LOAN DOCUMENT

PHOTOGRAPH THIS SHEET

LEVEL

INVENTORY: 0

AD-A280 017

DTIC ACCESSION NUMBER

WL-TR-94-4051

DOCUMENT IDENTIFICATION

Nov 93

DISTRIBUTION STATEMENT A
Approved for public release
Distribution Unlimited

DISTRIBUTION STATEMENT

HANDLE WITH CARE

ACCESSION FOR

NTIS	GRA&I	✓
DTIC	TRAC	☐
UNANNOUNCED		☐
JUSTIFICATION		

BY

DISTRIBUTION/

AVAILABILITY CODES

DISTRIBUTION	AVAILABILITY AND/OR SPECIAL	
A-1		

DISTRIBUTION STAMP

DTIC
S ELECTE D
JUN 7 1994
C

DATE ACCESSIONED

DATE RETURNED

94 5 24 048

DATE RECEIVED IN DTIC

94-15603

27Pg

REGISTERED OR CERTIFIED NUMBER

PHOTOGRAPH THIS SHEET AND RETURN TO DTIC-FDAC

DTIC FORM 70A JUN 90 DOCUMENT PROCESSING SHEET PREVIOUS EDITIONS MAY BE USED UNTIL STOCK IS EXHAUSTED.

LOAN DOCUMENT

DTIC QUALITY INSPECTED 1

WL-TR-94-4051

EPISTEMIC PLANNING FOR MANAGEMENT AND MANUFACTURING

DR FRANK M. BROWN

ARTIFICIAL INTELLIGENCE RESEARCH, INC
PO BOX 459
ORANGE TEXAS 77630

NOVEMBER 1993

FINAL REPORT FOR 06/05/91-08/05/93

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION IS UNLIMITED.

MATERIALS DIRECTORATE
WRIGHT LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT PATTERSON AFB OH 45433-7734

NOTICE

When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely Government-related procurement, the United States Government incurs no responsibility or any obligation whatsoever. The fact that the government may have formulated or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication, or otherwise in any manner construed, as licensing the holder, or any other person or corporation; or as conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

This report is releasable to the National Technical Information Service (NTIS). At NITS, it will be available to the general public, including foreign nations.

This technical report has been reviewed and is approved for publication.

[illegible]
ELIZABETH F. STARK
Project Engineer,
Integration and Operations Division
Materials Directorate

[illegible]
STEVEN R. LECLAIR
Chief, Manufacturing Research Branch
Integration and Operations Division
Materials Directorate

John R Williamson
JOHN R. WILLIAMSON, Chief
Integration and Operations Division
Materials Directorate

If your address has changed, if you wish to be removed from our mailing list, or if the addressee is no longer employed by your organization please notify WL/MLIM Bldg 653, 2977 P Street Ste 13, WPAFB, OH 45433-7746 to help us maintain a current mailing list.

Copies of this report should not be returned unless return is required by security considerations, contractual obligations or notice on a specific document.

REPORT DOCUMENTATION PAGE

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE NOV 1993	FINAL	06/05/91--08/05/93

4. TITLE AND SUBTITLE EPISTEMIC PLANNING FOR MANAGEMENT AND MANUFACTURING

C F33615-91-C-5703
PE 65502
PR 3005
TA 05
WU 00

6. AUTHOR(S) DR FRANK M. BROWN

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
ARTIFICIAL INTELLIGENCE RESEARCH, INC
PO BOX 459
ORANGE TEXAS 77630

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
MATERIALS DIRECTORATE
WRIGHT LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT PATTERSON AFB OH 45433-7734

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
WL-TR-94-4051

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION IS UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This project developed multiple automated deduction systems based on a general knowledge-based reasoning technology which reduces the cost of decision making in large management and manufacturing systems. The project was carried out by first developing a general reasoning tool called Logistica which allows a user to easily implement sophisticated deductive reasoning systems and then by implementing specific rule systems in Logistica dealing with management and manufacturing problems. Logistica is a unique very-high-level reasoning tool based on pattern-directed invocation, multiple execution threads, and user-definable control structures. During the course of this project, Logistica was applied and used by two companies. First it was used as part of Ontek Corporation's work on PACIS for the Air Force Manufacturing Technology Directorate's Integrated Toolkit and Methods Corporate Data Integration Tools project (ITKM CDIT) to provide the key underlying capability of pattern-directed invocation. Logistica was also deployed into a manufacturing system being developed by Allied Signal Aerospace at the Department of Energy's Kansas City plant for translating product descriptions between different product description languages and for reasoning about actions.

14. SUBJECT TERMS ARTIFICIAL INTELLIGENCE, LOGISTICA, AUTOMATED DEDUCTION, Z MODAL LOGIC, NONMONOTONIC REASONING, FRAME PROBLEM, PLANNING

15. NUMBER OF PAGES 25

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev 2-89)

Table of Contents

0. Executive Summary

The application of computational techniques to manufacturing and to the management of manufacturing enterprises is both broad and diverse. The effectiveness of automating physical manufacturing processes sparked optimism that automating the less tangible management processes such as planning, scheduling and control could also be effective. The hope was and is to use computational techniques to reduce the cost of managing complex activities and to increase responsiveness to both scheduled and non-scheduled events. This latter type of automation, though, has proven much more difficult since it deals with the complexities of human mental activity as opposed to the operation of physical machines. Part of the difficulty relates to the computer languages which are used to represent management situations as well as the inferences which must be made regarding those situations. Formal logics of different kinds are usually the basis for computer languages which can handle inference. However, most of these languages deal with very constrained types of inference which are inadequate to represent the breadth of reasoning needed to automate decisions found in manufacturing environments. The goal of this Phase II SBIR has been to develop a computer language, Logistica, for building reasoning systems capable of modeling more complex situations than is currently the case.

Logistica is a computer language which is used as a tool for building inference systems. It is not based on a particular kind of logic, but allows different kinds of logics to be defined and used as the basis for reasoning systems. The value of this fact is that the language is very flexible and may be applied to many different situations. Currently, Logistica is being applied in two very diverse contexts. First, Logistica is being used as part of Ontek Corporation's work on PACIS (a Platform for the Automated Construction of Intelligent Systems). That work is being carried out under the Air Force Manufacturing Technology (ManTech) Directorate's Integrated Toolkit and Methods contract, Corporate Data Integration Tools project (ITKM CDIT). Logistica is also deployed into a manufacturing system being developed by Allied Signal Aerospace at the Department of Energy's Kansas City plant. In that project, Logistica is being used as a tool for reasoning about manufacturing actions, feature constraints on product designs, and as a tool for translating product descriptions between different product description languages. Both these applications provide real world problems which test Logistica's inference capabilities. Also, each of the application projects are deriving real benefit from Logistica. In the case of PACIS, one the fundamental technical requirements supporting several higher level requirements is met by Logistica. The PACIS project intends to incorporate Logistica constructs and formalisms into its system platform. At Allied Signal Logistica was originally purchased to build reasoning systems that would facilitate understanding and representing the results of mechanical manufacturing actions and mental actions involved in the design process. Allied Signal subsequently discovered that Logistica was also a very useful language for implementing translation processes, and for verifying design constraints and consistency. In fact, they have found Logistica to be so useful that one development group is considering rewriting substantial parts of their existing manufacturing system in the Logistica language.

This Phase II SBIR represents advanced theoretical work in computational, logic-based inference, which combined with the two implementations mentioned above, expand Logistica's scope beyond academic interest to include solutions to previously intractable real-world manufacturing problems.

1. Introduction

The overall, long-term objective of the work described in this technical report is to develop and implement a general, self-extensible, knowledge-based reasoning technology specifically targeted for use in comprehensive manufacturing and management operations systems. This technology provides manufacturing and management operations systems with an automated reasoning mechanism for inferring the potential consequence of actions, and for planning sequences of manufacturing and management operations to achieve specific organizational goals. This technology substantially increases the reasoning ability of current manufacturing and management operations systems, thus allowing the simpler decision-making tasks to be automated. This, in turn, leads to decreases in overall indirect costs to design and manufacturing products.

