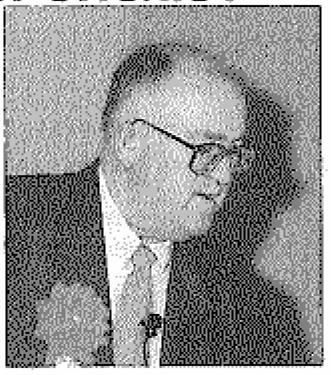
Invited Lecture

"Algorithmic and Knowledge Based Methods: Do They 'Unify'?" —With Some Programme Remarks for UNU/IIST—



Dines Bjørner
Director
The United Nations University's International Institute for Software Technology

Tanaka: Good afternoon. It is the time to start the afternoon session. My name is Hidehiko Tanaka. I am going to serve as the session chair.

This session is for the invited lecture by Professor Dines Bjørner titled, "Algorithmic and Knowledge Based Methods: Do They 'Unify'?". Professor Bjørner had been working at IBM for thirteen years after his graduation from the Technical University of Denmark in 1962. He moved to the Technical University of Denmark in 1976 as the professor of computer science. After holding several managerial positions, including Chairman of the Danish Government Information Commission, and Director of the Dansk Datamatik Center, he is now the Director of International Institute of Software Technology of the United Nations University. He published many papers, and is well known as a distinguished researcher in the area of programming methodology such as the "Vienna Development Method".

Today, we expect that we can hear about his latest consideration from a very wide viewpoint in the programming. Professor Bjørner.

Bjørner: Thank you very much, Professor Tanaka-san.

I hope you can hear my voice. It seems that we have quite a few people in the audience, so, if you think you've been drinking too much Japanese tea for lunch and see three screens, you have not been drinking too much.

I am very greatful to have been invited to come here and address you. The title of the talk was, of course, chosen on the basis of the main topic of this conference and on the basis of my own personal background. Very briefly, there will be three parts to the talk: first, I will talk in general, very much so, about model-oriented or algorithmic software development; then, about the contrast between algorithmic and knowledge based methods trying, to suggest a way in which we may understand the two; and finally, I will talk a little bit about my next job.

When I was invited to give this talk, the organizers, the people here expressed a desire to hear about strategies for the United Nations University International Institute for Software Technology for prompting research and development, creativity in the national cooperation, and so on. And I will certainly cover that in various parts of the talk.

When I acknowledged this invitation, I wrote, and I still stand by that, that the decision by MITI to start ten years ago the Fifth Generation Computer Project, I believe, has had enormous and very positive influence on world wide research, engineering, and application on knowledge based computing systems. Japan has, thus, as an example, made the world of computing, I believe, dramatically more professional and exciting.

What is the background on my giving this talk? Basically, I am myself primarily working in the algorithmic or modeling approach to software development, whereas, the project we are here to hear about is primarily focused on the knowledge based approach. For that reason, I solicited my colleague, Professor Jørgen Fischer Nilsson to help me writing the paper.

So, let me repeat again. First, in general about algorithmic development, I will just very briefly outline some of the issues. For each of the three parts there I will express some dogmas; then, I will go into a comparison between the two approaches: first, I will briefly show a case study; and independent of that, I will systematically "tabularize" the algorithmic versus the knowledge based approaches as I see it. Mind you, this is certainly not a deep, let alone, a scientific tabularization, but it is an attempt; maybe it will inspire an undoubtedly much better such comparisons. And I conclude that part by trying to relate in a more mathematical sense, perhaps, the two approaches. Then, I will sketch what my next job assignment is all about.

In the algorithmic approach, in the modeling approach to software development, the way I see it, basically we persue a very extensive and thorough development of the requirements before we even start thinking about the software. Once we believe that we have captured the requirements, then, and only then, are we ready to start the software development.

The requirements development, to me, consists of describing the problem domain, of modeling various aspects of the domain, and of capturing the aspects which are to go into software. Once that is done, we can progress into the software development, to the specification, the stepwise design, and into excutable code.

A number of facets, a number of steps are involved, not necessarily sequentially, in the requirements modeling. Such things as understanding which are the components, what are the invariants governing the various components of our problem domain, what are the input-output behavior of some of the things that occur, the actions, the events; how are the events traced together or linked together in behaviors; and in total, how does it all hang together. Some of the aspects for which one can write down, more or less independent or not independent, but isolated models that can then be composed and related.

Orthogonal to doing these things, one may focus on safety criticality aspects, if that is required, of the system that one is modeling, of probabilities of failure, of efficiency of the suggested system, and of the computer human interfaces. So, each of the latter can be applied to parts of the former. And again, independently, one can simulate various aspects of these. But, what should be remembered here is that all these activities are concerned with understanding the problem domain and not concerned with understanding the software yet.

