Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

TANGRAM: PROJECT OVERVIEW

R. R. Muntz
D. Stott Parker

April 1988
CSD-880032

Tangram: Project Overview †

R.R. Muntz

D. Stott Parker

Computer Science Dept.
University of California
Los Angeles, CA 90024-1596

ABSTRACT

Today, most computers are used for the modeling of real-world systems. Demands on the extent and quality of the modeling are growing rapidly. There is an ever-increasing need for environments in which one can construct and evaluate complex models both quickly and accurately.

Successful modeling environments will require a cross-disciplinary combination of different technologies:

System modeling tools
Database management
Knowledge base management
Distributed computing

None of these technologies by itself provides all that is needed. A modeling environment must offer high-speed retrieval and exploration of knowledge about systems, as well as integration of diverse information sources with existing modeling tools.

Tangram is a distributed modeling environment being developed at UCLA. It is an innovative Prolog-based combination of DBMS and KBMS technology with access to a variety of modeling tools.

April 28, 1988

†Supported by DARPA contract F29601-87-C-0072.

For more information contact:

Richard R. Muntz (muntz@cs.ucla.edu)
(213) 825-3546 ofc., 825-7879 scy.

D. Stott Parker (stott@cs.ucla.edu)
(213) 825-6871 ofc., 825-1322 scy.

Tangram: Project Overview †

R.R. Muntz

D. Stott Parker

Computer Science Dept.
University of California
Los Angeles, CA 90024-1596

1. Introduction: Modeling

> Harrison Brown warns that modern economies are complex and vulnerable because dangerous dynamic forces are at work (growing population, decreasing resource base, growing gap between rich and poor nations, political troublemaking on a world-wide scale, etc.). Despite these dynamic forces and the complexity of world economies, policy makers continue to make decisions without the benefit of the powerful analytical tools that are available to them. That the decisions they make are bad, is self-evident.
>
> – George Dantzig [31]

Today, numerous environments containing specialized tools are under development. Often these tools are for creating and managing models of some kind [35, 42, 74]. There are many kinds of analytic models alone:

- Stochastic processes/queueing models
- Statistical models
- Structural models
- Equational models (numeric or symbolic)
- General constraint-based models
- Rule-based models
- Semantic network/entity-relationship models
- Object-based models, with methods or behaviors.

In diverse science and engineering disciplines, ranging from medicine to the social sciences, models are applied in computer-aided specification, design, and analysis, including simulative and statistical analysis. The diversity of the models used stems partly from their evolutionary development in different fields.

Modeling will be one of the main enterprises of the information age. Today's emphasis on modeling will only increase in the future. Darwinian pressures force information systems to grow in sophistication, to better answer questions such as 'What happened?', 'What is going on?', and 'What would happen if...?' Future modeling environments must deal with three significant needs:

- Structurally complex data
- Deep interpretation of data
- Integrated management of data.

1.1. Structurally Complex Data

Data used in models is becoming more detailed in terms of structural complexity. Present modeling systems have trouble in representing this detail. For example, while structurally simple data is handled well by relational database technology, recently a great deal of attention has been given to more structurally complex data that has interesting, non-tabular structure. This data is commonly called 'complex objects' or 'non-first normal form' (NFNF) records in the data management field, and semantic networks or knowledge bases in other contexts.

With increasing detail of description, the list of differences between any two objects will grow. *Deep models* – models involving great detail, or multiple levels of detail – are always more structured than simple or generic models, which are sometimes called *shallow models* by comparison. As we model objects in greater detail, we must deal with the fact that some objects have individualized attributes that may be unknown or irrelevant for other objects. Relational structures model objects homogeneously, with every object being represented by the same sequence of parameters or attributes. Relational storage therefore becomes impractical when we seek greater detail.

The ability to store complex structures is powerful: with it, one can store not only facts about objects, but also rules governing the behavior of objects. This has immediate consequences in modeling, since it permits distinctions between data and interpretation of data to disappear.

1.2. Deep Interpretation of Data

In addition to managing more detailed data, the operations one wishes to do on this data become more sophisticated, or deep. These operations have been impractical in the past for a number of reasons.

First, analytical or abstracted models are often used instead of doing direct analysis of real-world data. There are inherent problems with such models:

(1) Abstracted models are often shallow. In some cases an abstracted model permits the researcher to get at essential aspects of the real-world system being studied, but then only at those aspects. Detailed information about a system requires direct analysis of larger quantities of data.

(2) Abstracted models require *estimation* of suitable parameter values. Typically, parameter estimates are defined as either the result of statistical queries against observed data, or the solutions of a system of constraints. It is not always straightforward to obtain these estimates.

(3) To be of real use, an abstracted model must be *validated* against the real-world data it supposedly describes. Again, while this can be thought of as a query (Does the model match the data?), validation is an unpleasant responsibility that is often avoided by model-builders.

Second, models may be too large for conventional machines, even mainframes. Simulation of microprocessors with significant numbers of gates takes weeks of CRAY-2 time. Interpretation of more detailed data will require significantly more processing power than is available today. Some argue that the current push into supercomputers is justifiable only because it will permit us to model systems that have been intractable in the past.

Third, researchers developing models have lacked appropriate data management tools. Laborious data extraction from tapes or files is a traditional problem in large-scale modeling.

Unfortunately, using modern DBMS will not be adequate for modeling needs.* The queries permissible in today's DBMS are very restricted: They map tables into tables using a handfull of well-defined operators. More generally, queries should be *any* kind of computable feature extraction operation.

It is important to consider what one really wants from a modeling environment. Three general operations are performed against a collection of data in the process of modeling: *abstraction* of models, *prediction* of the future using models, and *validation* of models against the past.

Abstraction is the least understood of the three kinds of operations, since it involves some sort of creative or associative recall on the part of the modeler. The process of abstraction requires compiling information from different contexts, or perspectives. In other words, abstraction requires feature extraction, filtering, data reduction, etc. Important operations today include:

standard relational query
aggregate computation (min, max, average, count, sum)
statistical analysis (regression, analysis of variance, etc.)
pattern recognition (trends, transients, etc.)
estimation of model parameters
estimation of certainty factors
induction of rules

No existing tools, even the most advanced database systems, provide what is needed here.

Prediction is a process of evaluating various possible scenarios consistent with a current world model. It typically involves some sort of simulation. Mathematical models involve structurally simple data, and either the solution of systems of equalities and inequalities or continuous-time simulation. What we might call structured models (with structurally complex data) involve either discrete event simulation, or some variety of rule- or object-based simulation.

Validation, finally, matches predictions obtained from a model of a system against the actual behavior of the system. This involves the definition of metrics or figures of merit on model performance, and verifying somehow (statistically or formally) that a model achieves a level of accuracy.

These three operations are used in an abstraction-prediction-validation cycle. The results of each cycle are used in refining the model for better accuracy. A reasonable modeling environment must support these three operations and their iterative application in the refinement of a model.

1.3. Integrated Management of Data

To see the need for integrated data management, consider the problems encountered in a simple case study. A 4-site model of the LOCUS distributed operating system was simulated using the

*Interestingly, work in the database field has approached modeling from another direction. A database is, after all, a model of some enterprise, or more generally of some collection of objects and relationships among them. The DBMS of today permit only static models. That is, they capture the *state* of some system. The behavior of these systems must be captured with external programs, not by the DBMS itself. Thus the DBMS accomodates simple models on vast quantities of data, instead of intricate models involving a few parameters.

