank you very much. The next speaker is Ross Overbeek.

Overbeek: Well, I believe my perspective is, perhaps, a little different from the other speakers. We were asked to speak about the role that we saw for logic programming and for parallel processing in the 21st century.

I believe to speak about that we are going to have to face and address essential questions. In our minds we are going to have to ask:

> "When will the very best applications, the best implementations of specific applications, be based on logic programming systems? And when will it occur in more than a few limited instances?"

I think that to understand the role of this technology we have to look at such questions, which are fundamental and basic -- and they are divorced from the basic research topics that we all have come to appreciate and love to discuss.

From my limited perspective I see technology moving through several stages of development:

1. Early explorations and formulation of basic issues
2. Growing awareness of the potential uses of the new technology
3. Experiments on "simple kernel" prototypical problems
4. Gradual adoption of the technology in a few applications
5. Widespread acceptance of the technology

I was involved in working early on in parallel applications, and I watched parallel architectures (and applications on them) move through these different stages. I vividly remember being told by people that there was no point in worrying about these machines -- that they were going to be insignificant and outclassed by sequential machines.

I believe that parallel processing has moved into the fifth stage, and it will be widely adopted. Perhaps paradoxically, I've also reached the point where I think it is, perhaps, less significant than I used to believe -- just as it finally became accepted. But logic programming is what I am more concerned about, because logic programming is the topic that I find more important. And I don't think it's clear yet what role it will play in the next century. I don't think its role has been determined, and I don't think it will be determined by its technical aspects.

I think maybe we could begin by looking at the applications that are now emerging, if we wished to gain some understanding of where this technology will go. In my mind, I break them into three basic categories.

a) There is a set of applications which I've said are based on expressive power --- that's, of course, not well stated. But what I meant by it was there is a set of applications that will use a language like Prolog because of the ease of doing prototyping, the ease of working with databases, the ease of creating interfaces.
b) There is a second category of applications based on distributive computing. The best example here is the work done on simulating telephone switches.
c) And, finally, an area I've been more active in is the coordination of parallel applications. In this context we've watched the basic ideas of committed choice logic programming gained real in wide-spread application. Indeed, until recently, I believe the Intel's Delta was one of the more powerful machines available, and I think it was the case that the first application brought up on it was based on PCN, a language which has the same intellectual roots as KL1.

So, you have these different areas. What strikes one immediately is the point that Bob brought out, that there isn't a unifying principle here beyond the general framework of logic, that these really are wildly different classes of applications.

So, one might ask, "Will unification occur ?" --- will there be a unification between the dialects like Prolog and the dialects based on committed choice? One reaction a person could have is "no -- essentially we have a very fast moving technology, one that will drive applications like Prolog and UNIX, C workstations, that the mass software market will dominate that area, that the committed choice applications will probably be limited to extremely large machines, probably doing largely numerical computations". It's my own, somewhat unorthodox, opinion that we should build upon these two environments and not attempt a unification.

And then, there is the more dispassionate view, perhaps; some leaders may feel that one cannot go wrong by first trying to understand an area thoroughly, to work out an appropriate solution before attempting to apply it. This has never been my tendency, unfortunately; but I can understand the position, and I can understand why there are groups that take this view.

I would like to advocate that you, at least for a moment, take the perspective of someone of developing a large application in this technology -- that you force yourself to consider the world from the point of view of a person developing an application; not of advancing the technology but attempting to develop immediately and rapidly the best implementation of a given application, whatever it is. I think you immediately, in the case of logic programming, see a trade-off between two basic factors:

1. One is the size of the market that is using the technology of logic programming. You're faced with reducing the market for your product, if you tie it to technology that is not widely available.
2. On the other hand, there is the cost of development. This is the strength of logic programming in many ways; that based on declarative logic and ease of expression, one can reduce the cost of developing functional applications.

But to understand this trade-off, we have to look at issues that occur to anyone building products. I have to wonder what would happen if I listed the computing languages that have been proposed, developed, and used over my period in programming? I have to wonder how many computer scientists could pick out C as a significant language if they were not aware of the history; if they based their decision upon just the characteristics of the language.

I believe that C attains its significance due to many factors; many of them totally unrelated to its intrinsic characteristics as a programming language. Similarly one could consider UNIX -- why did UNIX become important? What were the factors that produced its current popularity? And even stranger, why is MS-DOS playing such a major role in the world today? I think we have to come to grips with these questions if we want to understand the role of logic programming in the next century. The factors that determine the spread and adoption of technology are quite different than those that computer scientists normally think about.

Now, I would like to move into a discussion of the applications that I work on, because I am actually fairly optimistic about the use of logic programming. Indeed, I work on biological applications now most of the time; one of the essential application areas is integrating databases relating to genetic sequence data. And although people will tell you that these databases are destined to be huge, complex, and that they will strain the technology of the coming decades, I don't believe that's true at all. I believe that what is really important about them is that they have a complex structure, that they're hard to integrate, and that it's precisely the strength of logic programming and logic that are required to turn this massive data into a functional resource. So I actually believe that the best integrations will grow out of logic programming efforts, some that are done in Japan, some in Europe, and some in America. Further, I believe that it is already recognized within the biology community as probably being one of the most likely approaches to a successful integration. I'm very optimistic about an area like biological databases; and I believe that we should search for other areas like that.

Another area reflecting another aspect of the logic programming paradigm is work I've done in phylogeny with Hideo Matsudo of Kobe University and a group at the University of Illinois. Phylogeny involves reconstructing the tree of evolution -- attempting to recreate, in some sense, the ancestral sequence; to trace back and gain a perspective on the universal ancestor. In this application performance is critical. If one runs it on a workstation, if one uses the standard computations based on maximum likelihood, the computations would run for years on workstations. So, parallel computation is utterly critical.