The first set of dogmas is, therefore, basically this: that, if we cannot in precise natural language describe what we are about to do, if we cannot write down mathematical models of the various aspects that I listed on the requirement modeling before, and if we cannot capture these requirements that have to go into the software succinctly, then we can have little or no trust in the development. And I say this because, it does mean that I have very little trust in any of the developments that I am myself undertaking; there is still a lot of work to be done here.

After requirements development, we go into software development, which I see as consisting hand in hand of programming and software engineering. In programming, I treat specifications, designs, and code as formal objects. I reason about them; I calculate properties.

And again, a set of dogmas here are that, if we cannont express the functions that we expect our software to exhibit, once implemented, if we cannot beforehand, before costly implementation write down all the behaviors we expect this software to exhibit, and the assumptions on the environment upon which it is built, if from these functional and behavior specifications, we cannont calculate properties before, again, costly implementation, then we can have little or no trust. Here it seems that there is a lot of work that ought to be done and used. But still, of course, there is a lot of research to be done.

Hand in hand with programming, oftentimes the same people, typically the same developers perform acts of what I call software engineering: plan, monitor and control quality assurance. After software has been put into service, monitor and control if the environment still conforms to the written requirements. In software engineering we primarily look upon the design documents, the software, as physical objects, we execute them, we observe them, we compose them.

And again, a set of dogmas are needed, I believe. Whenever we say that a product conforms to its expectations or to the requirements, whenever we configure a new product out of various versions, et cetera, et cetera, we make claims that the software is still fulfilling requirements and specifications. And as long as these claims cannnot be supported by calculated evidence, then we can have little or no trust. And again, there is very little we can do about it today.

Concluding the first part, the generalities part, concluding the three dogmas we can say that in mechanical engineering, they do perform calculations on, for example, ship designs before costly implementation. Electronic engineers perform calculations on, say, data communication designs before costly implementation. Civil engineers perform calculations on, for example, bridge designs. The clients who contract ships, satellite or bridges they do expect their insurance companies to help them in validating these designs, in inspecting these calculations.

In requirements and in software engineering, we are, today, in a better position to perform such calculations, only nobody seems to be doing it.

So, that was the end of the first part where I assumed that the tradition of algorithmic software development as witnessed by the PASCAL school, was the basis.

Now, I would like to go into comparing algorithmic to knowledge based development.

Let me first give an overview example. Imagine that you have a railway computing system (Fig. 1): a system in which you can do planning of train schedules, or in which you can do a realtime regulation of the train traffic, or in which you can do service functions like ticketing and reservations, et cetera.

OVERVIEW EXAMPLE

Railway Computing System: Planning, Regulation, Service & Mgt.

Three approaches:

1. The Compiled Algorithmic Approach
   $\mathcal{C}$ Specifics of railways hidden in code: in constants and code structure; composition of statements.

2. The Interpreted Algorithmic Approach
   $\mathcal{I}$ Tracks, trains, schedule, etc.: constant data structures; computation proceeds through interpretation.
   Interpreter ($\mathcal{I}$) is general: same interpreter can handle wide variety of railway systems.
   Execution is slower, but easy introduction of changes to the schedule, tracks, trains, etc.

Fig. 1

For such a railway computing system --- and it doesn't matter, of course, that it hasn't anything to do with railways --- you can either take a compiler approach in which the specifics of the particular railway, the Shinkansen, for example, and its many stations, are hidden in the code. The fact that there may be a specific number of stations down the line, may be embodied in the composition of a similar number of statements in the code.

In the interpreted approach, we think of the tracks, the trains, the schedule, et cetera, as constant data structures and the computation proceeds through and interpretation: that is, the interpreter is general. For a different, constant data structure, potentially the interpreter can be reused for another railway system.

We can compare the two approaches (Fig. 2) and say that compilation represents a partially evaluated version of interpretation that the compiled code, respectively of the interpreter, does not express any reasoning about train systems. Instead, the compiled code and the interpreter express computations, embody in the case of a train regulation, mathematical laws of kinematics, and in the case of schedule planning, some laws of operations research. But these laws are packed into the compiler code respectively into these specific interpreter.

In the knowledge based approach, we have to think of these laws; whatever governs, the planning, the scheduling, the train control regulation, we have to imagine these laws as being represented in logical form as rules, and what in the interpreted approach amounted to some constant data structure, we have to think of them as being represented as some facts.

And now, the computation here proceeds by means of an inference machine, and basically the same inference machine can be used for a broad range of systems, not just railway systems.

To show an example of what I mean by a base model, I give an algorithmic model-oriented specification (Fig. 3); and with a few additions, maybe one more page, ten, fifteen more lines, I think we can capture anything you would ever want to know about a railway system, something that recall to what time we have, a schedule which says from stations to stations certain trains depart and arrive at certain times, and that there is a graph, a track system, that between stations contain tracks with various segments of individual lengths and that the trains have a location on the track or at stations, and that the trains have certain length, velocity, acceleration or deceleration.