PAWS simulation package by Steven Berson of UCLA.†

Altogether, nine different load-balancing algorithms were considered. In each experiment, labelled by single letters in the table below, all sites used the same criteria for deciding when to migrate process load to other sites. The criteria and offered load varied among the experiments: load balancing was done by inspecting cpu and/or disk queue lengths, and offered load differed by distribution and by number of customers (between 2 and 20):

Experiment	*Load balancing on*	*Work done*
r	disk queue lengths	Exponential
s	cpu queue length	Exponential
u	sum of cpu and disk queue lengths	Exponential
v	sum of cpu and disk queue lengths	Hyperexponential
w	sum of cpu and disk queue lengths	Erlangian
x	NO BALANCING, NO REMOTE ACCESS	Exponential
U	NO BALANCING	Exponential
V	NO BALANCING	Hyperexponential
W	NO BALANCING	Erlangian

The resulting average response times, in milliseconds, were obtained:

Experiment	*Number of Customers*									
	2	4	6	8	10	12	14	16	18	20
r	2490	2390	2440	3200	3520	4610	5830	6630	8900	10500
s	2320	2450	2820	3150	4080	4510	5160	7240	9510	11700
u	2470	2450	2620	2560	3600	3670	5010	6130	7920	10400
v	2430	2610	2260	2720	3210	3860	4660	5280	7680	7750
w	2330	2570	2720	2930	3260	4050	4690	6350	8060	9690
x	2750	2990	3140	3490	3910	6230	6190	8570	9060	10500
U	2700	2880	3680	4410	5130	5310	7190	8650	11100	13600
V	2220	3520	3100	3710	4780	6340	7480	9660	10900	12400
W	2580	3150	3200	3950	4640	5950	7780	8830	10600	13800

These results show that the load balancing strategies taking both cpu and disk queue lengths into account performed better than the others. It is difficult to 'get a feel' for what the differences are here, however.

Modest as it is, this experience underscores the importance of an environment for modeling:

(1) Data management of experiments requires enormous effort. The data above were extracted manually from the printed output of many PAWS runs (possibly with errors), required creation of several different versions of the data for different tools (*S*, *grap*, *tbl*, etc.), and so forth. The data above cannot be queried automatically now.

(2) Various types of measurement data that must be captured beyond the standard statistics (utilization, throughput, queue length, queueing time, chain population, point-to-point

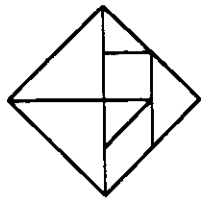
† Each site had 1 cpu and 1 disk. The access patterns were selected to be 80% local, 20% remote disk access, with an average disk seek time of 30ms. Load on the system was generated by between 2 and 20 customers. The PAWS simulation system was used to simulate several tens of thousands of events, in order to obtain mean response time.

timing, etc.) typified above. Event traces, for example, are needed for detailed understanding of what goes on in distributed systems: cyclic behavior, correlated events, race conditions, catastrophe-theoretic behavior, etc.

(3) There are many different kinds of queries that may be made against measurement data: statistical summary and analysis, pattern detection on time series, browsing through subsets of event trace, animation with start/stop/replay, and of course graphical display (bar plots, contour plots, histograms, scatter plots, time-series plots, etc. [10])

2. Tangram: A Modeling Environment

Tangram is an environment for developing models of the scope suggested above. It is implemented as an extension of Prolog that includes integration with the Unix environment and database managers, and provides distributed processing constructs. This section gives an informal overview of the design of the environment. In later sections we describe more thoroughly the individual research projects which together comprise Tangram.



2.1. Functions of Tangram

The main goal of Tangram is to provide an interactive modeling environment. The experiences of the previous case study highlight some functions that such an environment should provide. A functional diagram of Tangram in use is sketched in Figure 1. With this system a user should be able to:

1. Select models from the Model Base
2. Select an experiment from the Model Base
3. Run the experiment
   - 3.1 Find available tools from the environment Knowledge Base
   - 3.2 Execute the experiment with a specific tool (or testbed)
   - 3.3 Store results of the experiment in the Measurement Data Base
4. Query the results of the experiment interactively

We describe these capabilities further below.

2.1.1. Model Management

Just as DBMS are managers for data, Tangram is a manager of models. Model management includes the storage and retrieval of 'data dictionary' knowledge about available models, workloads (load generators, benchmarks), experiments, experiment output, and tools. Where multiple models are used to describe a single system from different viewpoints or levels of abstraction, model management also provides information on how these models relate.

The Tangram environment is to support various modeling packages, and possess knowledge on how these packages are applied. This knowledge comprises an expert system on easy, effective access to modeling tools. The SACON system developed by Bennett et al. [13] illustrates the kind of functionality needed: SACON inspects a structure and recommends a particular subprogram from the (large, complex) MARC environment for structural analysis. For example, given the description of an airplane wing, it applies knowledge about the domain to decide to use the MARC inelastic-fatigue program to analyze stress and deflection of the wing.

2.1.2. Measurement Data Management

Modeling experiments generate massive quantities of data. Tangram is concerned with issues in capturing this data from different tools, translating it to a common format, storing it, and supporting arbitrary queries with parallel processing. This presents challenges in developing data management technology. Not only is the data structured, but it also contains important temporal information. Also, current DBMS do not support 'exploration' of the data in the way that exploratory data analysis systems such as *S* [10] do. It is important to be able to support exploration of a model, encouraging a modeler to get an intuitive understanding of its behavior. The modeler should be able to view his model actually 'running' with various kinds of graphical displays, for example.

Parallelism can make interactive real-time modeling possible, where it would not be possible on this scale otherwise. We see stream processing as the most natural parallel data management paradigm. A great portion of the Tangram project is, then, concerned with stream processing. The *Tangram Stream Processor (TSP)* is a stream-based system founded on the abstraction of transducers. A transducer maps input streams to output streams. We discuss TSP in greater detail in a later section.

2.1.3. Support for Advanced Modeling Tools

Current modeling tools typically force the modeler into expressing his model in a limited framework, and investigating the model's behavior with a limited set of query facilities. These tools are rarely extensible, i.e., they do not permit addition of new features. Tangram provides tools that permit 'declarative' specification of models supporting complex structuring of knowledge and deep interpretation as discussed earlier. We currently envision an object-oriented environment for developing these tools, supporting knowledge base management and arbitrary query processing. The environment will be extensible, permitting the addition of new kinds of models.

We have developed a methodology for building modeling tools based on Markov processes. A prototype system has been built based on this methodology [14]. In the system, users specify system components in an object-oriented framework. This level of specification is significantly higher than that provided by most modeling tools, which require input in the form of Markov chains, Petri nets, etc. Lower-level derivates (such as Markov chains) can be obtained from this specification when this is desired, or the specification can be simulated directly.

In the longer range we are developing methodologies for designing and implementing modeling tools that are applicable to computer systems research. These will include analytic, statistical, simulation as well as expert system-like 'conceptual' modeling. We are currently extending the work already done with Partial Order Programming [69].