The specific objective of this SBIR Phase II project was to develop multiple automated deduction systems based on a general knowledge-based reasoning technology and suitable for use in large management and manufacturing systems. The motivations for this project are described in Section 2. of this report. The objectives of the project were carried out by first developing a general reasoning tool called Logistica which allows a user to easily implement sophisticated deductive reasoning systems, and then by implementing specific rule systems in Logistica dealing with management and manufacturing problems. Logistica is a unique very-high-level (VHL) reasoning tool based on pattern-directed invocation, multiple execution threads and user-definable control structures. It is described in Section 3. of this report. Our approach to developing complex reasoning system by abstracting out commonly used features and embedding them into the Logistica language is discussed in Section 4., along with some of the lessons which were learned during the development and use of this system.

During the course of this project, Logistica was applied and used by two companies. First it was used as part of Ontek Corporation's work on PACIS (a Platform for the Automated Construction of Intelligent Systems). That work is being carried out under the Air Force Manufacturing Technology (ManTech) Directorate's Integrated Toolkit and Methods contract, Corporate Data Integration Tools project (ITKM CDIT). In that project, the Logistica technology is providing a key underlying capability — pattern-directed invocation — which is being used to help meet several important technical requirements. Logistica was also deployed into a manufacturing system being developed by Allied Signal Aerospace at the Department of Energy's Kansas City plant. In that project, Logistica was used as a tool for reasoning about manufacturing actions, feature constraints on project designs, and as a tool for translating product descriptions between different product description languages. This deployment took the form of a sale to Allied Signal Aerospace of Logistica software, as well as a set of Logistica rules which implement nonmonotonic reasoning. These applications and deployments of Logistica technology are discussed in Section 5. of this report. Finally, some conclusions are discussed in Section 6.

2. Project Motivations

A great deal of effort is currently being expended on developing large manufacturing and management control systems in order to improve productivity in the industrial base. These efforts are intended to model many aspects of the design and manufacture of products, including the representation of those processes, and the use of the representation for management of those processes. While the representation of such knowledge, as well as the storage and retrieval of such knowledge, are themselves significant achievements, such systems would be significantly enhanced by the ability to automatically determine the important consequences which might follow from analyzing data reported to the system in the context of the base knowledge represented in the system. One very important class of consequences which we would like to be able to determine, are those which follow when an action or event occurs. For example, when the manufacturing system is told that a red-colored work-in-process part was moved from one shop floor location to another, it probably should infer that the part is at the new location. For obvious consequences, this is accomplished by storing a rule or law describing what happens when something is moved. Unfortunately, many actions, such as those in a complex environment like a manufacturing enterprise, generally have many unobvious consequences. For example, if a red light had been shining at the first location, but a green light were shining at the new location, the property that the part appears red-colored might also have changed as a result of the move. For this reason we cannot, as is often done in current planning systems, just make an assumption that everything except the obvious results remain the same. We need to be more precise, i.e., we need to need to be able to say something like 'everything remains the same unless it contradicts the direct results of the action or certain qualitative physical laws (such as an object may only reflect the color of those sources of light for which there is such a source at the given location)". This problem, of specifically describing what does and does not change is known as the 'Frame Problem' in artificial intelligence, and is the subject of a growing body of research [Brown87; Ford & Hayes90]. Theories and logics for deducing such unobvious consequences are generally called nonmonotonic reasoning systems.

The basic scientific problem addressed under this SBIR was, therefore, to predict the unobvious consequences of actions involving manufacturing operations and their management. Our opportunity to significantly address this problem lies in the fact that we have developed a scientific theory of such nonmonotonic reasoning mechanisms that explains the reasoning carried out in almost all of these kinds of problems. The theory has been shown to encompass and generalize almost all other competing theories of such reasoning in the literature [Brown86a,b,c,d,87,90; Brown & Park87,88]. Thus, in order to actually build computer programs which carry out such reasoning, there was a need to find a way to develop a technology which would allow a user to quickly implement sophisticated reasoning process. This would entail experimentally developing a computer model of the reasoning process, based on our scientific theory of such reasoning. In the course of this project we developed just such an implementation technology, which we call Logistica. Logistica also draws on our previous research on automatic deduction systems. Logistica was then used to develop a comprehensive rule system for nonmonotonic reasoning about actions, called NONMON. The Logistica system and NONMON are described in Section 3.

3. Logistica: an Expert System for Building Expert Systems

Logistica is a lexically-scoped functional programming language used for implementing deductive reasoning processes. Although at the simplest level Logistica looks like the Scheme dialect [Abelson85; Rees86] of LISP [McCarthy60; Church41; and Steele90] it differs in that symbols may have any number of definitions instead of always just one as in Scheme. The reason for this difference is that there are usually many axiomatic properties of any symbol and thus to represent them in a useful modular fashion a separate definition is needed to represent each property. For example, one definition of conjunction might represent the associative law of conjunction, and another definition might represent the idempotent law of conjunction. In the simple case where no symbol is defined more than once, Logistica is essentially Scheme. For this reason, Logistica can be viewed as one possible explication of the thesis that deduction and computation are similar activities [Hayes73; Kowalski79]. Allowing multiple definitions of symbols is reminiscent of Prolog's [Colmerauer73; Kowalski74] use of several clauses beginning with a particular predicate to axiomatize that predicate. The difference is that Logistica allows multiple definitions of all its symbols, including predicates, functions, and logical connectives. The consequence of this rather slight conceptual change is quite significant, in that expressions may have multiple values. Control facilities similar to the cut operation of Prolog are included for dealing with multiple values. Because Logistica enables the specification of arbitrary deductive systems (by representing the properties of symbols as 'backtrackable rewrite rules' which are associated to the symbols themselves), it is independent of any specific logic and it provides instead the basic tools for deduction [Araya90a and 90b; Brown86a, 86b, 87, 89, 91a, 91b, and 92; Hundal91; and Park88].

A second important way in which Logistica differs from the Scheme dialect of LISP is that it allows symbolic expressions to be the results of evaluation. The reason for this difference is that symbolic expressions involving variables, such as those occurring in the scope of an existential or universal quantifier, generally do not evaluate to specific values, but are essentially place holders which may be manipulated by logical inference operations.

These differences while allowing new and powerful programming metaphors in themselves, are integrated with the language Scheme, a popular LISP dialect. This integration gives the programmer a seamless and natural progression of tools, from the straightforward functional programming approach of Scheme or LISP to the powerful and highly descriptive expression of alternative symbolic computations afforded by Logistica. These two major differences from LISP are discussed in more detail below, in sections 3.1 and 3.2, respectively. Some example programs exhibiting the utility of these additional capabilities found in Logistica are also described. Section 3.3 rounds out this introduction to Logistica with a brief presentation of its other important features.