Let that model here be the basis for some first comparisons. (Fig. 4) The first comparisons now try to compare how I would approach basically the same problem in the knowledge based approach. In the model-oriented approach, I might write down the domain equations, whereas, in the classical Prolog type, I would write down some appropriate predicates and there would have to be some integrity constraints like there would have to be some invariants on these which could be expressed in various clausal forms.

In the two approaches one would deal with irregularities in the schedule in different ways. (Fig. 5) If the schedule is irregular, then we would have to give clauses for each pair of stations and trains in departure and arriving times, like we would have in the model-oriented approach. If it is fairly regular, namely, the schedule is sustained everyday, seven days a week, 365 days a year, then, one could invent appropriate predicates which would speak about the intervals with which trains depart, and the time it takes for a train to travel certain distances.

1-2 Compiled and Interpreted Algorithmics
$\mathcal{A}$ Compilation represents a partially evaluated version of interpretation.
Compiled code respectively interpreter does not express reasoning about train system.
Instead they express arithmetic computations, classical mathematical laws of kinematics and operations research.
Laws are "baked" into compiled code, respectively into specific interpreter.

3 The knowledge Based Approach $\mathcal{K}$
Represents laws in logical form, Rules, and represents railway system components, Facts.
"Computation" proceeds, by means of inference machine $\mathcal{M}$. Same $\mathcal{M}$ for broad range of virtually "anything".

Fig. 2

BASE MODEL DOMAINS: $\mathcal{A}$

1.0	*rs:RS*	$= Time \times SCH \times G \times TS$
2.0	*sc:SCH*	$= S_m\,(S_m\,(T_m\,(D_m A)))$
3.0	*g:G*	$= S_m\,(S_m N_1{}^{+})$
4.0	*ts:TS*	$= T_m\,(LOC \times KIN)$
5.0	*lo:LOC*	$= S \mid (S \times N_1 \times S)$
6.0	*k:KIN*	$= L \times V \times AD$
7.0	*ve:V*	$= N_0 \times \underline{s}\text{-}max{:}N_1$
8.0	*ad:AD*	$= INTG \times \underline{s}\text{-}max{:}INTG$
9.0	*Time*	$= Week \times Day \times Hour \times Min$
10.0	*Week*	$= 1 \mid 2 \mid \ldots \mid 52$
11.0	*Day*	$= 0 \mid 1 \mid \ldots \mid 6$
12.0	*Hour*	$= 0 \mid 2 \mid \ldots \mid 23$
13.0	*Min*	$= 0 \mid 1 \mid \ldots \mid 59$
14.0	*D, A*	$= \ldots$
15.0	*s:S*	= TOKEN
16.0	*t:T*	= TOKEN
17.0	*le:L*	$= N_1$

Fig. 3

Representation $\mathcal{A}$	
Railway Schedule &c.	
$\mathcal{A}$	Algorithmic Development Method
1	Domain Equation: $SCH = S \rightharpoonup_m (S \rightharpoonup_m (T \rightharpoonup_m (D \rightharpoonup_m A)))$ De-Curried: $SCH = (S \times S \times T \times D) \rightharpoonup_m A$

versus

Representation $\mathcal{K}$	
$\mathcal{K}$	Knowledge Engineering
1	Schedule given as a 5-ary Predicate: $SCH(S, S, T, D, A)$ with indicated argument types: $S \times S \times T \times D \times A \to \mathrm{BOOL}$ Functional constraints on arrivals could be expressed Horn-clause-wise as an integrity constraint: $error() \leftarrow SCH(S_1, S_2, T, D, A')$ $\& \; SCH(S_1, S_2, T, D, A'')$ $\& \; A' \neq A''$

Fig. 4

Let us make a first comparison before we go to the more general ones.

So, hopefully being objective here, the knowledge based form seems to be neutral and uncommitted with respect to how you look up, you could say, facts; the five arguments to our search predicates are the same kind. You could think of six arguments and input and output is symmetric, which they certainly are not. In the algorithmic form, the way in which I chose to model the domain favored what I might have believed was the most usual application or the most common way of using my data structure.

Other access forms, than finding what the arriving time would be, given stations and trains, and departure times — other access forms would have to be rather tediously specified, and they will not be easy to understand in the algorithmic approach. However, in the algorithmic approach, those are just an efficient elaboration.

REMARKS

If schedule is irregular, then SCH given as collection of factual clauses:

$SCH(s_1, s_2, t, d, a)$.