We developed a program based on maximum likelihood, ran it on the Intel Delta, and we constructed a phylogenetic for microorganisms that included 500 organisms; this is the first time that people had used this technique to build trees for more than 15 or 20 organisms, so this is one of the limited applications for which a parallel processing is utterly required. And, in what is a common phenomenon, we ran into severe load balancing issues. The original applications was recast in a dialect in PCN (invoking C subroutines); and this reflects, in my view, a common and important use of committed choice logic programming. Essentially, this is what people are referring to as a coordination language. For the dialect to be significant in this context, we must be able to link it successfully with existing application codes; the existing investments in both C and FORTRAN and in the numerical community are just too large to ignore.

Finally I would like to talk briefly about another area of applications I'm working in: geographic information systems. I now do work consulting for some industrial companies on where to site stores. It's a database activity; it is one where you create a database of information about geographic areas and then you use this database to support statistical modeling. What I found surprising is that the most cost effective way to proceed was using Prolog. We routinely construct databases now that contain over half a million clauses, and they work extremely well on relatively inexpensive hardware. Certainly the cost of the hardware in the system is far less than the personnel, the cost of the project is clearly dominated by the cost of personnel, and the potential advantages of flexibility are overwhelming. That is, if the application works correctly there is at least hundreds of millions of dollars that could be saved. If we could site stores more quickly and more accurately, the cost of the personnel or the cost of the computing workstations is almost totally insignificant. What is important is giving the right answers as quickly as possible. Now, there are obvious drawbacks in using Prolog to do this sort of activity. But what's interesting is that it is the most cost effective way at this point in time to do it.

So, that will be all for my initial statement.

Kowalski: Thank you very much. We continue now with the next panelist, Koichi Furukawa.

Furukawa: I would like to discuss from the standpoint of a scientific sales manager of FGCS Product Incorporated.

To sell our products, we need applications; and the applications must be such that no other approaches to them can compete ours.

The most essential requirement on the quality of such applications is that they must extensively rely on heavy symbolic computations because that is the most strong feature of our machine. The next question is, then, "is heavy symbolic computation crucial in real life applications?" My answer to this question is "yes". We can currently find only few such applications which require heavy symbolic computations. However, there are many hidden application areas whose bottle neck is the huge amount of symbolic computation and therefore no one has tried to solve them so far. Such problems include knowledge acquisition from database, data analysis, many combinatorial problems such as human genom analysis, so called inversion problems such as diagnosis, disign, abduction and constraint satisfaction. In the term of logic, inversion problems are interpreted as non-deductive problems. In such problems, you have to guess the structure of some machinery from input / output behaviour, unlike to guess input / output behaviour from its structure; thus we call such problems as inversion problems.

There are two important aspects in logic programming: as a tool for representing knowledge and as an inference engine. From the knowledge representation's aspect, it has the capability of expressing open world by negation as failure. It has been commonly believed that logic programming is just for deductive power, but this is not true. A theory of stable models provides a model theory on general logic programs including negation as failure. Also, these formulations provide natural formalization of abduction, induction, and analogy.

From an inference engine's aspect, it is well known that automatic searching and combination of top-down and bottom-up search strategies provide powerful inference capability.

For the problems of handling negation as failure and abduction, Inoue proposed a method to compute them by deductive procedure. That is, negation as failure and abduction problems are translated to proof problems of first order logic, and therefore non-deductive inference problems can be translated into deductive problems. As a result, this method provides a straightforward way to solve a class of inversion problems related to abduction.

The most important feature of concurrent logic programming for our project is, of course, its expressiveness of concurrency. The feature is crucial for general purpose concurrent programming; many useful and complex applications were written in KL1, including the operating system of PIM. To overcome the apparent weakpoint of our concurrent logic programming language, we tried to recover completeness, or equivalently OR parallelism, by inventing several search programming methods for computing all solutions. As a result, we succeeded to effectively recover the completeness property in our concurrent logic language. Therefore, in our case, we have now both reactiveness and completeness.

In order to solve logical inversion problems, or equivalently abduction problems, by PIM, we need appropriate programming methods. Since abductive inference can be translated into deductive inference, we can achieve the target simply by realizing an effficient parallel theorem prover in KL1. Fortunately, we have already developed a very efficient parallel theorem prover called Model Generation Theorem Prover (MGTP) which proves theorems by bottom-up procedure.

From these advantages I pointed out, it is very natural to conclude that concurrent logic programming is expressive enough for both concurrency and searching. Also, it has affinity to parallel computation, therefore, it provides very appropriate bases for artificial intelligence systems and for more general information systems in 21st century. Concurrent logic programming and parallel processing based technologies have great potential for solving many future AI problems.

Kowalski: Thank you. That was an amazing performance, compressing so many exciting ideas into such a compact presentation. Let us continue now with the next panelist, Shunichi Uchida.

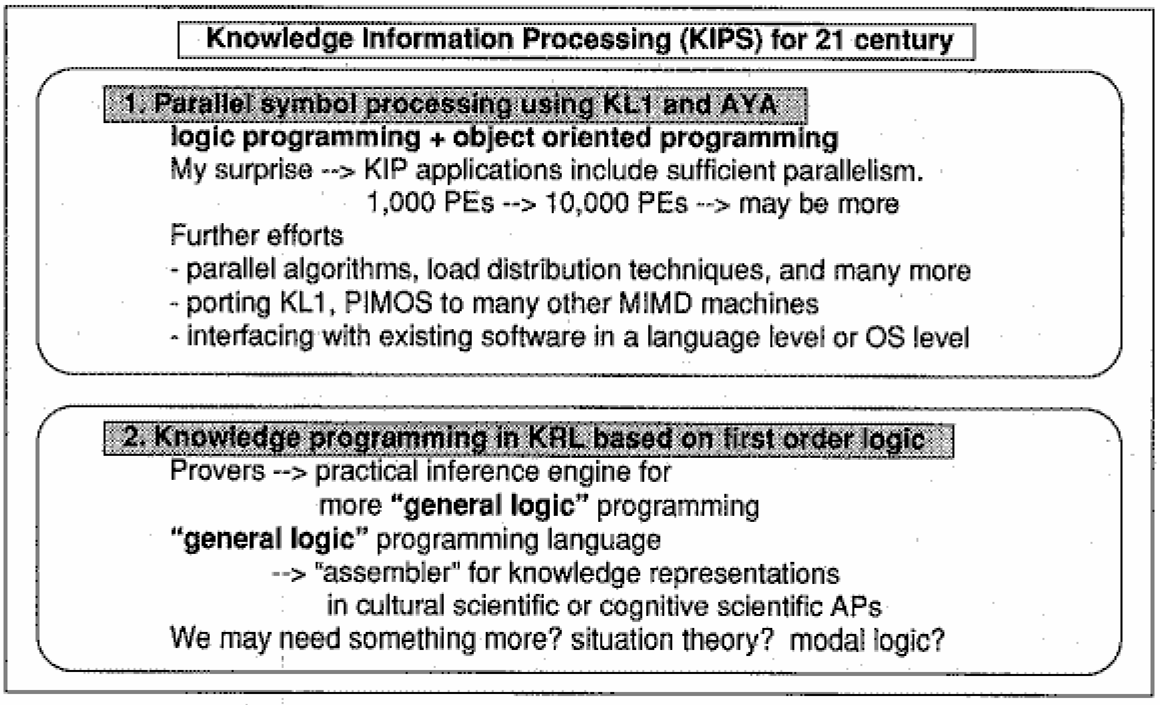
Uchida: Good afternoon, ladies and gentlemen. As one of the key members carrying out this project, I would like to thank you for the great interest many of you have shown in this project, and I am also very proud of the very high rating given to this project by many of you. Thank you very much.