3.1 The Multi-Valued Nature of Logistica

The first difference is that symbols may have any number of definitions instead of always just one as in Scheme. The reason for this difference is that there are usually many axiomatic properties of any symbol and thus to represent them in a useful modular fashion a separate definition is needed to represent each property. For example, one definition of conjunction might represent the associative law of conjunction, and another definition might represent the idempotent law of conjunction. In the simple case where no symbol is defined more than once, Logistica is essentially the same as Scheme. For this reason, Logistica can be viewed as one possible explication of the thesis that deduction and computation are similar activities. Allowing multiple definitions of symbols is reminiscent of Prolog's use of several clauses beginning with a particular predicate to axiomatize that predicate. The difference is that Logistica allows multiple definitions of all its symbols, including predicates, functions, and logical connectives. Logistica allows not only single-valued expressions, but also multi-valued and no-valued expressions. Single-valued expressions evaluate to a single value. For example:

```
(* 5 8)                                          ⇒ 40
```

No-valued expressions evaluate to no values. Such a result is indicated in the meta language as failure. For example:

```
#fail                                            ⇒ failure
```

Multi-valued expressions are created by defining additional values. They evaluate to more than one value. For example:

```
(let((x 1))(define x 2)(define x 3)x)            ⇒ 3
                                                 ⇒ 2
                                                 ⇒ 1
```

Multiple definitions are specified in a combination of two ways, either by using multiple define statements (as shown above) or by pattern matching, in more than one way, the formals of a closure being applied, as shown below:

```
(let(((list ...a x ...b) '(1 2 3))x)                      ⇒ 1
                                                           ⇒ 2
                                                           ⇒ 3
```

In this last example, ...a and ...b are segment variables which match zero or more expressions in the argument '(1 2 3), or rather the list, (list 1 2 3). Once multiple values are produced, computation may proceed on the individual values in a generic manner by merely applying operations to the entire expression:

```
(+ 10(let(((list ...a x ...b) '(1 2 3))x))                ⇒ 11
                                                           ⇒ 12
                                                           ⇒ 13
```

The different values may then be joined together as a single value:

```
(stream->list
   (join(+ 10(let(((list ...a x ...b) '(1 2 3))x))))) ⇒ '(11 12 13)
```

The join special form returns a stream of all the values of its argument expression. The procedure stream->list creates a list of the values from that stream. The operations of generating multiple values, applying operations to multiple values, and then joining those results together over and over again is an enduring metaphor in Logistica programming.

Here is a simple example program using multiple values to solve the 'eight queens' puzzle. The eight queens puzzle asks how to place eight queens on a chessboard so that no queen may be captured by any other queen, i.e., where no two queens are on the same row, column, or diagonal. One way to solve this problem is to work down the board, row by row, placing a queen in some column position in each row such that no previous queen has been placed in that same column and or along any diagonal from that position. We think of the queens' positions on a chessboard as being represented by a list of numbers where the last number represents the column position of the queen placed on the last row, and the second to last number represents the column position of the queen placed on the second to last row, and so forth. Thus `'(4 2 7 3 6 8 5 1)` represents the placement of 8 queens with the row 1 queen in column 4, the row 2 queen in column 2, the row 3 queen in column 7, and so forth. Likewise `'(8 5 1)` represents the partial solution of placing the row 6 queen in column 8, the row 7 queen in column 5 and the row 8 queen in column 1. Using this representation, the following program solves this puzzle:

```
(define (queens(list ...s col ...t)
                 (!and b(!not(list ...p
                                  (!test c(=(+(length(list ...p))1)
                                            (abs(- col c))))...q))))
         (queens(list ...s ...t)(cons col b)))
(define!(queens '()b) b)
```

The first definition of **queens** takes the list of remaining column positions, and a partial solution consisting of a list of column positions of queens already placed on the latter rows of the board. This procedure checks — for each column in the list of columns — whether some queen already placed on the board can capture a queen at that position through some diagonal. If so, it fails, but if not, the procedure recursively calls itself. The call is made using the list of columns with the currently chosen column eliminated and the new solution board consisting of adding the newly chosen column position to the next early row position of the old partial solution. The second definition of queens simply returns the resulting solution when all the queens have been placed.

Calling the **queens** procedure with the initial list of columns and an empty initial board produces 92 solutions, of which the first few are shown below:

```
(queens '(1 2 3 4 5 6 7 8)'()))        ⇒  '(4 2 7 3 6 8 5 1)
                                       ⇒  '(5 7 1 3 8 6 4 2)
                                       ⇒  '(5 2 4 7 3 8 6 1)
                                       ⇒  '(6 4 2 8 5 7 1 3)
                                       ⇒  '(3 5 2 8 6 4 7 1)
                                       ⇒  '(4 6 8 3 1 7 5 2)
                                       ⇒  '(3 6 4 2 8 5 7 1)
                                       ⇒  '(6 3 7 2 8 5 1 4)
                                       ⇒  and so forth
```

If only one solution is desired we call:

```
(head(join(queens '(1 2 3 4 5 6 7 8)'()))))
                                       ⇒  '(4 2 7 3 6 8 5 1)
```

3.2 The Symbolic Nature of Logistica

The second difference is that when Logistica evaluates an expression it returns the normal form of that expression rather than the meaning of that expression. The reason for this difference is that Logistica always has a representation for its expressions and their normal forms, whereas it may not have a representation of the meanings of expressions if those meanings refer to things outside of Logistica. For example, the expression π refers to a real number which is not itself an expression of Logistica, just as the expression Colonel'William'Barret' Travis refers to the commander of the Alamo who is not himself an expression in Logistica. Thus, unlike traditional programming languages such as Scheme whose expressions are restricted to those expressions whose meanings are themselves expressions in some datum language, Logistica makes no such restrictions whatsoever. In the simple case, where the expression being evaluated is the name of an expression in some datum language, Logistica operates just like Scheme, except that the final output is quoted. The symbolic nature of Logistica is well exemplified by the following automatic theorem prover for the propositional logic, which proves theorems in the propositional logic by symbolically rewriting them into conjunctive normal form and then testing each disjunction for the occurrence of a proposition and its negation:

```
(define $and '$and)
(define $or '$or)
(define $not '$not)
(define!($or) #F)
(define!($or ...a($not x) ..b x ...c) #t)
(define!($or ...a x ..b($not x)...c) #t)
(define!($or ...a #t ..b)#t)
(define!($or ...a($or ..b)...c) ($or ...a ..b ...c))
(define!($or ...a($and x ..b)...c)
        ($and($or ...a x ...c)($or ...a($and ..b)...c)))
(define!($and) #t)
(define!($and ...a #t ..b) ($and ...a ..b))
(define!($not($or ...(!do(i 1 n)(!ref i x))))
        ($and ...(!do(i 1 n)($not(!ref i x)))))
```

```
(define!($not($and ...(!do(i 1 n)(!ref i x))))
        ($or ...(!do(i 1 n)($not(!ref i x)))))
(define!($not($not x)) x)
(define!(implies x y) ($or($not x)y))
(define!(iff x y) ($and(implies x y)(implies y x)))
```

The first three definitions define `$and`, `$or`, and `$not` to be symbolic. Because of the cuts on all the succeeding definitions, the effect of these rules is to return these symbolic values only if no other definition is applicable. The next six definitions define properties of `$or`, namely that `$or` of no arguments is `#f`, left and right inverses, the zero law of `$or`, associativity, and distribution `$or` over `$and`. The next two definitions define properties of `$and`, namely that $and of no arguments is `#t` and the identity law. The next three definitions are deMorgans laws and the law of double negation. The last two definitions are the definitions of `implies` and `iff`.

In this program each of the variables: `$and`, `$or` and `$not` is used both as a procedure and as a symbolic data structure to which other definitions are applied. For example, the `$or` distribution law uses `$and` as a data structure, the `$or` associativity law uses `$or` itself as a data structure, and the `$or` inverse laws uses `$not` as a data structure. Since these variables are also used as procedures, they can be said to be used as both procedures and data structures. Being able to treat variables as both data structures and procedures is an important capability for describing the semantics of complex domains such as manufacturing enterprises. There is a parallel trend in computing systems towards a merging of data structures and procedures.

This program can be used to prove theorems in propositional logic when the variables in the theorem are given symbolic values. For example the variables `a, b, c` may be defined to be symbolic values as follows:

```
(define a _a)
(define b _b)
(define c _c)
```

Evaluating the following theorem then results in `#t`:

```
(implies($and(implies a b)(implies b c)(implies c a)
            ($or a b c))
        ($and a b c)))                          ⇒ #t
```

3.3 Overview of Logistica

A Logistica program denotes a function which maps a tuple of streams into a stream. A stream (or 'lazy list') is either empty or an ordered pair consisting of a 'head' and 'tail'. The head is a data object. The tail is either a stream or a 'promise'. A promise is a procedure that when executed produces the stream representing [the rest of] the original stream. The fact that a Logistica program is a function from a tuple of streams to a stream can in many cases be ignored, however. Instead, we can take a conceptually simpler, more traditional view that Logistica is a function that maps a tuple into a value. This latter view will generally be taken outside of this section. However, precision sometimes requires one to speak of Logistica in terms of its underlying streams.