If schedule is 'fairly' regular *SCH* could be given as clausal form rules of principal form:

```
18.0   SCH (S1, S2, T, D, A) ←
  .1      TRAINS
            (S1, S2, T, FST, ITV, LST, LAG),
  .2      INTERVALS (FST, ITV, LST, D),
  .3      A = LAG + D
```

INTERVALS yields all possible values for D according to a regular schedule.

Factual clauses can be derived from this rule form by means of partial deduction thus expanding the *INTERVALS* predicate.

Fig. 5

If I go down to scheduling, if I'm a customer wishing to perform a certain jurney involving changing trains, and so on, then, I could imagine a more elegant predicate being specified in the knowledge based approach in the Prolog than in the algorithmic approach. And rescheduling could, then, be accomplished by some form of meta-reasoning on the scheduler rules.

Let me generalize this and try to compare the two approaches, and now I am not doing it with respect to the particular railway example that I started out with.

So, I'm basically going to read these transparencies; they are all in the seperate folder for the invited talk. (Table 1) In the algorithmic approach, we specify I/O functions which are then to be implemented in a mathematical sense. So we speak about implementation relations, refinement relations, and so on. In the model-oriented, the algorithmic approach, we use such data structures as SETS, as Cartesian products, as triples, as maps. And we do make a distinction in the algorithmic approach between the specification language and the concrete programming language.

In the knowledge based approach, one by one, we formulate the description and the specification as assertions; we think of computation results understood as proofs corresponding to the logical consequences, and we have a dual view that our specification language is also our programming language.

So, there are some presumptions at the root of what we are doing. (Table 2) In the algorithmic approach we do make the distinction that programs are not the same as specifications, that specifications are refined in stages into executable programs, and that it is the job of the individual programmer and software engineer to do so, and many of us do believe that specifications should not necessarily be executable.

The focus is on implementing data structure specifications, and finally, there is the notion of compilation crucially burried.

In the knowledge based approach, logic is our object language; programs and specifications are often confused or partially identified with each other, whereas, in the algorithmic approach specifications are refined; in terms of Robert Kowalski, we think of executable specifications in the sense of algorithm equals logic plus control.

And here, in the next point I want to make, creeps in some of the problems. In my listing of the algorithmic aspect, I am primarily focusing on the software development. Certainly, I started out by advocating in the algorithmic approach a lot of work on the requirements development. In the requirements development, the focus was on domain knowledge. In knowledge engineering development, the focus remains on the domain knowledge, whereas, the focus in algorithmic development, in the software development, we have the focus on increasingly more efficient implementations. I'll come back to that.

Table 1

Table 1 $\mathcal{A}$	
Formal Software Specification Aspects: Aims	
$\mathcal{A}$	Algorithmic Development Method
1	Specification of I/O function in mathematical sense to be implemented.
2	Use of such data structures as sets, Cartesian products, tuples, maps.
3	Distinction between abstract specification and concrete programming language.

versus

Table 1 $\mathcal{K}$	
$\mathcal{K}$	Knowledge Engineering
1	Description and specification of real world relationships as assertions
2	Computation results understood as proofs corresponding to logical consequences of $\mathcal{K}$.
3	Dual view of logic as both specification and programming language.

Table 2

Table 2 $\mathcal{A}$	
Formal Specification Aspects/Presumptions	
$\mathcal{A}$	Algorithmic Development Method
1	Program ≠ Specification
2	Specification refined in stages into Executable Programs, Specifications not necessarily Executable
3	Focus on implementing data structure specification, for example Stacks, Queues, etc.
4	Finally: Compilation

versus

Table 2 $\mathcal{K}$	
$\mathcal{K}$	Knowledge Engineering
1	Logic as Object language Program and Specification often confused or partially identified
2	Specification of Executable Specifications = Declarative Programming cf. Kowalski: Algorithms = Logic + Control
3	Focus on domain knowledge
4	Finally, efficiency through general methods such as: Constraint Satisfaction and Intelligent Backtracking

So, compilation here compares then in knowledge engineering approach to constraint satisfaction and to intelligent back tracking.

This table attempts to focus on some of the mathematical forms of computation. (Table 3) In the algorithmic approach, we have a concept of computation as that of functions, as that of stepwise refinement by means of imperative programming language constructs, which in a sense leads to a notion of deterministic compilation. Behind it all, there is the notion of the calculus making us understand what is going on.

Table 3

Table 3 $\mathcal{A}$	
Mathematical Form of Computed Object	
$\mathcal{A}$	Algorithmic Development Method
1	Functional Conception of Computation Stepwise Refined and eventually algorithmitised by means of imperative programming language constructs, Operations on Data Objects $\Longrightarrow$ Deterministic Compilation λ-Calculus reduction as Computational Basis
3	Specifications are often 'functional' (algebraic)

versus

Table 3 $\mathcal{K}$	
$\mathcal{K}$	Knowledge Engineering
1	Relational Conception of Computation $\Longrightarrow$ Links with Relational Database, Non-determinism ("Backtracking") (Quasi) Parallelism $\Longrightarrow$ Proof Rule Deduction (Resolution) Unification as Computational Basis
2	Derivation of answers from Assertions ($\models$, $\vdash$) $\Longrightarrow$ Deduction Abduction (Cause Analysis) Induction (Machine Learning)
3	Specifications often 'relational' structures stressing ($m : n$) relationships.