I would like to present a slightly different viewpoint from the other panelists. As my background is in hardware architecture, I am now very happy to see that the machines we have built have opened up new application fields, not only in engineering but also in very scientific applications like theorem proving. Here, I would like to mention the two topics shown in this slide (Fig. 9) and stress that logic is a very appropriate and useful backbone for future computer science.

First of all, I would like to mention about knowledge information processing in the coming century and insist that we should proceed with research on two topics. One of these is, of course, parallel symbolic processing. In our case, we will use KL1 and its higher-level extension, AYA, which is regarded as a combination of logic programming, KL1 and object-oriented programming.

Quite recently we have attained very high utilization ratios for many processors in several application programs such as a theorem prover and a protein sequence analysis system. We were surprised at this result. Honestly speaking, before this occured, I did not believe that we could attain almost linear speedup by using many processors.

So, now I have much stronger confidence that, if the hardware people can provide the software people with larger scale multi processors, say, with more than ten thousand PEs, then linear speedup may again be obtained. So, then, I am



Knowledge Information Processing (KIPS) for 21 century

1. Parallel symbol processing using KL1 and AYA

logic programming + object oriented programming

My surprise --> KIP applications include sufficient parallelism.
1,000 PEs --> 10,000 PEs --> may be more

Further efforts
- parallel algorithms, load distribution techniques, and many more
- porting KL1, PIMOS to many other MIMD machines
- interfacing with existing software in a language level or OS level

2. Knowledge programming in KRL based on first order logic

Provers --> practical inference engine for
more **"general logic"** programming

"general logic" programming language
--> "assembler" for knowledge representations
in cultural scientific or cognitive scientific APs

We may need something more? situation theory? modal logic?

Fig. 9

very sure that great research and development efforts shall be applied to the research and development of parallel algorithms, load distribution techniques, and many more parallel processing technologies. Those efforts will surely create a very big market, and, in the near future, the speed of this creation will be influenced by such efforts as introducing KL1 and PIMOS into the many commercially available MIMD machines which will appear in the next few years.

So, then, another important effort shall be to provide tools or software to interface our systems with existing software at the language and operating system levels. So, those efforts will accelerate the creation of new markets and also extend various research and development fields for parallel processing applications. It is likely that those technologies will be realized in the next five to ten years.

Then, we want to foresee what will happen after that. Recently, I have tended to see technologies from a much more macroscopic view. I believe, this is because I was assigned as a manager, but sometimes other people say that it is simply because of my age. But, anyway, recently I have really wanted to see what shall happen after these new kinds of technologies are realized and I try to imagine what future knowledge processing will be like.

The lower half of this slide shows an image of the future. This image makes me more confident that logic programming is a very nice backbone, especially for hardware architectures. Of course, I agree with other opinions, such as Peter Wegner's, that object-oriented models are indispensable for modeling real applications. However, from a bottom-up view, namely from the hardware viewpoint, I feel it difficult to find out which are the core operations or computations of the model to be implemented at the hardware level.

First-order logic or some restricted logic model is clearer and more understandable to us. Thus, I still prefer on the same backbone, namely, logic. I guess that the dreams of computer architects for the next decade revolve around more general inference engines for general logic programming. Currently, we have only very specific or restricted logic programming.

If we could have a general logic programming language as our daily tool, we can concentrate more directly on our applications. We must, first, pay attention to machines and operation systems to thoroughly understand them. After that, we can change our attention to the applications so we can program them efficiently.

So, if we could have general logic programming languages, we could probably eliminate the first step and just look at each application to include knowledge programming in it. So, in situations such as this, where knowledge programming is the main job, we feel that a first-order logic language will not be a high-level language anymore. So, probably some may say that this language is an assembler for knowledge programming.

Recently, we and myself, in particular, have had this kind of dream. I want to shorten my presentation. So, allow me to skip most of my slides.

In the future, I imagine, we will naturally have very powerful parallel symbol processing systems, and on top of this, we will have high-level inference engines as shown in this slide (Fig. 10). Today, some of these engines are being developed as theorem provers or language interpreters for object-oriented database languages. But, as long as we keep logic as the unified backbone, we can, probably, advance knowledge information processing not only in the field of natural science but also in the fields of cultural science and cognitive science.

In observing our application people writing a variety of knowledge in a variety of forms, for instance, in a legal reasoning system, I feel that we could compare the differences and qualities of the represented knowledge taken from these different scientific fields. Currently, we just have the single word "knowledge" to symbolize the quality or characteristics of knowledge; we cannot distinguish the knowledge in these fields.

So, we need an objective view to be able to compare the knowledge in the fields by using this unified language.