A user of Logistica writes a program as a series of definitions and an expression denoting the function from a tuple of streams to a stream. Multiple definitions for each symbol are allowed. The expression produces multiple values, each one being an item in the result stream. The user does not need to explicitly deal with streams, however, since the stream-handling process is built into Logistica's expression evaluation. We call the resultant stream a 'control stream'. We do this to avoid confusing it with streams that the user explicitly creates as data objects. These latter objects we simply call 'streams'.

Control streams are essential for building deduction systems because at any given point in a deductive process it is usually possible to apply any number of different deduction steps, each one leading to a different result. Streams capture this notion of deductive alternatives by allowing each value of the control stream to represent an alternative deductive result.

Various amounts of control may be exerted over the creation of the control stream. If no restrictions are placed on the execution, then *all possible alternatives* are produced. A branch of the deduction may be eliminated if one of the steps returns 'failure', i.e. the empty stream. Alternatives at a given point which have not yet been tried may be 'cut' — either explicitly, or by specifying rules to be used to determine when and what to cut. Cut alternatives will never be executed nor do they contribute to the creation of the resulting control stream. Specific solutions may be selectively produced by transforming the control stream into an ordinary stream, i.e., an object that can be directly manipulated.

A class of potential deduction steps is encoded as a procedure. In addition to procedures representing *total functions* abstracted with a list of unique formal parameters, Logistica allows procedures representing *partial functions* abstracted with a list of patterns that control when a procedure applies to a particular tuple of data. These patterns include variable, literal, list structure, list search (segment variables), and tree search patterns (schemators). Because some patterns may match in more than one way, the patterns may contribute to alternative paths of execution. The application of lambda abstractions is done by pattern-matching of the patterns against the input arguments.

In Logistica arbitrary data structures may be applied to arguments. When the data structure is not an abstraction, the tuple containing the data structure followed by the arguments is returned. This allows symbols to function both as procedures and as data structures at the same time and facilitates symbolic computation (as noted earlier above).

These features while powerful in themselves, are integrated with the language Scheme, a popular LISP dialect. This integration gives the programmer a seamless and natural progression of tools, from the straightforward functional programming approach of LISP to the powerful and highly-descriptive expression of alternative computations afforded by Logistica. A brief description of the Scheme-like features of Logistica follows.

Following ALGOL and Scheme, Logistica is a statically-scoped programming language. Each use of a variable is associated with a lexically-apparent binding of that variable. For this reason static scoping is also called lexical scoping.

Logistica has *latent* as opposed to *manifest* types. Types are associated with values rather than with variables. Some authors refer to languages with latent types as 'weakly-typed' or 'dynamically-typed' languages. Other languages with latent types are Scheme, Dylan, APL, Snobol, and other dialects of LISP. Languages with manifest types (sometimes referred to as 'strongly-typed' or 'statically-typed' languages) include ALGOL 60, Pascal, and C.

The values of Logistica expressions are called 'objects'. This use of the term object in this context predates, and should not be confused with, its use in object-oriented programming languages. All objects created in the course of a Logistica computation, including procedures and continuations, have unlimited extent. No Logistica object is ever destroyed. The reason that Logistica does not (usually) run out of storage in memory is that it reclaims the storage space occupied by an object, if the object cannot possibly matter to any future computation. This is a more sophisticated form of what is commonly called 'garbage collection' in most LISP environments. Other languages in which most objects have unlimited extent include APL, Scheme, Dylan, and other LISP dialects.

Logistica procedures are objects in their own right. Procedures can be created dynamically, stored in data structures, returned as results of procedures, and so on. Other languages with these properties include Scheme, Dylan, Common LISP and ML.

Arguments to Logistica procedures are always passed by value, which means that the actual argument expressions are evaluated before the procedure gains control, whether the procedure needs the result of the evaluation or not. Scheme, ML, C, and APL are four other languages that always pass arguments by value. This is distinct from the 'lazy-evaluation' semantics of Haskell, or the 'call-by-name' semantics of ALGOL 60, where an argument expression is not evaluated unless its value is needed by the procedure.

In addition to procedures, Logistica has objects called 'macros'. Macros receive their arguments unevaluated. The environment for macros is taken from the calling environment, and is augmented with the macro's parameters and local definitions. Like procedures, macros produce many values. Each value is re-evaluated in the calling environment. Macros can be used to define new special forms and to simulate normal-order evaluation.

Logistica's model of arithmetic follows Scheme and is designed to remain as independent as possible of the particular ways in which numbers are represented within a computer. In Logistica every integer is a rational number, every rational is a real, and every real is a complex number. Thus the distinction between integer and real arithmetic, so important to many programming languages, does not appear in Logistica. In its place is a distinction between exact arithmetic, which corresponds to the mathematical ideal, and inexact arithmetic on approximations. As in Common LISP, exact arithmetic is not limited to integers.

3.4 The Nonmonotonic Rule System: NONMON

NONMON is a set of Logistica rules which implements nonmonotonic reasoning about actions. Before explaining NONMON, it is important to first explain what nonmonotonic reasoning is and how this differs from the monotonic reasoning carried out in classical first-order logic. First-order inference is monotonic: given a set of sentences Δ, the inferences that can be drawn from Δ are a subset of the inferences that can be drawn from the set Δ"ø', where ø is a sentence. In contrast to this, nonmonotonic reasoning allows that inferences that can be drawn from Δ may not be a subset of the inferences that can be drawn from Δ"ø'. Typically, the reason for this lack of monotonicity is that such systems allow inferences to be made in the presence of incomplete information. We call such inferences 'default inferences'. For example, consider the case where: Clyde is a raven, Claire is an albino raven, ravens are normally black, but albino ravens are not black ravens. Under classical first-order logic with monotonic inference, we are not allowed to conclude anything about the color of Clyde. If we allow nonmonotonic inference, then because we know of no other facts to the contrary, we can conclude that Clyde is black since that is normally the case. If we should later add the sentence (corresponding to ϕ) that Clyde is albino, then we are no longer allowed to conclude that Clyde is black, and, hence, our reasoning method is nonmonotonic, i.e., $\mathrm{Th}(\Delta) \not\subset \mathrm{Th}(\Delta \cup \{\phi\})$ where Th(Δ) is the set of theorems of Δ derivable through classical and nonmonotonic inference.

Classical first-order logic works well for problems in mathematics or in greatly simplified models of the real world. It does not work well for problems which are too large or too complicated to completely formalize, nor does it work well for problems dealing with change over time, nor for problems where not all information is known, such as explanation. A number of logical systems have therefore been proposed for carrying out nonmonotonic reasoning, but most of these proposed solutions lack effective proof theories (i.e., formal descriptions of their rules of inference represented in a recursive manner) and are merely semantic descriptions of the 'valid' inferences. Such systems are based on carrying out computations over infinite sets of sentences and are completely computationally intractable. However, we have developed a monotonic theory of nonmonotonic reasoning which has a recursive proof theory and which has in fact been implemented as a set of Logistica rules. This theory is our modal logic Z which is a fragment of second-order modal quantificational logic, involving quantification over propositions but not over properties. It is a recursively axiomatized logic with a recursively enumerable set of theorems. It is like classical logic except that it allows one to express notions of what is possible with respect to a body of evidence. For example, if K is the theory about Clyde and Raven articulated in the previous paragraph where the sentence that "Ravens are normally black" is expressed as the modal logic sentence: 'If it is possible with respect to K for a raven to be black, then that raven is black", then by the laws of the modal logic Z we can mechanically infer that K entails that Clyde is black. Likewise, if J is the 2nd theory articulated in the previous paragraph, where it is essentially K augmented with the additional fact that Clyde is an albino, J will not entail that Clyde is black.