2.2. Prerequisites on Implementation

Tangram is implemented primarily in Prolog. Prolog is an excellent starting point for developing a modeling environment for at least two reasons:

---

*Currently Tangram is being implemented on top of SICStus Prolog. SICStus Prolog is a portable Prolog environment developed in *C* at the Swedish Institute of Computer Science.

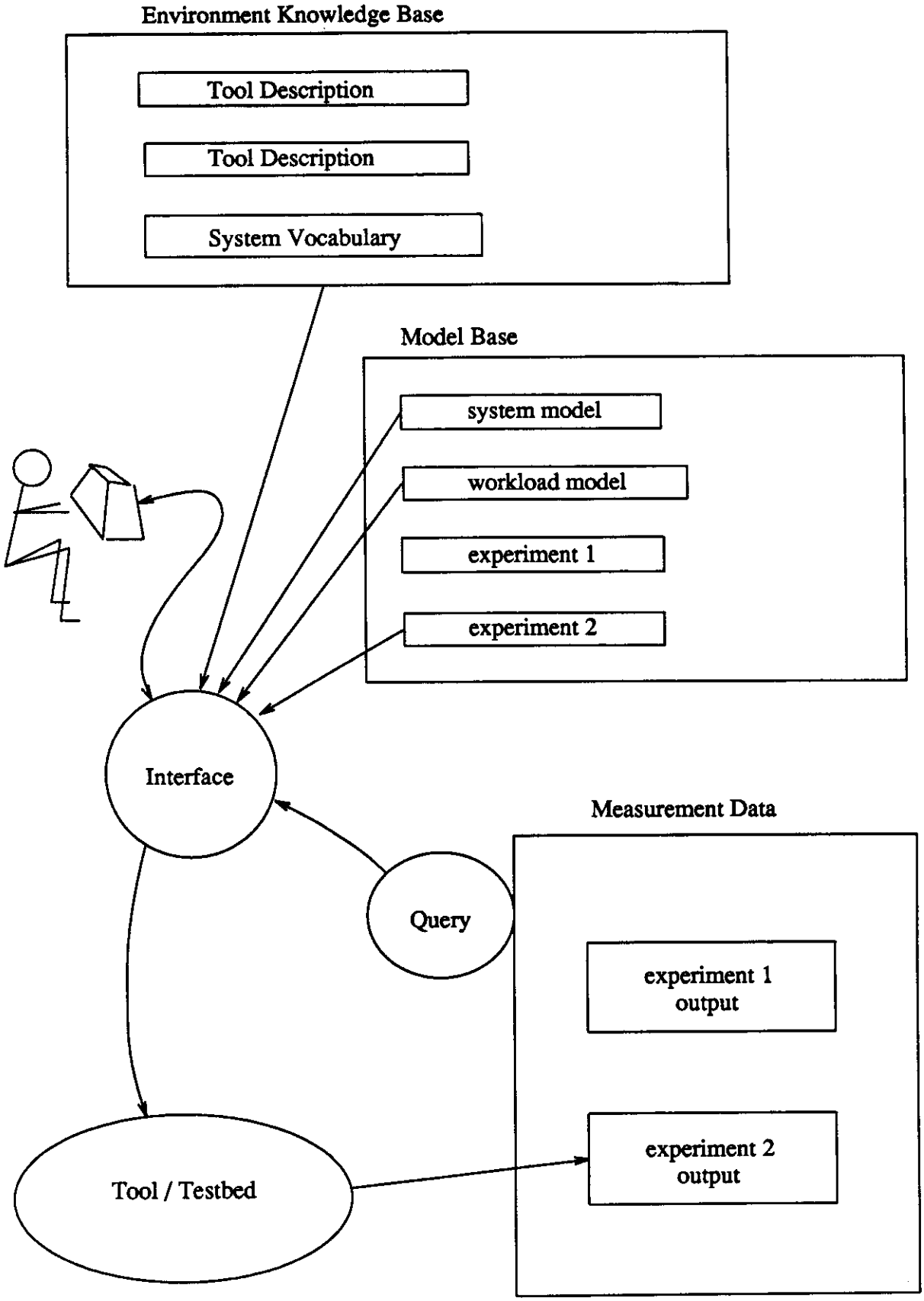


Figure 1. Tangram Modeling Environment Functions

(1) Prolog is unarguably the best candidate as a database/knowledge base language. It subsumes relational databases, supports complex structure in data through its terms and rules, and its first-order logic foundations are appropriate in many situations: unification and pattern matching, logical derivation and intensional query processing, backtracking and search, and most generally declarativeness. It easily supports increases in structure of models and 'expert system' techniques for interpretation of these models.

(2) Prolog is flexible. It is an outstanding vehicle for rapid prototyping, and permits easy access to existing systems that perform computationally intensive tasks efficiently.

To be effective, however, an environment based on Prolog must offer the features cited below.

2.2.1. Industrial Strength

As we pointed out earlier, increases in the quantity of data and in the complexity of interpretation require distributed computing/supercomputer techniques. Massive parallelism is needed to deal with the increased volume of information. Interactive display of model behavior is essential for effective modeling. An environment like Tangram incorporating these techniques will be successful only if it provides 'industrial strength' performance. Keys to success here are:

- Optimization
- Advanced data management technology
- High-performance interactive graphics
- Support at the operating systems level

2.2.2. Integration

A modeling environment must be able to combine many different systems of different types elegantly and efficiently. The many tools and testbeds for developing models that have taken man-centuries to develop should be accessible directly and conveniently from a single workstation. Prolog is excellent for representing and making inferences how these tools and testbeds should be accessed, but efficient access prohibits the use of *'glue job'* connections between them and Prolog. Database systems, for example, require stream access rather than the tuple-at-a-time access encouraged by Prolog. Keys to success here are:

- Modularization
- General ability to connect with diverse programs
- General knowledge representation of program functions
- Support for translation tools

2.2.3. Support for Evolution and Multiple Models

There are many ways to represent the same information. As models are refined over time they become more detailed, and focus on specific aspects of systems. Also, different models using different abstractions are necessary to represent complex systems accurately and efficiently.

Both evolution of models and different views of complex systems require different kinds of models, and hence different languages or *paradigms* for capturing different aspects of the real world.

Bobrow [16] criticizes Prolog on the grounds that there are many programming paradigms other than logic programming, and existing Prolog environments should, but do not yet, support them.

For modeling in particular, this criticism is of the essence. Paradigms can be low-level in nature, such as with parallel/distributed processing, object-oriented programming, or constraint satisfaction. They can also be more high-level or 'semantic' in nature, such as with extended queueing networks. Keys to success here are:

- Support for multiple modeling paradigms
- Representation of connections among paradigms

2.3. Partial Evaluation as an Implementation Strategy

The list of prerequisites above may seem somewhat imposing. It is not immediately apparent how they may all be achieved, or achieved with Prolog as a foundation.

Tangram uses partial evaluation as a performance-oriented mechanism for extending what paradigms are available in the Prolog environment. Software in the Tangram environment is written in one of two ways:

(1) As ordinary *Tangram Prolog* code. This code looks like 'standard' Edinburgh-style Prolog code, with the exception that a module system developed at UCLA is used, and many UCLA-specific predicates have been added.