Here, I expect that we may enter a new era of knowledge processing. I know that there are many different opinions, still, I believe that logic can be used as the central backbone especially in research on knowledge information processing. Thank you

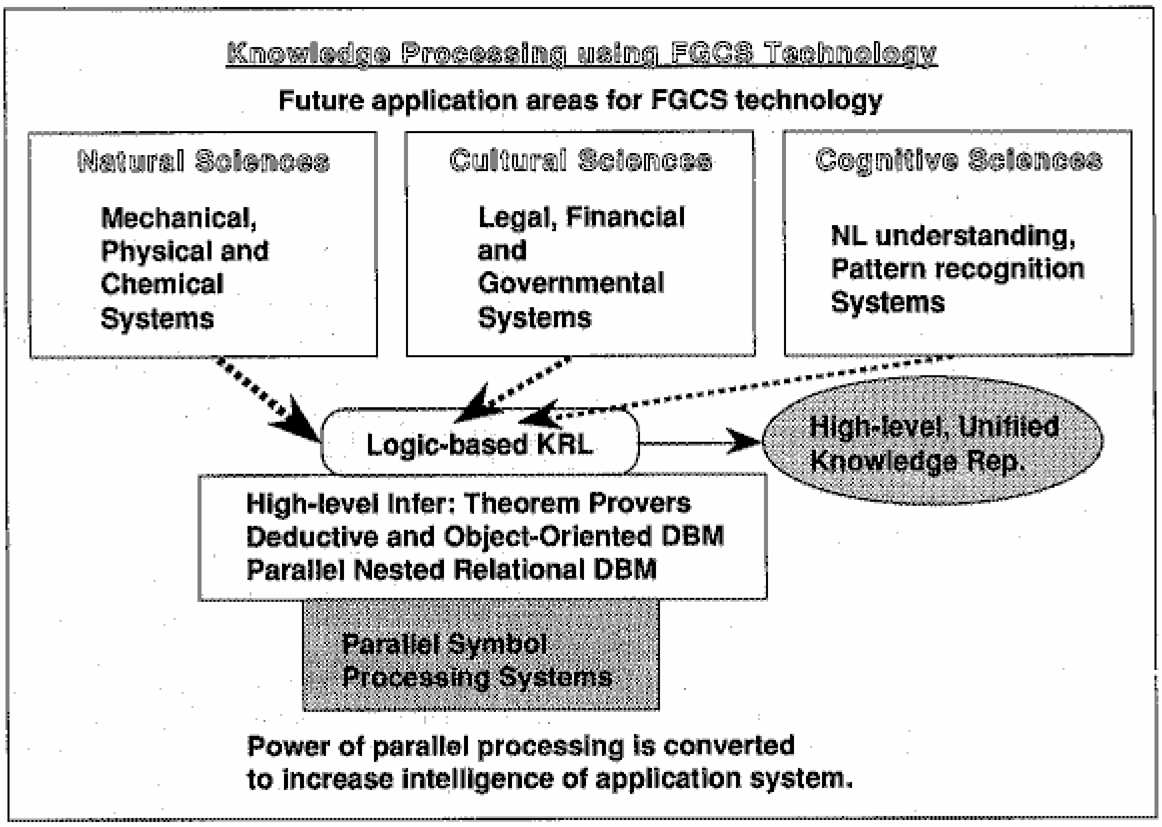

Knowledge Processing using FGCS Technology
Future application areas for FGCS technology
Natural Sciences
Mechanical, Physical and Chemical Systems
Cultural Sciences
Legal, Financial and Governmental Systems
Cognitive Sciences
NL understanding, Pattern recognition Systems
Logic-based KRL
High-level, Unified Knowledge Rep.
High-level Infer: Theorem Provers
Deductive and Object-Oriented DBM
Parallel Nested Relational DBM
Parallel Symbol Processing Systems
Power of parallel processing is converted to increase intelligence of application system.

Fig. 10

very much.

Kowalski: Thank you. I shall now change my position temporarily from that of chairman to that of panelist. I shall also try to follow the lead of my colleagues, especially those from ICOT, in being concise and to the point.

In my presentation I will concentrate on some of the problems that have arisen with the choice of logic programming in the FGCS technologies; and I will mention a number of reasons why I believe we should be optimistic about the prospects of these technologies in the future.

First, I will look at the sorting problem to illustrate the gap that has arisen between the use of logic programming for programming and its use for program specification. Then, I will look at the topic of non-monotonic reasoning, both to draw attention to the importance of the advances that have been made and to point out the contribution of logic programming.

Finally, I will draw some comparisons between logic programming and certain styles of natural language. I will look, in particular, at the language of public notices, such as the London underground emergency notice, which can be regarded as programs written in natural language to be executed by people. Such public notices resemble programs, because they are meant to be understood uniformly by different people in the same way. For this reason they need to be written in a style of natural language which is extraordinarily clear and precise. I will argue that there is a very strong similarity between such a style of natural language and the style of some computer languages. I believe that the strongest similarities are to be found with logic programming; but there are also some resemblances to be found with imperative languages and with object-orientation.

Based upon these examples, I will conclude that indeed there may be some inherent limitations of logic programming for computing, but these are the same as the limitations of the clear and precise use of natural language for regulating human affairs. Moreover, because it is extremely difficult to use natural language clearly, precisely and effectively, it may be similarly difficult to write good logic programs.

Let me turn now to my first example (Fig. 11).

```
Specification

Declarative   Y is a sorted version of X
                if Y is a permutation of X
                and Y is ordered

Procedural    to sort X into Y,
                generate a permutation Y of X
                of X test that Y is ordered
```

Fig. 11

Most of you will have seen this specification of the sorting problem many times before, written explicitly in its declarative form. Interpreted as a logic program, this specification becomes a procedure, in fact several, different procedures depending upon the mode of use. Given an input data object X, the procedure is a "don't know", non-deterministic procedure which generates permutations of X and tests them for orderedness. The procedure is extremely inefficient, having n factorial complexity, where n is the length of the input X. Many enthusiasts who are initially attracted to logic programming languages such as Prolog, because of the possibility of running specifications, rapidly become disillusioned because of such inefficiency.