In comparison, in the knowledge engineering approach, the knowledge based approach, we have a relational concept rather than a functional concept of computation. And this is why we see the links to relational databases. In contrast, because of the relational aspects, it is quite natural to think in terms of non-determinism. We then have unification as a computational basis.

In addition, we have, perhaps a more interesting variety of computational objecs: deduction, abduction, induction.

I touched upon the problem before. I will repeat it again in larger context. In the algorithmic approach, the people who developed the specifications are usually the people who also understand the problem domain, and they are also the people who will then be asked to provide an efficient implementation.

Where efficiency is not any more expected to result from compilation itself, the algorithmic developer is charged with securing efficiency in his or her stepwise development. (Table 4) So, from the specifications we derive, in many stages, the programs, usually informally, or usually not automatically. We have such notions as developing iterative loops, or iterative formulations from recursive ones. A paradigmatic example is that of compiler development from formal specifications of the static and dynamic semantics.

In contrast to this, we have that the knowledge based specifications are ideally executable, and we achieve efficiency through cleverly devised inference machine which features partial reduction, constraints satisfaction, and especially unification.

Initially we have an uncommitted choice between top-down use of rules versus bottom-up use of rules. The paradigmatic systems, which for the algorithmic one was the compiler example, here typically is that of deductive databases.

The two notions of types interfere --- in the notion of types, as we see it in standard ML (Table 5), one is concerned with avoiding paradoxes --- that notion of type basically has its root in Bertrand Russels' type theory. The notion of types

Table 4

Table 4 $\mathcal{A}$	
Towards Computational Efficiency	
$\mathcal{A}$	Algorithmic Development Method
1	Stepwise Development (Refinement) Specification $\rightsquigarrow$ ... $\rightsquigarrow$ Programs Informally or (semi) automatically Stressing Data Abstraction and Applicative Forms, Operations on sets (an example) become operations on lists.
2	Development of Iterative Formalisms from Recursive ones
3 4	Paradigmatic Systems, Examples: Compiler Development from Formal Specification of Static & Dynamic Semantics

versus

Table 4 $\mathcal{K}$	
$\mathcal{K}$	Knowledge Engineering
1	Specifications are ideally executable Efficiency eg. by Partial Deduction Constraint Satisfaction Methods Special Unification Methods
2	Initially uncommitted choice between Top-Down (Backward) use of Rules (cf. Recursive Forms) versus Bottom-Up (Forward) use of Rules (cf. Iterative Forms) Commitment through Meta-interpretation choice
3 4	Paradigmatic Systems, Examples: Deductive Databases = subset of Prolog's logic in which true declarativity and termination (decidability) is achievable.

that we have in the knowledge based engineering is more Aristotelian and is viewed as a classification of entities. Of course, the two overlap, but they have a different background.

In the algorithmic, we use types as a means of protection, as a compile time thing, as a choice of representation, whereas, here, in the knowledge engineering approach, we deal with various logics, terminological logics, syllogistic forms, all the sorts of logics, and so on, as generalizations of classification hierarchies.

In the algorithmic development, units of specifications are possibly non-determinate on the specfifed functions versus rules and facts in the knowledge based approach. The composition principle in the algorithmic approach is that a functional composition versus that of the conjunction.

This leads us into another way of comparing these two approaches, the various structural mechanisms. (Table 6) First recall my finely grained stepwise approach to requirement model-

Table 5

Table 5: Surprisingly few logic programming systems offer recursively defined, compound or structured types

Table 5 $\mathcal{A}$	
(Data) Types	
$\mathcal{A}$	Algorithmic Development Method
1	Traditionally Types as Protection and choice of representation, (for example: integer and floating point)
2	Types as Structuring Mechanism: Polymorphism Abstract Data Types

versus

Table 5 $\mathcal{K}$	
$\mathcal{K}$	Knowledge Engineering
1	Types viewed as classification of 'real world' entities $\Longrightarrow$ Terminological Logics (Concept Logics) Syllogistic Forms Order-sorted Logics as Generalisation of Hierarchies

ing, the facilities, the functions, the behavior, the safety, the performance, and so on; they helped give me some structuring, but with our present understanding it could not be mathematically explained.