(2) As code from an appropriate *paradigm*. A paradigm consists of both a language, and an interpreter for that language. In Tangram, there are many specialized paradigms, including:

- Parallel/Distributed processing
- Stream processing
- Object-oriented and functional programming
- Constraint satisfaction

Extension through addition of new paradigms is encouraged.

Currently paradigms in Tangram are implemented using a general partial evaluation scheme. With partial evaluation, the paradigm interpreter is used to translate paradigm language statements into Tangram Prolog code which can be optimized and subsequently executed. Loosely speaking, partial evaluation uses an interpreter as a 'macro' for expanding input statements into Prolog statements. As much evaluation of the 'macro' as possible is performed at expansion time, so partial evaluation can be thought of as a general optimization technique.

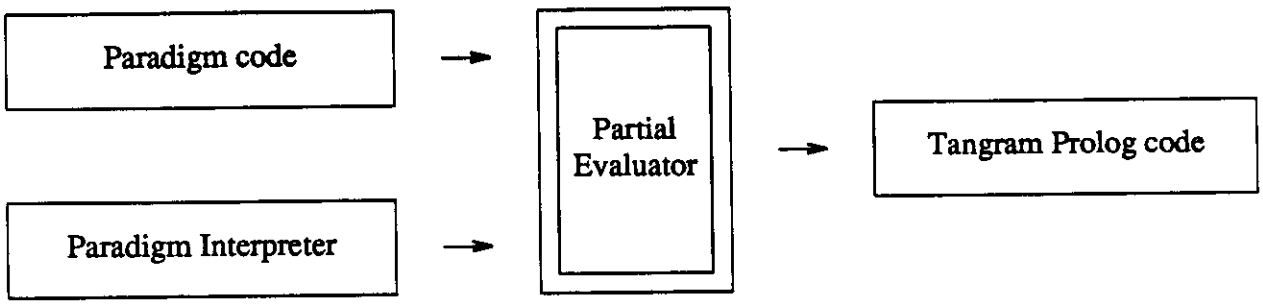


Figure 2. Partial Evaluation

Thus the interpreter for a paradigm can be used either for execution or compilation: paradigm code is either interpreted directly in Prolog, or partially evaluated into Prolog for subsequent execution. Recent work on partial evaluation is summarized in [87], along with issues in using it for interpreting parallel programming languages. Speedup factors of 40 have been reported from the use of partial evaluation instead of direct interpretation.

Partial evaluation can be performed multiple times, of course. That is, paradigms can be used hierarchically or accumulatively. For example, we can write the interpreter for a constraint satisfaction paradigm in the language of the object-oriented paradigm. In this case multiple levels of partial evaluation are needed.

2.4. Project Overview

The individual tasks of the Tangram project are summarized in the following sections:

- Computer Systems Modeling
- Constraint Processing
- Stream Data Processing
- Industrial Strength Prolog

We have selected computer systems modeling as an initial modeling domain, as this is a domain in which we are expert. Constraint processing is necessary for the specification of models, and stream processing is necessary for the evaluation of model output. The industrial strength Prolog extends Prolog to be an effective language for 'real' applications, as opposed to the 'toy' applications for which it has been used in the past.

3. Computer Systems Modeling Applications

The only good environments are those that are used by their developers. To help guide direction of the Tangram environment, we have selected the general application area of knowledge-based modeling of complex computer systems. Within this area, Tangram is focussed on two projects:

- Complex Computer System Modeling
- Distributed DBMS Performance Modeling

3.1. Complex Computer System Modeling (Steven Berson, Bill Cheng, Dick Muntz)

This project focuses on developing methodologies for computer system modeling. The main objectives of this project are the following:

(1) To construct an environment that integrates various analysis tools so that computer system modeling experts can access them through a common interface. The environment should also be capable of giving expert advice on how a model of a computer system can be analyzed, and why. Such an environment would also allow integration of new tools.

(2) To facilitate the construction of modeling packages that are tailored to particular application domains for non-experts.

Computer system modeling falls into general domains such as performance analysis, availability analysis, and reliability analysis. In order to analyze a computer system, a model in a certain 'domain' has to be constructed. For each kind of domain, there exist many tools that can analyze models in the domain. However, tools are usually applicable only to a limited range of problem areas, and they will perform with different efficiencies in different problem areas. One of the goals of our environment is to manage the complex relationships between various domains and tools. Users of our environment should be able to describe their models in the form that is natural for their application domain, and the system should be capable of translating that description to the form required by the appropriate analysis tool. A prototype for modeling computer systems based on Markov processes [14] is now running, and is being extended.

Sometimes, the exact analysis of a complex computer system model is infeasible due to the size of the model; it is then necessary to perform approximate analysis of the model. There are different techniques for the approximate analysis of computer systems, and each of them works well under different conditions. With knowledge based techniques, our environment can assist users in using these approximate analysis techniques.

Currently, we are focusing on Markov processes and queueing networks. Continuing work on the current prototype includes:

(1) Graphical interfaces for entering, editing, and displaying modeling descriptions.

(2) Query facilities based on the high-level model.

(3) Model debugging/consistency checking.

(4) Extensions to allow the modeler to specify approximation analysis techniques.

(5) Heuristic interpretation of the analysis results.

(6) Model 'optimization', e.g. providing automatic state space reduction where possible.

We also wish to explore the problems of dealing with real computer systems. This requires developing methods for:

(1) Describing real computer systems: both their structure and the measurements that are available.

(2) Query and display facilities based on the system description.

(3) Correlation of analytic and/or simulation models with a conceptual model of the real computer system. This would provide the basis for model validation.

3.2. Distributed DBMS Performance Modeling (Ron Hu, Dick Muntz)

A performance measurement environment is being developed for the distributed data management system manufactured by Teradata, Inc. This system provides a rich environment in which to study the application of our modeling methodology. The system itself has a complex structure both in terms of the hardware and software architecture. The performance issues include: configuration planning, evaluation of hardware/software design alternatives, evaluation of query optimization strategies, logical and physical database design, and regression testing of the performance of new software releases. We plan to investigate the application of our modeling environment to this set of problems. The study will concentrate on the database aspects as well as the unique features of the architecture which we expect to provide new insights. The system described in [47] is a simple approximation to what we have in mind, and resembles the Tangram environment illustrated in Figure 1.

4. Constraint Processing

Until recently, models often enforced a strong distinction between the data of the model (parameters, initial values, etc.) and the model itself. Modeling was done as a very rigid kind of programming, with a strong distinction between programs and data.

This distinction has begun to blur. Largely a result of the influence of AI tools in software, new modeling tools have developed in which one can treat parts of the model program as parameters, or define parameters with additional models. Basically the idea is that one can specify behavior information (rules) in the same way one specifies descriptive information (facts).

In its essence, this idea amounts to what is commonly called *declarativeness*. That is, one should be able to describe real-world systems by declaring the *constraints* on their behavior, rather than by giving programs or other procedural descriptions that somehow fulfill these constraints. Not only is it easier to specify constraints declaratively, it is also easier to validate the correctness of a model that is specified declaratively. It is notoriously difficult to verify that a program satisfies the constraints that are expected of a model.

An instructive example of the importance of declarativeness comes from today's database systems. During the 1970s database systems moved away from 'procedural' query languages to declarative languages in which users could specify the results they wished, without having to specify how the results should be obtained. Today, fourth-generation languages (4GLs) are accepted as important means for specifying queries and data processing requirements. Constraint processing can be viewed as the continuation of the evolutionary development from DBMS to more powerful information processing systems.