But the inefficiency can be avoided by using an efficient algorithm, such as quicksort instead, which can also be written in logic programming form (Fig. 12).

Notice that the algorithm can be written in both declarative and procedural styles. There is no inherent reason why the traditional declarative form, in which logic programs are normally written, should be more acceptable than the procedural form. There is no reason why logic programming languages should not allow both declarative and procedural syntax.

```
Program

Procedural   to sort X into Y
               split X into X1 and X2,
               sort X1 into Y1,
               sort X2 into Y2,
               merge Y1 and Y2 into Y
Declarative  Y is a sorted version of X
               if X can be split into X1 and X2
               and Y1 is a sorted version of X1
               and Y2 is a sorted version of X2
               and Y is a merge of Y1 and Y2
```

Fig. 12

Thus we see that the decision to use logic programming for representing programs or specifications is distinct from the decision to represent knowledge declaratively or procedurally. I believe that logic programming enthusiasts, both novices and experts, are not always clear about these distinctions; and this may be one reason why logic programming has not been more successful in the past. Many enthusiasts have been unable to distinguish between programs and specifications and to learn how to use logic programming in a high-level, but still efficient, manner.

Thus we have a software engineering problem. But it is one which can be overcome and which in the future can be the source of a great richness of programming styles within a single logic programming paradigm.

Let me turn now to my second topic, non-monotonic default reasoning. I believe that in the last decade in the fields of artificial intelligence and logic programming we have made a great advance in making logic more usable, by making it better able to represent default reasoning. Until now, if you wanted to apply formal logic to real world problems in areas such as economics and politics, you would run into problems, because traditional logic is only able to represent rigorous, exact statements which hold universally.

Non-monotonic logics allow us to reason with inexact statements such as

"all birds fly"

which hold by default, but are over-ridden by exceptions such as

"ostriches do not fly".

This kind of reasoning can be represented effectively in logic programming form using negation as failure:

> X can fly if X is a bird
> and not X is unable to fly
> X is unable to fly if X is an ostrich

Logic programming gives us a convenient formalism, a sound semantics and effective proof procedures for default reasoning. This is a great advance in our ability to apply logic in practice, both by means of computer and directly by human beings without the aid of computers. I believe that the significance of this advance is another reason why logic programming will have an important role to play both inside and outside computing in the future.

But the topic of non-monotonic reasoning has another lesson about the sociology of research, that different research communities can work on similar problems without being aware of the similarities. The so-called "Yale shooting problem" is a well-known example of a problem which arose in the field of artificial intelligence and proved to be a counter-example to many of the major approaches to default reasoning. Despite a large body of literature devoted to the problem, it was years before it was realised that negation as failure in logic programming provides a natural and effective solution to the problem. The essence of the solution is to interpret the negative condition in the situation calculus frame axiom

> property P holds in the state after action A
> if property P holds in the state before action A
> and not action A terminates P

as negation as failure.

The history of the Yale shooting problem teaches us that sociological problems of competition and cooperation between different research communities can have just as much significance as technical merit. On the other hand, the international collaboration fostered by the FGCS project has also taught us that greater understanding can exist between people working within the same technological community in different cultures than between researchers coming from different technological communities within the same culture.

My last example is the London underground emergency notice, which can be regarded as a program to be executed by people. Because it is likely that future computer languages will be more like natural language than they are today, this example may give us an idea of some of the features of those computer languages of the future (Fig. 13).

<u>LONDON UNDERGROUND NOTICE</u>

Emergencies

Press the alarm signal button to alert the driver.

The driver will stop immediately if any part of the train is in a station.

If not, the train will continue to the next station, where help can more easily be given.

There is a £50 penalty for improper use.

Fig. 13

Notice that the first sentence has a procedural form, but a declarative interpretation in logic programming form:

> You alert the driver if you press the alarm signal button.

This example reinforces the suggestion I made earlier that logic programs should allow both declarative and procedural syntax. Other examples would show that they could also beneficially allow an object-oriented syntax, without sacrificing logic programming semantics.

Still other computational paradigms seem to have analogues in other public notices; for example the notice:

> Please give up your seat
> if an elderly or handicapped person needs it.

This has the form of a condition-action production rule. It can also be interpreted in logic programming terms as an integrity constraint. Similarly, the notice

Do not obstruct the doors.

has the form of an imperative statement. But it too can be interpreted as an integrity constraint.

Thus the language of public notices, natural language used as a programming language to be executed by people, bears resemblance to many different computing paradigms. But it integrates them gracefully into a single uniform language. I believe that logic programming can be extended in a natural way, to include procedural as well as declarative syntax and integrity constraints as well as procedures, to provide a formal basis for such a form of computer executable natural language.

I would like to finish my presentation now by arguing that the resemblance between logic programming appropriately extended and the language of public notices suggests that logic programming may have limitations as a computational paradigm, similar to the limitations of precise natural language for human communication.

In the same way that it is extremely difficult to use natural language clearly and precisely, it may be difficult to write effective logic programs. In the same way that humans communicate with one another both verbally and non-verbally, by pointing and drawing pictures for example, computer languages of the future may need to integrate verbal, linguistic means of communication with non-verbal computational mechanisms. Nonetheless, like natural language in human communication, it seems certain that logic programming will have an important role to play in the future of computing.

That completes my presentation and the presentation of the panel as a whole. I would now like to invite members of the audience to join the discussion, either by making points which may have been missed, or by responding to provocations which may have been made by the panel.

[Questions and Answers]

Randy Goebel (University of Alberta, U.S.A): I'd like to respond to one of the provocations, in particular, Professor Wegner's.

In his history he missed out somebody, and I want to ask him how he would classify that person. In the '30s there was somebody who told us that we could think of the world in terms of objects and relations amongst them, and if we took that view of a world, whether that world was concrete or abstract, we could use that specification related to logic, that is the syntax and proof theory of logic, and that our understanding that is the relationships between the models that we were looking at could be discussed in terms of truth, and that we could simulate reasoning with proof theory. That person was Alfred Tarski; I claim he was the inventor of object-orientation, and that there is a fundamental confusion between what you call a model and what the logic programmer sees as a relationship to the world that he models with that abstraction in mind.