In the knowledge approach, we may think of such things as the schemes in the database world, as the inheritance classification hierarchies. In the algorithmic approach there's a lot of work, there's a lot of examples of structuring mechanisms in the sense of block structures, modules, and procedures, whereas, in the knowledge engineering approach, there seems to be much less structuring available.

Exceptions play an important role. Coming back to our examples of the railway, the schedule assumed some modularity; so no exceptions were made there. If I do not have some kind of given interval over which the schedule repeats itself, then I may have to fix this in the algorithmic approach by extending the schedule with all the exceptions. Or, I could repair that by just enlarging on the interval by detailing the data structure. (Fig. 6)

In the knowledge based approach I might avail myself of various features from non-monotonic logics by overwriting the irregular schedule with more specific singularities in order to avoid exploding the rules. So, in the algorithmic development, we handle exceptions by explicit enumeration, whereas, in the knowledge based approach, there is, of course, that fundamental distinction between being able to derive not A, and not being able to derive A. (Table 7)

Now comes the end of the second part of my presentation, I go back to the initial example of the first part, namely the railway example. I spoke about an inference machine which would reason over rules and facts. I spoke about an interpreter which would work on a constant data structure. And I spoke about a program that could be com-

Table 6

Table 6 $\mathcal{A}$	
Descriptional Structuring Mechanisms	
$\mathcal{A}$	Algorithmic Development Method
1	Finely grained requirements models Base, Function, Behaviour; Environment & Interface, Safety Criticality, Dependability, CHI
2	Block Structures and Modules with Encapsulation Procedures

versus

Table 6 $\mathcal{K}$	
$\mathcal{K}$	Knowledge Engineering
1	Database Conceptual Models and Schemes Inheritance Classification Hierarchies The Rule Clauses as 'self-contained' Units of Specification
2	(Definite) Horn Clauses

COPING WITH EXCEPTIONS:

- Schedule assumed some modularity, say, a weekly plan: working days & week-ends. No exceptions were made.
- $\mathcal{A}$: "fix" this either by 'extending' the schedule: a composition of a regular schedule, as first shown, with an exception schedule:

19.0	$SCH = REG \times EXC$	Schedule
20.0	$REG = S_m (S_m (T_m (D_m A)))$	Regular
21.0	$EXC = Time_m (T_m Attr)$	Exception
22.0	$Attr = \ldots$	Exception Attributes

- Or "repair" original schedule: complete orginal schedule to always spans a full year:

23.0 $SCH = Time_m (S_m (S_m (T_m (D_m A))))$

- Non-monotonic logic: modify the regular schedule by "overwriting" with the more specific singularities,
- thus avoiding proliferation of exceptions into the regular schemes.

Fig. 6

Table 7

Table 7 $\mathcal{A}$	
Domain Model Exceptions and Non-monotonicity	
$\mathcal{A}$	Algorithmic Development Method
2	Exceptions handled by Explicit Enumeration of Cases

versus

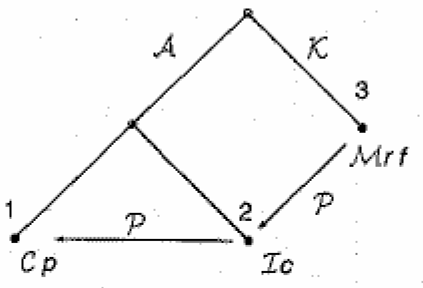
Table 7 $\mathcal{K}$	
$\mathcal{K}$	Knowledge Engineering
1	"Negation as (finite) Failure" (SLDNF-Resolution)
2	Distinction between $\vdash \neg A$ and $\not\vdash A$.

piled, all for the same system. And basically, one could expect some kind of partial evaluator, which applied to the inference machine the rules and the facts, basically derived the pair of the interpreter and the constant data structure.

Similarly one can imagine a partial evaluator, and hopefully the same which could be applied to the interpreter and the constant data structure, which would yield my program. (Fig. 7) So, such a partial evaluator should satisfy certain laws. These laws are as follows:

So, the meaning of inference machine applied to the rules and the facts should give me the same as the meaning of the interpreter applied to the constant data structure and should give the same as the meaning on my program. And, of course, I could indeed write down the inference machine, the interpreter, and the program in the same language, L. And that, of course, should be the same as the meaning of the compiler which may be written in some other language applied to my program, and the meaning of that program in some machine language. That's one aspect that must be satisfied, of course; and here comes the rules that must be satisfied by the partial evaluator. So, the first part is the meaning of the partial evaluator applied to the inference machine and the rules which should basically give me the interpreter. Some kind of isomorphic representational identity should be satisfied between the facts on my --- in this case --- a railway system in the constant data structure. If they were written all in the same language, then I would have an equal sign. And similarly the partial evaluator applied the meaning of that applied to the interpreter should give me the same.