Constraint processing is a modeling paradigm in which models may be developed declaratively. It is the modeling paradigm of choice within the Tangram project. Constraint processing is also used in other ways in Tangram, both in program analysis ('abstract interpretation') for optimization of programs, and in graphical display of information.

4.1. Partial Order Programming

A good deal of the Tangram work in constraint processing is based on partial order programming [69]. Partial order programming is a new paradigm developed by Parker at UCLA in which statements are constraints over partial orders. In this paradigm a problem has the form

minimize	u
subject to	$u_1 \sqsupseteq v_1$
	$u_2 \sqsupseteq v_2$
	$\cdots$

where u is the *goal*, and $u_1 \sqsupseteq v_1, u_2 \sqsupseteq v_2, \cdots$ is a collection of constraints called the *program*. A solution of the problem is a minimal value for u determined by values for u_1, v_1, etc. satisfying the constraints. The domain of values here is a *partial order*, a domain D with ordering relation $\sqsupseteq$.

The partial order programming paradigm has interesting properties:

(1) It generalizes mathematical programming, dynamic programming, and computer programming paradigms (logic, functional, and others) cleanly, and offers a foundation both for studying and combining paradigms.

(2) It takes thorough advantage of known results for continuous functions on complete partial orders, when the constraints involve expressions using only continuous and monotone operators. These programs have an elegant semantics coinciding with recent results on the relaxation solution method for constraint problems.

(3) It provides a framework that may be effective in modeling complex systems, and in knowledge representation for cognitive computation problems.

Recently we have applied partial order programming in the formalization of *directed logic programs* [70], logic programs which have specified input and output arguments. Such a formalization is important in defining the semantics of stream processing, and it unifies a number of diverse knowledge representation primitives as well.

4.2. Knowledge Representation (Stott Parker)

Modern knowledge representation systems tend to be frame-based systems or production systems with inheritance. They are largely ad hoc and support only certain kinds of shallow models [67]. The knowledge representation system planned for Tangram is formally founded on partial order programming, and will provide the following useful features:

Common systems of inference (syllogisms, parts, roles) [5]
Constraint solution mechanisms, including propagation and relaxation
Naming and Events
Semantic Unification
Contexts and Modalities
Meta-Level Capability & Planning
Schemata/Episodes/Definitions/Scripts/Prototypes/Defaults/Situations

These features should prove useful in the representation of both computer systems and of software packages used to model these systems. See [68].

We are developing a system for automated simple linear regression modeling. Certainly linear regression is the statistical test that is most useful in analyzing data. Our system is similar to the REX system described in [37], but iteratively refines a Box and Cox model, taking into consideration bias, normal distribution of residuals, and so forth. Issues in developing the system include how to represent functionality knowledge of the regression analysis tools available, and control knowledge for obtaining the desired transforms on the data from the regression tools. Currently the *S* statistical analysis package is being used.

A major issue in the design of the system is how much emphasis to put on interaction with the user about solution strategy. While statisticians prefer to explore their data and make decisions about transformations themselves, less knowledgeable users may prefer to have the system operate without any interaction in analysis. The issue is the degree to which statistical strategy in data exploration and analysis can be automated. Effective use of statistical analysis requires three kinds of expertise:

- Data expertise (general knowledge of what is in the data)
- System expertise (knowledge of what generated the data)
- Statistical expertise (knowledge of what the analysis means)

A modeler will have all three kinds of expertise only rarely. Statistical expertise helps avoid over- and under-interpreting data, and appears to be the most inviting to automation.

4.3. Constraint Satisfaction (Richard Huntsinger, Stott Parker)

The goal of the constraint satisfaction component of the project is the development of a powerful theoretical formalism which will provide semantics for a large class of model representations, and a correspondingly powerful constraint satisfaction system implementation [43].

Partial order programming [69] serves as an initial theoretical formalism. It provides a model-theoretic semantics and a procedural fixpoint-based semantics for model representations. These model representations are composed of partial ordering relations between variables and monotonic functions on variables. That is, models are represented as collections of constraints of the form $x \leq f(y)$, where x is a variable, $\leq$ is a partial order, f is a monotonic function, and y is a vector of variables.

This class of model representations corresponds to a subset of that on which relaxation can be successfully applied. Indeed, in practice, constraint satisfaction system implementations typically employ relaxation, and therefore operate only on restricted classes of all expressible model representations.

Partial order programming is currently being extended to permit some non-monotonic functions on variables, resulting in semantics for a larger class of model representations. This class corresponds to that on which a generalization of relaxation can be successfully applied. Equivalently, it is a class for which each model representation can be transformed to a new representation on which relaxation can be successfully applied. For example, the model representation $x \leq f(y)$, where f is a non-monotonic function, is provided semantics if $x \leq f(y) \Rightarrow x \leq t \circ f(y)$, t is an invertible transformation, and $t \circ f$ is a monotonic function [44].

The theorems comprising this extension to partial order programming suggest several constraint satisfaction algorithms. A constraint satisfaction system implementation is being realized as a tool which facilitates experimentation with some of these algorithms. Specifically, it employs a general parameterized algorithm; instantiated instances are various specific algorithms, including relaxation.

Parameters of special interest include

- the strategy for organizing functional dependencies,
- the strategy for detecting transformable constraints,
- the strategy for applying transformations, and
- the strategy for decomposing sets of constraints into approximating sets of constraints.

Some testbed model representations for the constraint satisfaction system implementation

include computer network configurations, solid-body animation sequences, and music compositions.

4.4. Abstract Interpretation (Arman Bostani, Stott Parker)

A lot of research has been done on the derivation and proof of properties of Prolog programs. This work has been fueled by the desire to close the performance gap that exists between imperative and logic programming languages running on conventional hardware. Several reasons have been cited for this performance gap. Firstly, procedures in logic programs are quite versatile and the code can be general enough to be used in many different ways. Secondly, in imperative programs, memory usage is explicitly controlled by the programmer; this not only saves memory, but also time since it avoids copying whole data structures when only slight modifications are made. Finally, type information that is available in an imperative language allow a better implementation of the programmers' data structures.

In the attempt to close the performance gap many researchers have implemented various automated inference systems which can generate polymorphic typing, mode declarations, determinancy information, etc. To formalize this work on Prolog programs, researchers have adapted the ideas of Cousot and Cousot [30] on the abstract interpretation of imperative programs to the field of logic programming. Since a characterization of the exact behavior of a program is in most cases computationally intractable, we are forced to interpret our programs abstractly. That is, the program is thought of as executing in an abstract domain where less information about the data items is accounted for. Results of the computation in this abstract domain then reflect the properties of programs operating in the exact model.

Abstract interpretation of Prolog programs has been used in several applications:

- Automatic inference of polymorphic types [25].
- Automatic mode inference of Prolog predicates. Debray and Warren [34] describe a data flow analysis which is more powerful than previous approaches which solely relied on purely syntactic information.
- Detection of determinancy. Mellish [59] discusses a method for detecting 'determinate' Prolog code (i.e., finding those predicates that never return more than one solution).
- Global optimization of Prolog programs. A general theoretical framework is provided by Bruynooghe [21] with which an optimizing Prolog compiler may use abstract interpretation for efficient code generation.