The reason nonmonotonic reasoning is important for building automated manufacturing and design systems is that engineers and designers have a deep intuitive understanding of defaults about the mechanical actions of manufacturing processes, as well as the mental actions involved in both the design and manufacturing processes. Thus, if useful automatic aids for design and manufacturing are to be implemented, they too will need to 'understand' these implicit assumptions. In order to reason about actions, a set of rules called NONMON has been implemented in Logistica for purposes of enabling nonmonotonic reasoning about actions. These rules are essentially derived rules of inference of the modal logic Z.

We illustrate below some simple examples of NONMON's ability to reason about actions. For the readers convenience all the examples are expressed in English. The actual logical representations used by NONMON are described in a related technical report entitled, "A Logistica Deduction System for Solving NonMonotonic Reasoning Problems using the Modal Logic Z" [Brown1993e]. The areas of reasoning include: reasoning about the future; reasoning about the past; counterfactual reasoning; and reasoning about concurrent actions.

3.4.1 Reasoning About the Future

Reasoning about the future involves the prediction of the results of carrying out a sequence of actions starting from some initial situation.

<u>Problem 1. Frame Problem</u>

Assumptions:	*After an action is performed, things normally remain as they were.*
	A block is on the table if and only if it is not on the floor.
	When the robot grasps a block, the block will be normally in the hand.
	When robot moves a block onto the table, the block will be normally on the table.
	Moving a block that is not in the hand is an exception to this rule.
	Initially block A is not in the hand.
	Initially block A is on the floor.
Conclusions:	*After the robot grasps block A, waits, and then moves it onto the table, the block will be on the table.*

3.4.2 Reasoning About the Past

Reasoning about the past involves filling in the missing details or giving a plausible explanation of what occurred in the past, from an incomplete partial description of what happened.

Problem 2. Unknown Initial Conditions

Assumptions:	*After an action is performed, things normally remain as they were.*
	When the robot grasps a block, the block will be normally in the hand.
	When the robot moves a block onto the table, the block will be normally on the table.
	Moving a block that is not on the hand is an exception to this rule.
	Initially block A was not on the table.
	After the robot moved A onto the table and then waited, A was on the table.
Conclusion:	*Initially A was in the hand.*

Problem 3. Unknown Actions

Assumptions:	*After an action is performed, things normally remain as they were.*
	When the robot grasps a block, the block will be normally in the hand.
	When the robot moves a block on to the table, the block will be normally on the table.
	Moving a block that is not in the hand is an exception to this rule.
	Initially block A was not on the table.
	Initially block A was not in the hand.
	After the robot grasped some block and then moved some block onto the table,
	A was on the table.
Conclusions:	*The block that was grasped was A.*
	The block that was moved on the table was A.

Problem 4. Actions of Unknown Kinds

Assumptions:	*After an action is performed, things normally remain as they were.*
	When the robot grasps a block, the block will be normally in the hand.
	When the robot moves a block onto the table, the block will be normally on the table.
	Moving a block that is not in the hand is an exception to this rule.
	Initially block A was not on the table.
	Initially block A was not in the hand.
	After the robot performed two actions, A was on the table.
Conclusions:	*The first of the two actions was grasping A.*
	The second of the two actions was moving A onto the table.

Problem 5. Unknown Order of Actions

Assumptions:	*After an action is performed, things normally remain as they were. When a person accepts a job offer from some employer, he will be employed by that employer. If Bill is offered a job at Berkeley or at Stanford when he is unemployed, he will accept it. Bill is currently unemployed.*
Conclusion:	*After Bill is offered jobs at Berkeley and Stanford at two different instants of time, he will be employed either by Berkeley or by Stanford.*

Problem 6. Unexpected Change

Assumptions:	*After an action is performed, things normally remain as they were. When the robot moves a block to another location, the block will be normally at that location. After the robot moved a block to Location 1 and then to Location 2, the block changed its color.*
Conclusion:	*The block changed its color only once, either after the first move, or after the second.*

Problem 7. Unexpected Absence of Change

Assumptions:	*When the robot moves a block to another location, the block will be normally at that location. After the robot moved block A onto the table, and then moved block B onto the table, at most one of the blocks A, B was on the table.*
Conclusion:	*After the two actions were performed, exactly one of the blocks A, B was on the table.*

3.4.3 Reasoning About Actions That Might Have Happened: Counterfactual Reasoning

Counterfactual reasoning is reasoning about what might have been, and usually takes the form, "suppose that x were to be the case, then y." For this statement to be a counterfactual conditional, x must be counterfactual, i.e., contrary with the present situation.

Problem 8. Counterfactual Reasoning about Unexpected Change

Assumptions:	*After an action is performed, things normally remain as they were. When the robot moves a block to another location, the block will be normally at that location. After the robot moved block A to Location 1, block B changed its color.*
Conclusion:	*Block B would have changed its color if the robot had moved A to Location 2.*

3.4.4 Reasoning About the Future in the Presence of Concurrent Actions

Concurrent actions for our purposes are those actions that occur during the same transition from one state to the next. At present, there seems to be no completely formal work on nonmonotonic reasoning about concurrent actions.

Problem 9. Concurrent Actions

Assumptions: *When two actions are performed concurrently, their effects are normally combined.*
When the robot moves a block to another location, the block will normally be at that location.
Conclusion: *After block A is moved to Location 1 and block B is concurrently moved to Location 2, A will be at Location 1 and B will be at Location 2.*

Problem 9a. Conflicting Concurrent Actions I

Assumptions: *When two actions are performed concurrently, their effects are normally combined.*
When the robot moves a block to another location, the block will normally be at that location.
Conclusion: *After block A is moved to Location 1 and block A is concurrently moved to Location 2, either A will be at Location 1 or A will be at Location 2.*

Problem 9b. Conflicting Concurrent Actions II

Assumptions: *When two actions are performed concurrently, their effects are normally combined.*
When the robot moves a block to another location, the block will normally be at that location.
Conclusion: *After block A is moved to Location 1 and block B is concurrently moved to Location 1, either A will be at Location 1 or B will be at Location 1.*

4. Approach Used to Design and Develop Logistica

As previously mentioned in Section 2., Logistica is a tool used to develop high-level reasoning systems for manufacturing and management applications. A Logistica System is thus Logistica plus the high-level rules modeling such reasoning. Thus, our overall approach used in developing such reasoning systems, was to abstract out the common reasoning functionality involved in such reasoning and to embed this functionality into the Logistica language. We then implemented and documented this language, and tested it on a very wide range of reasoning problems to ensure its generality and usefulness. The design of the language is described in the Logistica Reference Manual [Brown93a] and the wide range of reasoning easily expressible in it is described in the Logistica Programmer's Manual [Brown93d].

There are three main lessons to be learned from the success of this language development effort. The first lesson is that it is important to abstract out basic reasoning capabilities and to embed them into the underlying language, as this allows one to avoid implementing these capabilities over and over again. This capability is precisely what let us succeed in developing the NONMON nonmonotonic rule systems for reasoning about actions. Without this capability for quickly implementing such sophisticated reasoning systems, this project would not have been able to be completed. In other words, traditional programming methods that did not take advantage of generic or reusable reasoning rules would have been entirely inadequate from both a technological and methodological standpoint.

The second lesson we learned was that new programming systems are greatly enhanced if their syntax and semantics can both be viewed as a logical extension from previous elegantly designed systems. As we started the development of Logistica, we initially used our own LISP-like language for Logistica syntax. However, within a few months, it became clear to us that Logistica should be based as much as possible on the most theoretically elegant dialect of LISP, a dialect known as Scheme. Thus in evaluating Logistica expressions, even though expressions may have multiple values in Logistica they look like the single-valued expressions of Scheme. Likewise the pattern-matching capabilities of Logistica are implemented as patterns in the formal parameter list of a function and are an extension of what is normally allowed there. Because Logistica looks like the well-known Scheme dialect of LISP (even though it is much more sophisticated semantically), this facilitates its acceptance by scientists and engineers because they do not have to learn an entirely new language that does many of the same things as languages they already know do. They essentially have only to learn the differences which make Logistica more expressive and more general.