Wegner: There is a difference between formal models and descriptive models, and in object-oriented programming we describe modeling informally and use the term model to mean a representation or a simulation rather than a mathematical model. Descriptive modeling is defferent from model theoretic modeling.

Tarski was concerned with models that could be justified formally, and his results are not applicable to formal models. We have to give up justifying what we do formally in order to realize descriptive flexibility in modeling the real world. In my talk I specifically addressed the tradeoffs between formal and informal modeling, showing that formal modeling requires a sacrifice in communication flexibility (reactiveness).

Kowalski: I wonder if any other panelist would also like to respond to the question.

Gallaire: I am not sure I understand the difference you were making between reasoning and modeling because they are not independent. If you take constraint programming for example, I believe that the way you use it today, it's not modeling, it's not reasoning, it's both; the reasoning capabilities that you get through the constraint mechanism, give you a new modeling power: you don't model a problem in the same way depending on the fact that you will have or not have a constraint solver. So, the solver embeds a lot of modeling power at the same time; thus I don't think you can say that modeling and reasoning are purely orthogonal. I think the link between the two are much more intricate.

You can find lots of examples in operation research; look at the way they have modeled classical problems, and how, if you use constraints, the way problems are modeled; it's completely different, maybe more efficient, some cases will be less, but it's completely different.

Wegner: I think we should probably go on to a different point because we could go on discussing this one for ever.

Kowalski: Please state your name and affiliation, please.

Bob Keller (Harvey Mudd College, U.S.A.): I want to ask a question of two of the speakers, but I want to proceed with the comment which is probably heretical in this audience, for many people anyway.

I think we're making a big mistake by continuing to confuse the two paradigms, logic programming and concurrent committed choice programming. And I think, if this were a formal form, I would propose a resolution that we immediately adopt nomenclature as to separate these two concepts because I think it's very confusing to the generally unitiated public, and I think it's been the cause of some of the confusion about the fifth generation project. I think the project itself has made tremendous strives in both of these areas, but I think it is still important to separate those two concepts, now we're stating the fact the either can interpret the other in some sense.

The question I had was for doctors, Gallaire and Overbeek, each of whom, during their presentations, said that they felt the paradigm --- and I believe that Gallaire was referring to logic programming, whereas, Overbeek was referring to committed choice programming, but they felt that these paradigms could serve the role as a master, coordinator or conductor among lots of other programs or perhaps programs written in different paradigms; and I was wondering what was the basis of those comments? Why did they feel that these paradigms are more appropriate than other potential candidates?

Overbeek: Well, my personal view is that parallel processing breaks into a number of distinct domains. One domain is where you have large, numerical, or symbolic applications, but mostly numerical; that's where most of the work in parallel processing is devoted at this point. If you look at massively parallel applications in that context, I believe committed choice is an excellent way to manage a message passing approach to solving those problems, and where the bulk of the time is spent in a fairly limited set of routines written in a fairly low level language.

Now, there is a completely different world in parallel processing which might be characterized by workstations with a small number of processes, and for those you're not willing to invest any effort to exploit the parallelism. In that context OR parallel Prolog offers an ideal solution, in my opinion.

But those are completely different markets, and the one that's by far the most economically significant, in my opinion, is massively parallel processing devoted towards numerical applications. I think that what we're seeing in PCN and so forth are applications of the committed choice technology to that market, and it's an important market.

Gallaire: The type of applications I have in mind or the type of computing I referred to, is precisely what is today using workflow languages. New products, new packages which come on the market, which allow you to describe your own application in terms of existing applications which are distributed somewhere in the network. If you look at what they are able of achieving, it's still far from what you would want to do for complex applications building; there are no synchronization techniques; I do believe that the need is for committed choice features plus constraints; as a user, I do not exactly know in which order things will be done; I want to solve systems, and the application is the solution of the system; this is what I was referring to.

Furukawa: In my talk, I partially answered your question, I think. I prefer committed choice than Prolog --- I mean, DON'T KNOW non-determinism. There are two reasons: one is that, as I presented, the search paradigm or completeness can be effectively recovered by programming methodology in committed choice language. That is one reason. The other reason is, the architecture of languages, I mean, if you want to design the information system including hardware and in a layered way, concurrent committed choice language is located in rather lower level, and there are -- maybe what I said before is equivalent. The higher level description of such is compiled, translated into lower level description. That kind of layered architecture is, that is important, and something like we devise some enzyme to solve problems. Then, put that enzyme, then the program is digested to be processed by parallel hardware.

Richard Peer (Weizmann Institute, Israel)**:** One of the few things that people in the field of concurrent logic programming languages agree on is the name of the field, which is concurrent logic programming. So, first, I would like people to respect, at least, this minimal consensus that it was reached after ten years of research.

Second thing, I think Peter Wegner gave a very good exposition; even though he said he's an outsider, I think one of the few things that, sort of, were finally understood by the community who was not working directly in the field is this basic principle that, if you need to express reactiveness, you must have indeterminacy. In this sense, concurrent logic languages are not different from A'UM, actors, Ada, any other language which tries to express concurrency. No one calls Ada committed choice ADA, or A'UM committed choice A'UM, or small talk committed choice small talk. All these languages that express concurrency need to have indeterminacy.

And the third point is that maybe there in ongoing research which is still in its beginning, but generally to the lot of interest and enthusiasm is to combine these two paradigms, and that's the work in the direction of Pandora, and similar languages which show that maybe there is a possible technical solution both from a theoretical point of view and from a practical point of view to integrate the benefits of Prolog and the benefits of concurrent logic programming languages. So, maybe they are not that far apart after all.

Keith Clark (Imperial College, U.K.)**:** I'd like to make several points, if I could. First of all, I'd like to address the remarks of Peter Wegner concerning --- I think he used the phrase, 'completeness', and 'a lack of completeness' of languages like the committed choice languages in which one could do modeling. I think that really is a red herring. There is a subset of programming in the committed choice languages when you're just doing functional programming. And it was that if you like the guards of complementary.