The main focus of our research has been to devise a system that will be able to derive various properties of prolog programs such as mode, type, aliasing and predicate success information. Previous research has shown, however, that purely syntactic analysis of programs is insufficient for the derivation of truly useful information. Thus, from the very beginning, our goal has been to create a simple formalism under which we can represent both syntactic and semantic information in a unified manner.

To capture the semantics of Prolog programs, we use *'inexact success models'*. These success models provide information on the function of a predicate by classifying its actions based on the

types of arguments with which the predicate is invoked. Two elements determine the 'function' of a predicate:

- Instantiation of variables
- Success of the predicate

Information on the instantiation of variables can be used in the derivation of mode characteristics of the predicates. Also, the type classification above is similar to conventional polymorphic type inference. Determination of the success of predicates, however, affects the derivation of mode and type information. The idea of using success information in abstract interpretation is relatively new [21] and unexplored.

Our success models provide bounds on the number of choice points created by a predicate, and whether or not it will 'fault'*. Thus, our system will be able to detect possible fault conditions before the execution of a program. The task of determinancy detection [59] is also easily performed with this system (e.g., if a predicate generates only 0 or 1 choice points, it is determinate).

The design phase of the system for the inference of these success models is almost finished. Our next step will be to implement the success model derivation system. The system can then be used with applications such as optimizing compiler, interactive debugging tool, performance analysis, etc.

4.5. Graphics (Ted Kim, Stott Parker)

Many modeling applications require graphics. One approach would be to use a declarative graphics interface [41] based on formal picture description grammars. The formal framework offers some attractive parallels to proof systems. This paradigm offers the capability to describe, compose and generate pictures as well as 'prove' or recognize pictures. While we are pursuing some research in this area, we feel that this graphics description is too low-level for general use.

Our main effort is to provide a higher level constraint-based graphics interface [19]. The constraint-based orientation provides a declarative style of programming graphics. Our system uses the X Window system [75] to provide network graphics and windowing capabilities. X also offers the advantage of being a de facto standard. At the lowest level, our design will provide support for graphics by making the X library functions accessible in Prolog without requiring extensive changes to Prolog (e.g., no asynchronous operation).

On top of this layer, we provide a constraint-based graphics toolbox. This toolbox provides common primitives for design of graphical applications as well as support of constraint systems. Speed is very important to graphical applications. The challenge here is to provide general enough constraint solvers that are also fast. Towards this end, our design includes a notion of graphical state. The system responds to perturbations made to a current constraint solution, while attempting to resolve as few constraints as possible. The state from the previous solution is used as much as possible in forming the new solution. This allows local changes to be quickly solved.

---

*In most Prolog systems a program may terminate unexpectedly due to a fault. A fault may be caused, for example, by passing arguments of incorrect type to a system predicate.

Typically, the toolbox would be used to build graphical displays of modeling solutions. These displays would change automatically in response to perturbations of modeling data. This provides the basis for interactive model management, involving storage, retrieval, query, display, and editing of models [50, 58, 82]. Animation can also be provided in this fashion.

Continuity is also important in graphics, especially in animation. Changes in animation should be smooth and continuous. The graphical state helps here, but is not sufficient. Out of the possible space of solutions for the constraints, we would like to pick a 'good' solution. For the animation problem, 'closeness' of a new solution to an old one is the issue. For the automated display problem [56], the criteria could be such things as expressiveness or color choice. More generally, we can cast the problem as optimization. To address these issues, we are planning to include an optimization mechanism for our graphics constraint solver.

Graphics is also important in the presentation of refinement graphs of data flows in concurrent execution models. With the concurrent execution model, user annotations of programs can be used by the system to generate diagrams of refined data flows implied by the annotations. These diagrams allow the user to spot potential problems with his annotations.

5. Stream Data Processing

In this section we describe projects concerning the *Tangram Stream Processor (TSP)* [71], a stream data management system. It is an extensible system based on a functional sublanguage of Prolog that provides a programmable stream processing capability with a number of interesting characteristics.

5.1. Streams and Data Processing

Relational databases are founded on set theory: all relations are viewed as sets of tuples. Many developments have encouraged generalization of this model to one of *ordered sets*. For example, ordered data can be processed much more efficiently than unordered data. In fact, many standard query evaluation techniques are described in terms of operators (filters, mappings, actors) acting on ordered sequences of tuples. Moreover, temporal query processing seems to necessitate some kind of ordering if important kinds of queries are to be efficiently answerable. Also of course, the order of the tuples in relations is important in presentation of the relations to users.

In many important situations, then, it is advantageous to generalize the set foundation of the relational data model to an ordered set model. We call ordered sets *streams*. Stream-oriented processing is certainly not a new subject, although it has only recently come into its own right as a programming paradigm.

An area where streams are inherent to query processing is for temporal data, data with explicit or implicit time ordering. The analysis of streams has been done for many years as 'time series analysis'. Recently, the subject of time in databases has gotten increasing attention as more applications requiring temporal reasoning have been uncovered [3, 4, 12, 18, 27], and many interesting systems handling temporal queries in novel ways have been developed [2, 33, 49, 51, 52, 55, 77, 84, 85].

Previous research has concentrated either on database processing, or on representational issues and generality of modeling. Two important database systems include:

(1) TQuel [84], a relational query language with embedded time primitives, is an extension of Quel, both syntactically and semantically. TQuel is essentially a relational query language, resting on the relational model.

(2) The Time Sequence approach of Shoshani [77, 79] characterizes properties of temporal data and temporal operators without restriction to the relational model. Data are organized into *Time Sequence Collections (TSCs)*, which can take both relational and stream-like representations. Five basic operators provide an algebra working on TSCs.

These systems emphasize performance and complete handling of a well-defined set of query operators. Other researchers in temporal query processing have worked at more complex modeling, combining work on temporal logic and existing representational systems to define new approaches. Sadri [73] reviews three general recent approaches to temporal reasoning:

(1) The 'event calculus' of Kowalski [49] is an approach for reasoning about events and time within a logic programming framework.

(2) Allen's approach [3, 4] is similar to the event calculus, defining a set of binary predicates giving basic relationships among time intervals (whether they overlap, one precedes the other, etc.).

(3) Lee, Coelho and Cotta [52] present a temporal system for representing and reasoning about time-dependent information and events, specifically for business database applications.

In these approaches it is peculiar that stream processing has not been emphasized more heavily for temporal query processing, as well as for basic relational query processing. Tangram's stream processing approach permits it to handle queries definable under each of the systems listed here.

5.2. Streams and Parallel Processing

Concurrent, object-oriented, functional, and logic programming paradigms all intersect elegantly in the abstraction of streams: the general architecture of cooperating parallel actors transforming streams of events has found its way into many programming systems that have been proposed in the past few years. For example, many parallel logic programming systems have been developed essentially as stream processing systems. Typically, these systems fall into one of several camps:

(1) They resemble PARLOG [26, 39] and the other 'committed choice' parallel programming systems [78] (Concurrent Prolog, GHC, etc.).

(2) They introduce '*parallel and*' or '*parallel or*' operators into ordinary Prolog [53].