The third lesson learned is that our users find significant value both in Logistica itself and in the ideas and technologies inherent in Logistica. For example, even though Allied Signal has purchased the Logistica system with its nonmonotonic rule system NONMON for dealing with reasoning about actions, it has also found significant use for using Logistica as an implementation language for major aspects of its manufacturing solid modeling reasoning. This application is independent of the NONMON rule system. Likewise, Ontek has found that the deduction technology of Logistica itself is useful to handle a number of system problems in their knowledge representation system. These applications of Logistica are described in the next section.

5. Applications of Logistica

During the course of this project Logistica was applied in two industrial/commercial settings. One application was at Ontek Corporation as part of the development of its PACIS software system and the other was at the DOE's Kansas City Plant run by Allied Signal Aerospace as part of its engineering and manufacturing operations. These applications of Logistica are described in sections 5.2 and 5.3, respectively. Section 5.1 describes the overall objectives and criteria for applying Logistica under this project.

5.1 Objectives and Criteria for Application of Logistica

The purpose of applying Logistica under this Phase II SBIR project was to establish a 'proof of principle'. For this reason we picked two very different applications: one as an embedded theorem prover to support a very general, very powerful enterprise information integration platform under development as part of another Air Force ManTech program; and one to directly support representation of complex end-user rules in a real-world manufacturing engineering domain (namely process planning). The latter demonstration project was chosen because Integrated Product Process Development (IPPD)/Concurrent Engineering (CE) is becoming a critical productivity-enhancing approach in a number of commercial and aerospace/defense industries. IPPD/CE requires sophisticated tools to support description of the information entities, processes and business rules used by an enterprise in the course of designing, manufacturing and supporting its products. Logistica lends itself to the description of such business rules, and to the use of such descriptions by automated reasoning systems to support decision-making in the enterprise. These two applications are described respectively in sections 5.2 and 5.3.

5.2 Application to Ontek's PACIS Under ITKM CDIT

5.2.1 Technical Requirements for PACIS under ITKM CDIT

Under the ITKM CDIT project, Ontek has identified end-user needs and requirements for production-viable Integration Toolkits and Methods - Corporate Data Integration Tools (ITKM CDIT). These tools essentially comprise an enterprise information integration platform based on the ANSI/ISO Three-Schema [Database Management] Architecture (i.e., using external, conceptual and internal schemas) [ISO, 1987; Tsichritzis and Klug, 1978]. The technology is being operationalized using Ontek Corporation's Platform for the Automated Construction of Intelligent Systems (PACIS). 'Production viable' means that ITKM CDIT technologies must meet or exceed the performance capabilities of existing technologies and be sustainable in large-scale, heterogeneous information resource environments. Performance and sustainability comprise the two primary end-user or 'enterprise business-level' requirements.

Requirements for performance deal both with the local performance of the ITKM CDIT toolkit and its ability to optimize the performance of the foreign systems with which it must interact. Several sustainability requirements address the subsumptive integration of commercial database management systems (DBMSs) and their associated databases. Included are requirements for compatibility with existing computing environments, as well as functional capabilities to support direct, dynamic, and ad-hoc access and update from the PACIS platform to data stored in commercially-available DBMSs. Examples of such database management systems include IBM's IMS and DB2, and Oracle Corporation's ORACLE. Object-oriented DBMSs are also being utilized in an increasing number of production DBMS environments, and this trend is expected to ramp-up dramatically. The access capability must therefore be applicable to heterogeneous environments that run the gamut from 'flat' file management systems, through traditional hierarchical, network and relational DBMSs, to emerging object-oriented DBMSs. This access should be user transparent — i.e., end-users should not have to have any knowledge of the location of the data or the underlying systems that manage the databases. The toolkit must also do more than simply access the data in existing DBMS environments; an enterprise information integration platform must be able to dynamically interact or interoperate with those legacy systems. This includes the requirement to perform and propagate updates across a network of distributed, but integrated, systems. Other performance and sustainability requirements deal with the toolkit itself, in particular the need for the individual components or tools to be tightly integrated so that from a user standpoint, the tools can be viewed simply as elements of a complete suite of tools or a unified toolkit.

Based on these two end-user requirements for performance and sustainability at the 'enterprise business level', Ontek then identified detailed 5 technical requirements at the 'technology level'.

1. One key technology-level requirement is for Reference Mediation. Reference mediation is important for optimizing performance, by enabling the selection of different access methods depending on the nature of the data being accessed and the data management environment in which it resides. This is especially critical in large-scale database environments, like one would expect to find in integrated enterprises.

2. A second technology-level requirement is the capability to compile a single PACIS operation, such as an end-user query, into a complex or collection of subqueries expressed in the particular DBMS data manipulation languages (DMLs) of the various databases targeted by the query. This capability is critical for direct, transparent, dynamic, ad-hoc access to distributed, heterogeneous databases to be viable in a large-scale production environment — i.e., it is critical for performance. Examples of DMLs are DL/I for the IMS DBMS, and SQL for the ORACLE and DB2 DBMSs. The compiled DMLs must also be tailored to the specific computing environments (e.g., MVS/CICS, UNIX, etc.) in which the

DBMSs reside. The compiled subqueries can then be distributed to, stored in, and executed on the machines containing the target databases.

3. A third important technology-level requirement can be referred to as Global Instantiation — for sustainability, the toolkit must be able to create instances (i.e., new records) in systems external to the PACIS-based data/knowledgebase, including systems in the enterprise's existing information resources environment.

4. A fourth technology-level requirement, mandated by the end-user requirement for sustainability, is the existence of a Global Condition Handling mechanism to automatically propagate the consequences of events occurring in one area of the system across other system elements for which these events have consequences.

5. The fifth major technology requirement, is for Global Presentation. This capability will enable the various ITKM CDIT technologies to operate as a single logical toolkit comprised of an underlying platform (i.e., PACIS) supporting multiple functional tools which provide various interrelated services such as: conceptual analysis and modeling; enterprise model integration; legacy database query, update and integration, etc. Tool integration imposes more detailed requirements on the use of the baseline PACIS representation system, and drives the requirement for a new global environment to support common, unified graphical user interfaces (GUIs).

The five key technical requirements described in the previous section, particularly the compiler, the global instantiation capability and the global condition handling capability, drive a sixth important technical requirement — the need for the ITKM CDIT toolkit to perform Pattern-Directed Invocation. Like the other five, this requirement is also necessary to meet the enterprise business-level requirements. However, the first five technical requirements may be considered 'vertical' requirements that address specialized areas. There are some inter-relationships among them, but they can be thought of as fundamentally distinct. The sixth requirement, on the other hand, is a 'horizontal' requirement, which, while addressing distinct functional capabilities, serves to enable or support the capabilities of the other five technical requirements. This can be explained in more detail as follows:

6. The sixth technology-level requirement is for a functional capability to perform Pattern-Directed Invocation. The problems of scope, complexity, sustainability and performance that information integration raises requires new techniques for basic system operations such as compilation, condition handling, instantiation and process invocation. The melding of database operations — particularly instantiation — and traditional programming operations requires pattern-directed invocation. As noted earlier, a compiler, a global instantiation facility and a global condition handling facility are necessary to provide the performance and sustainability required by day-to-day system operations in a large-scale, integrated information environment. Even the construction of a compiler for PACIS, as well as the creation of both a global instantiation facility and a global condition handling facility, entail that PACIS support pattern-directed invocation. The compiler requires pattern-directed invocation because the language translation job is just too complicated to try building it using purely deterministic methods. The global instantiation facility and the global condition handling facility require pattern-directed invocation because they must have the ability to invoke routines based on the current state of the system, rather than based on a previously-defined calling sequence.