Now, I think you would agree that a functional language is declarative. And an execution strategy and evaluation strategy for functional language would be complete. So, we get complete for that case.

In fact, concurrent logic language is far more powerful because they've got the logical variable, the concept of incomplete messages. That, in fact, is what is used to model objects. There's even a relaxation of this deterministic case when you have complementary guards, which Steve Gregory called the "sufficient guards' property", which you also have a nice logical property, which means, you will always get at least one answer; because, of course, one of the worries, if you lack completeness in searches, perhaps one process, evaluation of one conditionable commit to a clause find a solution that no one else can find, right? But, there is a relaxation as I've said of the deterministic program in which you can still retain that property.

And interestingly, in that relaxed set, you can emulate all the object-oriented programming. So, for that case, we've got this sort of completeness as nice completeness.

So, they're the remarks on that point.

I'd like to take some more that were raised by Herve and Ross. I actually do agree, now, that we have to in some sense sell this new technology in the market place. And I think the fifth generation effort in Japan must continue. I must explain the very powerful methodologies they must have developed for their concurrent, their parallel symbolic applications. These need to be exposed. We have to show to the world why it is now a superior technology for developing fine grain heterogeneous applications on parallel machines.

I was very pleased to hear Uchida-san say that there were these plans --- well, I know they are at these plans to port KL1 and PIMOS to some stock hardware running UNIX and to have interfaces either via UNIX or C to other applications. I think this is extremely important. I actually also think that there is a major application area for concurrent logic programming as a coordination language as Ross said. More than that, as a language for distributed AI applications, I would like to see distributed implementations of these languages. A whole new ball game is breaking out --- computer supported cooperative work. I think we've got a lot to do in that area as well. These haven't currently been addressed in this program. Thank you.

Kowalski: This area concerning concurrrent logic programming is certainly one where important advances have been made in the FGCS project. However, I wonder if we could address some of the other areas of FGCS technologies - such as artificial intelligence or knowledge information processing, which have received so much media attention; or the parallel inference machine which could have important implications.

Pierre Deransart (INRIA, Rocquencourt, France): I think that computer science has a very short history, and logic programming, an even more shorter history. It's quite strange to ask the question about the future of logic programming with a so short history and so promising results in some sense. But it's natural also after less than twenty years of development to ask this question to oracles, please tell us what is the future. Did we work well or not, and so on. So, I would like to give the answer, a very short answer from the part of research. I think that we know the answer; we know what we are doing, and we know what we cannot do, also.

We know that basically we are working on reliability. We want to improve the reliability of software technology. We want to solve new problems also, for example, constraint logic programming permits to solve new problems which would be very difficult to handle with classical technology in software technology. So, we know what we want, and we are trying to make some progress at this track, but we know the limits also. But, we are living in a difficult time in some sense, because on one side, in software technologies, software industry is something special; because we are trying to build things which are impossible to build. If you are trying to build a building, you will use very fine technology to be sure that the building will not fall or the bridge will not fall down. But, in software technology, we are asked to produce things which we know are impossible, namely, safe software or absolutely safe software. So, any step in this direction is an interesting progress.

And on the other side, the period is interesting because industry, sometimes, and more and more is asking research --- okay, we have difficulties; what should we do? Could you help us to decide what we have to do, or to help us to find what we have to do? So, as I've said, we know in some sense, we know partially what we can do; we know also what we cannot do. But, because we know the limits, and they are strong limits, we need funding to continue this research. Imagine that, if you ask physician to find nuclear fusion by the end of the year, so, if you don't find nuclear fusion by the end of the year, so we get the funding. So, it is a nonsense. This may happen. It's not impossible not to happen; but it seems natural in computer science just to ignore that we need funding to make more improvement.

And I think that the question then is, did we do a step forward, did we do some progress since ten or fifteen years in terms of software reliability? So, this question, we cannot answer it directly. Just to look at what happens in the industry, there have been different event, for example, ICLP in Paris with industrial exhibition, and it was clear there were many interesting application and many satisfactory application from the point of view of the industry also. And, there has been recently, in London, exhibition and conference on Prolog applications, and the message --- I have to say that most of the applications presented, industry applications, were based on classical Prolog, not on big extension of logics. We know that these extensions are absolutely fundamental for the future, but we know also that the real applications now develop on many of them on the bases of classical prolog. Also classical prolog is on the track of standardization, which is a good thing for the industry.

So, if we want to know what is the real impact, and there is one; look at the Prolog 1000. Chris Moss at Imperial College is trying to recall all the industrial applications, so, you can send him example of applications and we will see the Prolog 1000 with many application, which is a very good thing, I think very interesting thing for the future. Thank you.

Kowalski: Thank you. Does anyone on the panel want to respond?

Overbeek: You mentioned that reliability --- that you knew that what we are after is reliability. I find that a very limiting notion. That's not what I believe the vision of the fifth generation project was, although reliability is certainly a noble thing to seek. I see computing as a tool for searching for the solutions, the answers to scientific questions. What was the nature of the universal ancestor? What were the original components of the automata that became life? Is global warming a reality? These are what machines are for. And, well, notions like reliability are subordinate concepts that allow us to pursue substantial visions. Computer scientists tend to focus on notions like reliability; I believe as scientists we must go beyond that, to a far more general, far broader vision.

Murry Shanahan (Imperial College, U.K.)**:** This point relates to the issue of reliability, actually. I wanted to come back to Peter Wegner's point about formal justification. I was wondering whether he would feel safe on his flight back home if he knew that the engineers who designed the aircraft he was flying on hadn't appealed the formal principles of physics. And it seems to me that logic is to physics as --- well, logic relates to physics as in engineering. And the point of using logic programming as a paradigm in computer science is that logic is very very close to logic programming, whereas, when you use something like objects, then the distance is much greater. That's why I think that we have to appeal to paradigm like logic programming in order to make that gap smaller. And I think it's slightly scandalous to reject the notion of formal justification when we're building more and more complex systems in computer science.