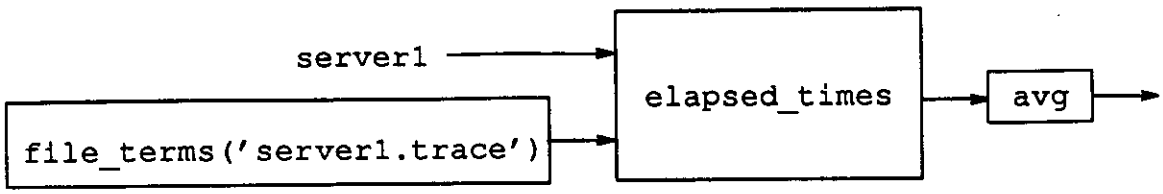
(3) They are extended Prolog systems that introduce streams by adding functional programming constructs [32, 48, 54, 62, 86]. The thrust of this introduction is to make Prolog more like either Lisp or Smalltalk or both.

TSP has drawn on the designs of a number of previous systems which have included stream concepts. These include FAD [6], various dataflow database systems [7, 8, 9, 15, 38], and LDL [11, 89].

After some experience with the tuple-at-a-time and whole-query-at-a-time (embedded query language) Prolog/DBMS interfaces that have been developed to date, we feel a better way to integrate Prolog and databases is through streams [65]. Only minor extensions to Prolog are sufficient to provide fairly efficient stream processing [72]. A stream interface offers an effective medium between these two alternatives, uniformly integrating bulk operations at the DBMS end with incremental evaluation at the Prolog end. Prolog stream processing avoids backtracking through a database, using efficient iterative (tail recursive) processing instead. It is a natural approach for applications like analysis of modeling data.

5.3. The Tangram Stream Processor

The Tangram Stream Processor is founded on the abstraction of transducers. A transducer is a mapping from some number of input streams to one or more output streams. Thus, a transducer may be viewed as an automaton. However, a transducer can take parameters, and as such need not have only a finite number of states. Thus, it is better initially to view transducers as mappings instead, and diagrams of transducer networks resemble dataflow diagrams:



Transducers are the basic building blocks of TSP, and are maintained in an (extensible) library. Since arbitrary transducers are permitted, the expressive power of TSP is equivalent to that of any general programming language. Consequently, the stream-based transducer model is more general than many previous approaches: it is capable of handling traditional database queries and non-traditional queries that reason about time in event databases.

TSP has several further unique aspects:

(1) TSP permits operation on general stream structures, including for example both lists and array models of data. It supports definition of and parallel evaluation of operators on these stream structures, including the operator families of the APL programming language, NIAL [57], and the Nested Array model of data upon which both are based [60, 61]. This includes the ability to define *higher-order operators* on streams, such as aggregate operators (*min*, *max*, *sum*, etc.), APL's reduction operator, LISP's *maplist*, etc. In addition, it permits us to define many useful statistical operators on streams, as in the *S* data analysis system [10].

(2) TSP permits operation on *infinite streams*. A stream may represent a non-terminating sequence of values. This is not permitted, for example, by APL.

(3) TSP permits both *lazy and eager evaluation* of streams. Lazy evaluation permits efficient evaluation of some kinds of queries.

(4) TSP transducers are naturally implemented as concurrent processes. These transducers provide opportunities to place natural boundaries on parallelism, a feature not enjoyed by some parallel Prolog systems.

The resulting system may be used for '*database-flow*' processing, a combination of 'dataflow' and database processing, as well as general feature extraction and data reduction operations that fit in a pipeline structure.

Execution of queries in TSP is quite efficient, in the common situation that the input streams are sorted properly. In fact, TSP query processing can be considerably more efficient than that in relational DBMS. For many TSP queries a single scan of the input streams is sufficient, requiring linear time and constant space, while relational DBMS approaches require significantly more resources. Also, TSP can handle kinds of queries not easily handled by relational query processing systems, including the following:

1. Sliding window queries [77]
2. Event calculus queries [49]
3. Pattern matching queries
4. Abstracting state information from event data
5. Reasoning about time.

As an example of a query that reasons about time, consider asking about what investment strategy would have been optimal over a given period of stock market history. This requires innovative accumulation of dividends, interest rates and rules for compounding interest, days which are holidays, and many other important details. These *'hindsight queries'* illustrate the potential of stream processing in database analysis.

5.4. Log(F)

The Tangram Stream Processor is based on Log(F), an integration of Prolog with a functional language called F*, developed by Sanjai Narain at UCLA [63, 64]. Log(F) is the integration with Prolog of a functional language in which one programs using rewrite rules. This section reviews the major aspects of Log(F), and describes its advantages for stream processing.

F* is a rewrite rule language. In F*, all statements are rules of the form

LHS => *RHS*

where *LHS* and *RHS* are structures (actually Prolog terms) satisfying certain modest restrictions summarized below.

A single example shows the power and flexibility of F*. Consider the following two rules, defining how lists may be appended:

```
append([],W) => W.
append([U|V],W) => [U|append(V,W)].
```

Like the Prolog rules for appending lists, this concise description provides all that is necessary.

Log(F) is the integration of F* with Prolog. In Log(F), F* rules are compiled to Prolog clauses. The compilation process is straightforward. For example, the two rules above are translated (partially evaluated) into something functionally equivalent to the following Prolog code:

```
reduce(append(A,B),C) :- reduce(A,[]), reduce(B,C).
reduce(append(A,B),C) :- reduce(A,[D|E]), reduce([D|append(E,B)],C).
reduce( [], [] ).
reduce( [X|Y], [X|Y] ).
```

An important feature of F* and Log(F) is the capability for *lazy evaluation.* With the rules above, the goal

```
?- reduce( append([1,2,3],[4,5,6]), X ).
```

yields the result

```
X = [1|append([2,3],[4,5,6])].
```

That is, in one `reduce` step, only the head of the resulting appended list is computed. The tail, `append([2,3],[4,5,6])`, can then be further reduced if this is necessary. Demand-driven computation like this is referred to as lazy evaluation or delayed evaluation, and is basic to stream processing [1].

Log(F) is a superior formalism for stream processing, and thus for database query processing. From the example above, it is clear that the rules have a functional flavor. Stream operators are easily expressed using recursive functional programs. The syntax is convenient, and can be considered a useful query language in its own right.

It turns out furthermore that Log(F) has a formal foundation that captures important aspects of stream processing:

(1) Determinate (non-backtracking) code is easily detected through syntactic tests only. This avoids the overhead of 'distributed backtracking' incurred by some parallel logic programming systems.

(2) Log(F) assumes that stream values are *ground terms*, i.e., Prolog terms without variables. Again this avoids problems encountered by other parallel Prolog systems which must attempt to provide consistency of bindings to variables used by processes on opposing ends of streams.

These features of Log(F) make it a nicely-limited sublanguage in which to write high-powered programs for stream processing and other performance-critical tasks. Special-purpose compilers can be developed for this sublanguage that produce highly-optimized code.

We must stress strongly that Log(F) is an *extension* of Prolog. The Log(F) code shown above runs as shown. The issues here are not so much language design issues as in developing compilers for Log(F) that generate fast code. Where speed is not critical, the full power of Prolog is available to its users now, in a stable development environment.