$\mathcal{C}$ - $\mathcal{I}$ - $\mathcal{K}$ APPROACHES



$\mathcal{P}$: Partial Evaluator
Functional must satisfy the laws:

- $[\![\mathcal{M}]\!]_L\, r f = [\![\mathcal{I}]\!]_L\, c = [\![p]\!]_L = [\![[\![\mathcal{C}]\!]_{L'}\, p]\!]_M$
- $([\![\mathcal{P}]\!]_L\, \mathcal{M} r, f) \approx (\mathcal{I}, c)$
- $[\![\mathcal{P}]\!]_L\, \mathcal{I} d \approx p$

LEGEND:
- p : railway system program (to be compiled)
- c : railway system constant data structure (to be interpreted)
- f : knowledge base facts which reflect the railway system components
- r : knowledge base rules which reflect laws of railway systems
- $\mathcal{C}$: compiler
- $\mathcal{I}$: interpreter
- $\mathcal{M}$: inference machine

FUTAMURA PROJECTIONS:

$$[\![[\![\text{mix}\ P\ S]\!]\ D]\!] = [\![P]\!]\, S\ D$$
$$T = [\![\text{mix}]\!]\ I\ P$$
$$C = [\![\text{mix}]\!]\ \text{mix}\ I$$
$$CG = [\![\text{mix}]\!]\ \text{mix}\ \text{mix}$$

Fig. 7

Now, that's all very nice, and looks very pretty, but the question is, how do we construct that partial evaluator. And that is, of course, a good question; and if I knew the answer to that question, I probably wouldn't be standing here, and that's why I'm presenting it for you to help me to find out. There is a growing experience in building such partial evaluators. They must satisfy various laws.

The most classical one of these are shown here at the bottom. So, MIX is a partial evaluator, and the idea is that MIX applied to the meaning of the program and its static data, the meaning of that applied to the dynamic data, the meaning of that should be equal to the meaning of the program in some other language, perhaps, applied to the static data and the dynamic data.

Given an interpreter and a source program applying the meaning of the partial evaluator to that, I get the target program. Given that, I can now write down the rules, the laws that MIX must satisfy. So, the meaning of MIX applied to MIX and interpreter gives me a compiler, and the meaning of MIX applied to MIX, and MIX gives me a compiler generator. Now, this is a twenty-year old knowledge that I'm presenting you with; perhaps it is more elegantly written down by Professor Neil Jones, but certainly you should know about it very well; these are known as the Futamura projections, these last three equations.

So, the question is, we would like to know how to construct this Partial Evaluator, P, and the point of me bringing this here after my many tabular comparisons is that, in a sense --- to be made precise, of course --- in a sense, that P must embody --- certainly the first P here must embody exactly the meta --- I shouldn't use the word 'knowledge', but --- it should embody the facts, or the things we know about the relationships between the algorithmic approach and the knowledge based approach. So, the various comparisons that were made in the tables somehow express that partial evaluator; and that is my point.

This brings me to the end of the second part, the more technical part; and I should now finish by telling you a little bit about what the United Nations University is and the fact that they have decided to create a research and training center to be located in Macao and to be started on June 8th, next Monday. So, the UNU/IIST is endowed by 30 million dollars from the governments of Portugal and China, and the governor of Macao. Its activities will be oriented towards the developing world. It seems rather ambitious in its spread; it shall help, it shall assist in the software usage, in software technology management, in development of software, in curriculum development, and in research. It hopefully will be able to do so, and it will operationally do so through a combination of projects, courses, research, consultancy, and dissemination.

We plan that the UNU/IIST should engage in three kinds of projects: small explorative projects which will apply research ideas to small but difficult subsets in order to investigate feasibility of new approaches and new applications. Such projects may lead to demonstrator projects and they may lead to other research. Demonstrator projects apply what we hope to be scalable state-of-the-art techniques to applications, and they serve as the basis for education courses. And finally, they may lead to technology transfer projects. The idea is that the UNU/IIST should not itself develop software but should mobilize forces in the developing world so that eventually the projects will transfer to new or previously identified groups in the developing world.

Training is achieved through a combination of projects and courses. We work with three kinds of courses: training, awareness and education courses. Training in the application, software installation, operation and use. Awareness courses focusing on software technology management, and education courses which, in the sense of this talk, very much focus on the things we've been talking about, namely, three-month specialist courses for post-graduates both from universities and from industries.

Research will be done, but it will not be a major feature of the institute. We would certainly hope to be able to bring the results of the industrial world to bear, and vice versa there are many fascinating results and ideas in, say, China and India that ought to be more widely known in the industrial world. Initially, our own research will be conducted by our own staff, by visiting experts, and research scholarship students. And almost always in joint collaboration with other institutes and industries in the developing world. Initially we will focus on the algorithmic approach to software development, and we will initially focus primarily on a continuous time interval temporal logic called the duration calculus.