Two lower-level technical requirements, which represent further refinements of the overall requirement to support pattern-directed invocation, have also been identified. Existing pattern-based languages such as PROLOG and ML allow for pattern-directed invocation, but with certain limitations. These limitations must be overcome in order to meet other ITKM CDIT requirements for a compiler, a global instantiation capability and a global condition handling capability. The additional requirements described below are based partly on overcoming the limitations of existing pattern-based languages.

1. Alow multiple matches on operations and definitions. The PACIS compiler requires this capability because there may be many possible target code solutions to any one query, and deciding which one is best for the purpose at-hand requires having them all available at the same time for comparison. A condition mechanism requires this because there is more than one possible action, or condition handler, that may be invoked during a fault in a three-schema database transaction. For example, you might have to do both a DB2 image rollback and an IMS checkpoint/restart.

2. Support dynamically-definable interpreter control structures — This will be a critical capability in the ITKM CDIT environment, since the dynamic semantics of other systems must be subsumed to support their integration. The subsumption of dynamic semantics includes representing and eventually executing the control structures these disparate systems use. These systems may well have different meanings for the same control structure named in another system. This puts the burden on the PACIS interpreter to carry multiple definitions of control structures.

5.3.2 How Logistica Meets ITKM CDIT Technical Requirements

Logistica helps meet the ITKM CDIT technical requirements by providing a working system which satisfies the pattern-directed invocation requirements described above in Section 5.3.1. Use of Logistica in this 'proof of principle' setting has indicated that these difficult requirements can be satisfied to a sufficient degree, and has helped demonstrate specifically how one might go about implementing this capability in a production version of an enterprise information integration system.

A copy of the Logistica system was delivered to, and installed at, Ontek. Ontek programmers then studied the Logistica system and determined that this technology appeared to be able to satisfy many of the requirements for a pattern-directed invocation capability. AIR worked with Ontek staff to help them understand the basic ideas behind Logistica and to explore possible ways to embed Logistica's pattern-directed invocation capability within Ontek's C-based PACIS knowledge representation product. Ontek is currently planning to reimplement various Logistica functionality in C as part of PACIS. AIR will be provided a royalty or other similar compensation from the sale or lease of PACIS software incorporating Logistica's pattern-directed invocation capability.

Two specific examples of how Logistica meets the required pattern-directed invocation requirements of PACIS are described below:

1. The Queens example described in Section 3.1 demonstrates that Logistica provides for multiple matches on both operations and data as required for the PACIS condition handler. Logistica provides explicit support for multiple matches on operation definitions and multiple input states. A given input to a pattern-directed interpreter may possibly match: none; exactly one; or many operation definitions. Also, a given input operation may have: no arguments; exactly one set of arguments (one input state); or many sets of arguments (many input states). The power of a pattern-directed invocation mechanism depends on how it handles the possible operation matches and their application to the possible

number of input states. Of the existing pattern-based languages such as ML and PROLOG, Logistica is the only one that can handle all of the possible cases of operation matches and input states. In ML, for instance, even if there are several operation definitions which match the input pattern, only the first match will be returned. This means that ML can only execute one of possibly many definitions for a given operation. PROLOG does return matches of multiple operation definitions, but whether one or many operations are returned, they are applied to only one input state. In the case where there are many operation definition matches, as well as multiple input states, Logistica invokes the Cartesian cross-product of 'operation definitions X input states'.

2. The propositional deduction system illustrates that Logistica supports dynamically-definable interpreter control structures as is required for the subsumptive requirements of PACIS. Thus Logistica provides dynamically-definable interpreter control structures whereas older languages, such as PROLOG, 'wire down' the basic syntactic control structures they use. In other words, the control structures of these other languages are static. Symbols such as 'and' are primitive and have only one definition. In Logistica, even these basic control structures can themselves be defined. In addition, since Logistica allows for multiple definitions of operations, the control structures can be multiply-defined.

In summary, Logistica has provided Ontek with the basis for its required pattern-directed invocation capability. That capability can now be operationalized in PACIS and used for ITKM CDIT technologies. This is one of the most important requirements of Ontek's ITKM CDIT project, as this requirement is a basic enabling requirement for the 5 other technological requirements of the project.

In early 1994, Ontek's technologies (which include logic based on the Logistica technology) will be ready for demonstration. Ontek eventually plans to commercially market products based on ITKM CDIT technology.

5.3 Deployment into Allied Signal

Logistica has been deployed (i.e., sold, installed, implemented and put into operation) for Allied Signal Aerospace at the Department of Energy's (DOE's) Kansas City Plant. The business transaction took the form of a sale of 1 copy of Logistica running on a SUN SPARCstation, and the sale of 1 copy of the nonmonotonic rule system, NONMON. The nonmonotonic rule system was implemented in Logistica and makes use of its pattern-directed invocation capability and its ability to represent multiple values. This nonmonotonic reasoning capability was critically needed by Allied Signal for developing an automated manufacturing system.

Although Allied Signal originally purchased Logistica, as well as the NONMON nonmonotonic rule set written in Logistica, for use in understanding and representing the results of mechanical manufacturing actions and mental actions involved in the design process, Allied Signal subsequently discovered that Logistica itself (without the NONMON rule set) is a very useful language for implementing translation processes, and for verifying design constraints and consistency. In fact, they have found Logistica to be so useful that one development group is considering rewriting substantial parts of their existing manufacturing system in the Logistica language. The decision to consider this further use of Logistica was made despite Allied Signal's considerable past investment and experience in other, less powerful languages. It was felt that Logistica offered the potential of providing much needed system capabilities which could not be operationalized (or at least not as efficiently and cost-effectively) using existing technologies. These applications of Logistica without NONMON are briefly described in Sections 5.3.1 and 5.3.2 below. The originally-envisioned project, a longer-term application of Logistica coupled with NONMON, is described in Section 5.3.3 below.

5.3.1 Machining Form Feature Translation Project

Under the National Initiative for Product Data Exchange (i.e., NIPDE), the DOE's Kansas City Plant run by Allied Signal is currently working with a number of other companies to create a single unified model of machining form features. This work is being carried out in anticipation of creating new STEP (Standard for The Exchange of Product model data) application protocols in the area of form features such as pockets, holes, slots, bends, and chamfers. The application protocols take the form of 'context-driven integrated model' (CDIM) specifications. The basic idea is to have a company create product designs which are represented in a standard way, so that they can be transmitted to and used by other companies such as subcontractors, vendors, customers or support organizations. This work involves the following two steps:

1. First, the Navy RAMP Product Translation System (i.e., RPTS) project at the South Carolina Research Authority (SCRA) and Grumman Data Systems is providing a source of machining form feature files expressed as RAMP CDIM specifications. These files are initially created by using the Pro-engineer feature-based solid modeling system with a form feature-based editor. The files are then translated to the RAMP CDIM language. Because RAMP CDIM is already supporting significant applications downstream in the design, manufacturing and support processes, it offers a valuable source of data for NIPDE.

2. Second, although RAMP CDIM is currently supporting downstream applications, it is effectively still a 'closed system', rather than a consensus model produced and agreed upon by a wide-range of companies. It is therefore not acceptable to many companies as a 'standard' format for exchanging form feature data. Fortunately, there is a consensus model for product data exchange using the STEP standard that is acceptable to a wide range of companies, namely NIPDE. Numerous companies and groups have agreed to accept NIPDE CDIM product specifications and to use them for manufacturing applications that use form features. These applications include process planning, generative inspection, and costing. The companies and groups that have agreed to use NIPDE specifications for this purpose include: the National Center for Manufacturing Sciences (NCMS), Pratt & Whitney, Ford Motor Company, Martin Marietta, Lawrence Livermore National Laboratory, Cognition Systems, and Techno-Soft (a company spawned as a result of the Air Force rapid design system program).