Wegner: I am prepared to be scandalous, because, I think that a purely logical proof often turns out to be wrong because it hasn't taken real world realities into account. I would not feel safe in an aircraft just because of a formal proof that it would not crash. Informal common sense testing of contingencies is usually more important than formal proofs. Conforming to the laws of physics is necessary but not sufficient for safety, since there are many practical not amenable to proof that must be considered. Many of the early logic proofs about the correctness of programs turned out to be wrong later so that. I would hate be a victim of a plane crash of a plane controlled by a verified program whose correctness proof turned out to be wrong.

Fumio Mizoguchi (Science University of Tokyo, Japan)**:** Japanese is sometimes quiet, like me, so I make some comment.

As a chair of the application track panel, I must report something related to this panel. In that panel, there was a discussion about the socalled paradigm starting from knowledge presentation, constraint logic programming, inductive logic programming, and parallel programming. In that panel it was rather difficult to merge all of the paradigm, but there was some consensus to use C as a basis for programming --- there was a nice picture from Catherine Lassez showing constraint logic programming CLP, then, at the end, the future C or C++. So, if we need a paradigm different from the paradigm that the language naturally provides what should we do for this kind of the exploration of paradigm? If the paradigm is one thing, and then the computation is conventional language such as C, then what should I do for the separation of the paradigm and computation?

So, I have some questions about the roles of the parallel language and logic programming language. Is it alright to use them as thinking paradigms, or as language devices to develop explorations of the paradigm? And if the basis of every computation is provided by C or C++, then, what is going to happen in the next century?

Kowalski: Would any of the panel like to reply?

Furukawa: I think the importance of the language is for helping thinking your own idea as a program, and such high level languages like Prolog or KL1, GHC help your thinking. And the merit of C or C++ may be for its efficiency. But, I think that there are many good compilation technique and that kind of merit could be imported in higher level language scheme. So, I think importance is productivity for software production and ease for thinking. Therefore, the role of higher level language is very important.

Uchida: I'd like to mention the differences between OR-parallel Prolog and AND-parallel Prolog, and also the differences between logic programming methodologies and conventional programming methodologies in C or FORTRAN.

When you are using FORTRAN or C, you are very close to the hardware. To begin with, you have to think about the details of the hardware resources you are given to use. But, using logic programming, your position is slightly higher than in FORTRAN or C. You can forget about the details of hardware resources, such as the maximum size of your memory. So, you can directly face the application problems.

However, you have to think of how to model your application. This is another step in programming—the modeling of the problem. You have to design an algorithm, write a flowchart and program, consider the details of hardware resources and so on. You have to think of many things at the same time.

In logic programming, this procedure becomes much simpler. This is one merit of high-level languages. OR-parallel logic programming is much higher than usual AND-parallel logic programming in terms of resource management. So, the last slide Bob showed us indicates one limitation of logic programming. He is trying to use one simple logic programming language to directly handle natural languages or pictures. The range of the applications is too wide for us to cover with a single language. So, this seems like a curious approach to me.

I still believe that current logic programming languages are still low level and are used mainly to control hardware systems. No efficient OR-parallel logic language is provided. We must think about how a language hides details of the hardware system. This point is very important when comparing OR- parallel language and committed-choice AND-parallel language. I consider this to be the most important point.

Kowalski: Perhaps I should respond just briefly. My last slide was the worst case scenario. We have just a few moments left; and we have one more contributor.

Catherine Lassez (IBM T.J. Watson, U.S.A.)**:** I just want to make a little clarification on what Mizoguchi-san has said. In my slide, C was actually meant constraint, not the programming language C, and although in some cases we use C, yes, but I also advocate strongly the use of formal model along the line of logic programming to apply in other domain.

Kowalski: Thank you. I'm conscious that there is one area that the panel discussion has not yet addressed, and that is the area of parallel inference machines. Is there anyone who would like to comment on their significance?

Wegner: I nominate Bob to make a comment on the significance; because I think that he feels it's significant.

Kowalski: Thank you. I do indeed believe that ICOT's work on the parallel inference machine has much greater significance than has been recognised so far.

The essence of my argument is that the parallel inference machines are not special purpose machines designed to support an esoteric, unconventional logic programming language for a narrow range of AI applications. But rather they are general-purpose parallel machines that support mainstream models of computation. This is because concurrent logic programming, the computational model directly supported by PIM, is as much a model of object-oriented programming and of mainstream approaches to concurrency, such as CSP and CCS, as it is a form of logic programming.

From this point of view, the FGCS work on PIM has almost certainly been the largest, most innovative, and most successful development of parallel computer architecture and its associated software so far. I have little doubt, therefore, that it will have an important influence on the development of parallel computer architectures of the future.

Wegner: The operating system that was described has many mainstream features like process migration. FGCS has considered many technical problems that are not specifically dependent on logic programming that could easily be transferred to other paradigms, and this is very significant.

Kowalski: It is now time to draw this panel to a close. In doing so, I would like to summarise a few conclusions, which I believe are representative of the discussion we have had.

Firstly, I think we all agree that logic programming must be reconciled with object-oriented programming. Perhaps this is another way of saying that the gap between "don't care" and "don't know" forms of logic programming must be closed.

Secondly, we should be aware that in evaluating the future prospects of FGCS technologies it is not only technical merit that matters. Sociological considerations are also important. As Ross Overbeek mentioned in particular, it is not at all certain that the programming language C has achieved its dominant position in computing today for technical reasons alone.

Thirdly, during this week we have seen many impressive results presented and demonstrated by ICOT. It is not possible to evaluate these results completely in such a short time. It may even be that some of the proposals that have been presented, for closing the gap between "don't care" and "don't know" forms of non-determinism for example, might already have solved many of the problems we have been discussing during this panel. More time is necessary to complete the evaluation process.

Finally, on behalf of all of the participants at this Conference, I would like to thank ICOT for having given us a very exciting ten years, for having organised this Conference and presented so many impressive results, and for having introduced us to the future of computing. Thank you.