There will be created some kind of organic network that will tie university groups, software houses, and other centers together.

We will operate through offering scholarships to post-graduates from universities and industries to participate in Macao in projects and courses, short courses or seminars or long intensive residential courses, and also to invite six to nine-month research scholarships or research scholars to come to Macao.

So, basically the UNU/IIST will have many features; we hope to be able to do consultancy work for the UN system; we certainly are planning a number of workshops and panels already this fall; we'll do some research; we'll have various kinds of courses, and we'll have various kinds of projects. And in this we will interface with different segments of institutions in the developing world.

Now, I come to the conclusion. I tried to be faithful to what I was expected to say in that I have mentioned some of the UNU/IIST strategies for prompting research and development. In my talk on the algorithmic parts and the algorithmic and knowledge based parts, you have seen some of the attitudes that lie behind our research and development. I have made a plea for calculations at all levels. I have briefly mentioned some of this. In the invitation there was also asked that I address the issue of creativity and future prospects, and in the interest of time, you can read that in the submitted paper.

So, I would like to acknowledge and thank my chairman, Professor Hidehiko Tanaka, and Koich Furukawa-san for inviting me; I've had great help from Kanbayashi-san in preparing these. And also thanks to my co-author. So, with this, ladies and gentlemen, thank you very much for your attention.

Tanaka: Thank you very much, Professor Bjørner.

As we have a few more minutes for this session, so, I would like to have some discussions using these minutes. Is there any questions? Use the microphone, please.

Wolfgang Bibel: Professor Bjørner, I must admit that I have a very different view of the comparison of the algorithmic and the knowledge part. So, basically my first question will be, could it be that you take a too narrow view of the knowledge engineering part. As I would figure, you view knowledge engineering simply as prolog programming, and that seems to me to be a very narrow view; bacause, just think of extraction of programs, real programs from proofs, or, think of integrating abstract data types with the knowledge specification of a given problem, and many other things like that. Even compilation into a real efficient program, that is the target of many researcher in program synthesis. So, may be it is believed that all these things are not so well and widely known as they should be, but there is, I feel, a distorted view if you just put these two things, algorithms on the one side and knowledge engineering on the other side. That is my first question. Couldn't it be that it is too narrow?

The other question is, that knowledge engineering has a wide perspective, and the perspective is that we can do modification of programs. In the sense, you just modify the knowledge specification and then take the consequence into the synthesis part. My question is, is there anything comparable on the algorithmic part, or, isn't it true that an algorithmic programmer has to re-do all these things from scratch.

Bjørner: Okay. So, you did not ask two questions; you asked three questions at least. First of all, I was almost saying 'yes' to you when you had the first part of your first question. Yes, I do take a narrow view of knowledge engineering, namely, I take that view in which everything can be written down precisely in mathematics, in closed form. But, to the second part of your first question, no. I certainly felt that the aspect you mentioned with extracting programs from proofs and so on is indeed a part of my comparison. I think you'll find that in the inductive programming, abductive programming, and soon. The point here is that the developer, whom I asked to develop a compiler or a database system, that person is charged with implementing all the efficiencies, whereas, the problem domain modeler who comes out with a succinct logical form of the problem domain in the way that you would do it. That person is not charged with any efficiency of use. Such is asked for by some other people who invent clever algorithms for resolution and for unification and many other things for extracting the programs from the proofs and so on.

In your second question. I intend to agree with you there, but only historically. If you can develop a very precise requirements model for banking. And if any new software and addition to existing software follows exactly that, then you have a homo-morphism so that any changes are very easily introduced. But still, you have to algorithmitize. But you don't have to go back and change any of the old things. But that's a more technical argument.

E.A.Feigenbaum: Could you give us some historical background on how the International Institute of Software Technology came to be located in such an extraordinarily strange interesting place like Macao?

Bjørner: They had the money. So, there was a commission in 1985, and certain people traveled around to Singapore, to Hong Kong, to Macao, to South Korea: these four places. And the Macao people, the government of Macao immediately came up with the promise for the money. If you think Hong Kong is more exciting than Macao, you will also think that Macao has certain wonderful tranquil Portugese features that Hong Kong does not have.

So, basically, a need was identified for having such an institute, and then, people went around asking where the money was, and the Macao people came up with 30 million U.S. dollars. It should be put in the following context: the United Nations University has other institutes: they have one for world development economy in Helsinki, Finland; they have one for new technology: social consequences of new technologies in Maastricht in Holland; and they have other institutes predominant in the developing world. But all of them charged with issues that have to do with the developing world. All of them were in the industrial world, but charged with issues in the developing world. So, certainly this institute ought to be in the developing world, and it is.

I personally find Macao an exciting place. I'm not talking about the prostitution or anything like that or gambling, but it is a fine place. I invite you to come and visit us.