The missing link between the first and second steps is that the available product data is expressed early in the process in RAMPS CDIM format, whereas the companies downstream want the data to be expressed in NIPDE CDIM format. Allied Signal has taken on the task of translating the RAMPS CDIM into the NIPDE CDIM language. In order to do this, they need an advanced reasoning technology that allows translation rules to be easily expressed in a modular way. Since Logistica provides this capability, Allied Signal decided to undertake a project to write the necessary translation rules in the Logistica language. They are also using the Logistica system to execute these rules (i.e., to process through the rules with data, in the same manner as traditional computer programs are typically executed). Logistica is therefore crucial to the success of the machining form feature translation project, as every piece of product data in every test file will be manipulated and translated by the Logistica rule system.

5.3.2 The Ford Project

Allied Signal is currently involved in negotiating a cooperative agreement with DOE and Ford Motor Company. The agreement covers a multi-year, multi-million dollar project to implement a STEP repository at Ford. Allied Signal is proposing to use Logistica on this pending project to check reasoning about Euler constraints to insuring consistent geometry, and to do inference on product designs.

5.3.3 Longer-Term Plans

Together, Logistica and the nonmonotonic rule set NONMON provide Allied Signal with a unique capability for reasoning about the consequences of action, in general. The longer-term plans for use of Logistica at Allied Signal involve building a rule base for the domain of process planning for manufactured parts. The rule base will include descriptions of various actions, both planned and unplanned, which can occur in that domain. Using that rule base, along with the generic reasoning rules of NONMON, Logistica would be used to support planning and decision-making activities.

6. Conclusion

Logistica is a functional programming language used for implementing deductive reasoning processes, specifically for modeling and executing the logic or rules of processes or actions in some domain of interest. Logistica provides more generality than other languages of this nature, and is therefore more powerful and flexible in its application. It is particularly useful for cases where there are multiple input states to be studied, or where there are several alternative processing paths to be considered. Logistica is extremely useful in cases where the possible states or processing paths are not known or predictable in advance (i.e., where the states or processes are 'indeterminate').

Automating reasoning processes is important for developing 'smart' systems to support analysis and decision-making in complex enterprises. Logistica is intended to be used to describe the logic or rules of domains such as engineering and manufacturing operations, as well as their supporting management or business-level processes. Under this Phase II SBIR, the Logistica language was refined and then operationalized in a software tool. The software was developed and tested in a LISP processing environment, running on workstations under the UNIX operating system and on Apple Macintosh personal computers. Two applications of the technology were undertaken to establish a 'proof of principle' of Logistica's capabilities.

First, Logistica was used as the basis for designing a pattern-directed invocation capability for Ontek Corporation's Platform for the Automated Construction of Intelligent Systems (PACIS) software. This work was performed under the Air Force ManTech Directorate's Integration Toolkit and Methods (ITKM) program, Corporate Data Integration Tools (CDIT) project. Logistica was determined to be a quite suitable technology for supporting a pattern-directed invocation capability. Logistica is serving as an important technological basis for meeting many of the 'horizontal' technical requirements of ITKM CDIT — i.e., fundamental requirements which serve to support 'vertical' or specialized capabilities in specific technical areas.

Secondly, Logistica and its associated NONMON rule set have been successfully deployed into a major manufacturing enterprise — the DOE's Kansas City Plant run by Allied Signal. NONMON is a set of Logistica rules which implements nonmonotonic reasoning about actions. At Allied Signal, Logistica has been utilized on a major project involving translation between disparate product model data specification formats. Other uses are being planned or proposed include using Logistica rules as part of a STEP repository and as the basis for a system for reasoning about manufacturing actions, initially in the domain of manufacturing process planning.

In summary, this SBIR Phase II contract demonstrated the use of Logistica as a solution to several classes of real problems in information system and manufacturing environments. Phase III is intended to be the commercial deployment of this product either alone or as a part of a more general management and manufacturing system.

References:

Abelson, H. and Sussman, G. J., 1985. Structure and Interpretation of Computer Programs. MIT Press, Cambridge, Massachusetts, U.S.A.

Araya, C. and Brown, F.M, 1990. Schemata: A Language for Deduction. From Proceedings of the 9th European Conference on Artificial Intelligence, Stockholm, Sweden.

Brown, F.M., 1977. "Doing Arithmetic Without Diagrams". Artificial Intelligence, Vol. 8, Spring 1977.

Brown, Frank M., 1989. Final Report: Frame Planning in a Military Environment. Future Battle Laboratory, Ft. Leavenworth, Kansas, U.S.A.

Brown, Frank M., 1991. "The Modal Quantificational Logic Z Applied to the Frame Problem", in International Journal of Expert Systems Research and Applicatiions, Special Issue: The Frame Problem. Part A. Ford, K. and Hayes, P. (Eds). Vol. 3 number 3, pp169-206 JAI Press 1991. Reprinted in 'Reasoning Agents in a Dynamic World: The FRAME Problem". Ford, K. and Hayes, P. (Eds.), 1991. JAI Press.

Brown, F. M., et al, 1993a. Logistica 1.0 Reference Manual. Artificial Intelligence Research, Inc. 1993.

Brown, F. M., et al, 1993b. Logistica 1.0 User's Manual (Macintosh version). Artificial Intelligence Research, Inc., 1993.

Brown, F. M., et.al., 1993c. Logistica 1.0 User's Manual (Unix version). Artificial Intelligence Research, Inc., 1993.

Brown, F. M., et al, 1993d. Logistica 1.0 Programmer's Manual. Artificial Intelligence Research, Inc., 1993.

Brown, F. M.and Leasure, D., 1993e. "A Logistica Deduction System for Solving NonMonotonic Reasoning Problems Using the Modal Logic Z". Technical Report for SBIR Phase II Contract #F33615-91-C-5703. Artificial Intelligence Research, Inc., 1993.

Church, A., 1941. "The Calculi of Lambda-Conversion". Annals of Mathematical Studies, No. 6. Princeton University Press, Princeton, New Jersey, U.S.A.

Colmerauer, A., Kanoui, H., Pasero, R. and Roussel, P., 1973. Un Systeme de Communication Homme-machine en Francais. Research Report, Groupe Intelligence Artificielle, Universite Aix-Marseille II, France.

Hayes, P. J., 1973. Computation as Deduction. From Proceedings of the 2nd MFCS Symposium. Czechoslovakia Academy of Sciences, Prague, Czechoslovakia.

Hockenberger, R. L. and Reich, A. J., 1988. "Soldier-Machine Interface Design for a Military Operations Planning Expert System". Lockheed Corporation. Unpublished ALBM Report.

International Organization for Standardization (ISO), 1987. Information Processing Systems —Concepts and Terminology for the Conceptual Schema and the Information Base. ISO TR9007 (E). International Organization for Standardization (ISO).

Kowalski, R., 1974. Predicate Logic as Programming Language. From Proceedings of IFIP74. North-Holland, Amsterdam, The Netherlands.

Kowalski, R., 1979. Algorithm = Logic + Control. CACM, August 1979.

Nassi, I., et al, 1992. Dylan. Apple Computer, Inc., Cupertino, California, U.S.A.

Steele, G. L., Jr. and Gabriel, R. P., 1993. "The Evolution of LISP". ACM Sigplan Notices, Vol. 28, No. 3, March 1993.

Sombe, L., 1990. Reasoning under Incomplete Information in Artificial Intelligence. John Wiley & Sons, Inc.

Takeuchi, I., et al, 1983. "Tao-- A harmonic mean of Lisp, Prolog and Smalltalk". ACM Sigplan Notices, Vol. 18, No. 7, July 1983.

Tsichritzis, D. and Klug, A., (Eds.), 1978. "The ANSI/X3/SPARC DBMS Framework", Information Systems, Vol. 3, 173-191.

Van Hentenryck, P., 1989. Constraint Satisfaction in Logic Programming. The MIT Press, Cambridge, Massachusetts, U.S.A.