5.5. Stream Transducers and Log(F) (Lewis Chau, Dick Muntz, Stott Parker)

In [71] we show how transducers can be written in Log(F) to solve both traditional and very novel query processing problems. It is easy to develop significant stream transducers with compact sets of rewrite rules. We currently have an implementation of Log(F) in Prolog that performs all standard database query processing primitives, and many nonstandard ones as well. For example, transducers can be developed that manipulate streams of 'elapsed time' data, or streams of 'service time' data. We can define transducers as follows:

```
% Elapsed times for first-come, first-served discipline
fcfs_e([],_) => [].
fcfs_e([a(T)|L],State) => fcfs_e(L,append(State,[T])).
fcfs_e([d(T)|L],[T0|S]) => [T-T0|fcfs_e(L,S)].

% Elapsed times for last-come, first-served discipline
fcfs_e([],_) => [].
lcfs_e([],_) => [].
lcfs_e([a(T)|L],State) => lcfs_e(L,[T|State]).
lcfs_e([d(T)|L],[T0|S]) => [T-T0|lcfs_e(L,S)].
```

```
% Service times for first-come, first-served discipline
fcfs_s([],_) => [].
fcfs_s([a(T)|L],(N,T0)) => if(N==0, [fcfs_s(L,(1,T))],
                                    [fcfs_s(L,(N+1,T0))]
                              ).
fcfs_s([d(T)|L],(N,T0)) => [T-T0|fcfs_s(L,(N-1,T))].

% Service times for last-come, first-served discipline
lcfs_s([],_) => [].
lcfs_s([a(T)|L],[]) => lcfs_s(L,[(T,0)]).
lcfs_s([a(T)|L],[(T0,T1)|S]) => lcfs_s(L,[(T,0),(_,T-T0+T1)|S]).
lcfs_s([d(T)|L],[(T0,T1),(T2,T3)|S]) => [T-T0+T1|lcfs_s(L,[(T,T3)|S])].
lcfs_s([d(T)|L],[(T0,T1)]) => [T-T0+T1|lcfs_s(L,[])].
```

These transducers may appear a little forbidding. We can however make these available in a simpler and more natural form by introducing 'higher level' transducers:

```
elapsed_times(Server,S) => policy_elapsed_times(type(Server),S).

policy_elapsed_times( [fcfs], S) => fcfs_e(S,[]).
policy_elapsed_times( [lcfs], S) => lcfs_e(S,[]).

service_times(Server,S) => policy_service_times(type(Server),S).

policy_service_times( [fcfs], S) => fcfs_s(S,(0,_)).
policy_service_times( [lcfs], S) => lcfs_s(S,[]).
```

Interesting statistics (e.g. mean, standard deviation, etc) can then be calculated from this output stream by applying further aggregate operators. For example, the average elapsed times and maximum service times at Server 1 can be obtained with:

```
avg( elapsed_times(server1, file_terms('server1.trace')) )
max( service_times(server1, file_terms('server1.trace')) ).
```

Here `file_terms('server1.trace')` produces a stream of arrival and departure terms

```
a( ArrivalTime )
d( DepartureTime )
```

that are tallied by the transducers defined here. Also, `type(server1)` reduces either to `[fcfs]` or to `[lcfs]`, according to the type of the server.

5.6. Pattern Matching against Streams (Lewis Chau, Stott Parker)

In [71], we illustrate how Log(F) makes a powerful language for expressing transductions of streams. In this section we show how, specifying patterns with *grammars*, it also makes an expressive language for pattern matching against streams.

In order to match patterns against streams, the approach taken in Tangram is to let users specify patterns with *grammars*, which are compiled into efficient transducers. Moreover, users can express their patterns using a library of grammars. For example, regular expressions and, more generally, path expressions, can be easily defined with grammar rules:

```
(X+) => X.
(X+) => (X,(X+)).
(X*) => [].
(X*) => (X,(X*)).
(X;Y) => X.
(X;Y) => Y.
(X,Y) => append(X,Y).
skipto(X) => X.
skipto(X) => ([_],skipto(X)).
```

These Log(F) rules behave just like the context free grammars they resemble.

Pattern matching is signaled explicitly with the match transducer, which takes as its first argument a functional grammar term describing the starting symbol(s) of some grammars used for the match, and as its second argument a Log(F) term that produces a stream. For example,

```
match(([net_failure]+,[cpu_failure]),file_terms('experiment1.output')).
```

matches the pattern 'one or more copies of net_failure followed by a cpu_failure' to the stream of events in the file `'experiment1.output'` into a stream of event types.

The rules for pattern matching are very simple. The basic definition is as follows:

```
match([],S) => S.
match([X|L],[X|S]) => match(L,S).
```

The result of matching a pattern against a stream is what is left of the stream after pattern matching completes, i.e., the remainder of the stream that is not matched by the pattern. Simultaneously, arguments of the nonterminals are bound to values resulting from parsing the input stream. With this definition of `match`, we can immediately define grammars using rewrite rules. We call a collection of these rules a *functional grammar.*

In the section above we showed how transducers can manipulate streams of 'elapsed time' data. It is also possible to specify it with a functional grammar:

```
fcfs_e(Result) => fcfs_e([],[],Result).
fcfs_e(_,Result,Result) => [end_of_file].
fcfs_e(State,Current,Result) => [a(T)],
        fcfs_e(append(State,[T]),Current,Result).
fcfs_e([T0|S],Current,Result) => [d(T)],
        {T1 is T-T0}, fcfs_e(S,[T1|Current],Result).
```

The main issue here is finding a way to compile functional grammars and *match* into efficient transducers, then we can define efficient classes of functional grammars. Not surprisingly, deterministic tail-recursive grammar rules which do not construct large structures for their state can be compiled to efficient transducers. These grammars are much like classical right linear regular grammars, and like DCGs that are actually written in practice. See [24].

5.7. Distributed/Parallel Processing (Brian Livezey, Dick Muntz)

Log(F) benefits from two aspects of distributed processing. First, computation can be performed in parallel. There are many opportunities for computation concurrency in a stream-based programming environment. The execution behavior of sequential Log(F) is analogous to a pipeline in which only one stage is active at any given time. By placing different stages of the pipeline on different processors and replacing lazy evaluation by eager evaluation and stream flow, we can achieve a state in which all stages of the pipeline are active simultaneously. Second, in addition to this 'pipeline concurrency', we can achieve other forms of parallelism. AND-parallelism is achieved by producing all input streams to a given transducer simultaneously (i.e. concurrent reduction of all arguments to a function). OR-parallelism results from having different portions of the same stream being produced simultaneously on different processors.

The performance of queries on distributed databases is greatly affected by the relative locations of the data and the processes that operate on that data. Many techniques exist for optimizing queries on distributed databases. Distributed Log(F) provides the ability to exploit these techniques. By providing the ability to subdivide queries and specify the processor upon which each portion is to execute, Log(F) allows programmers to express efficient distributed queries.

Many stream-based concurrent programming systems [78] are designed for shared-memory multiprocessors and therefore attempt to exploit a very fine granularity of concurrency. In non-shared memory environments, where communication and processes are not cheap, any performance gained through concurrency will be lost in overhead. For such environments, it is necessary to allow a much coarser granularity of concurrency.

Ideally, in non-shared memory environments, the compiler should be able to recognize potential concurrency, balance it against the overhead, and decide how to distribute a given program. While such compilers exist for distributed database queries, no such compilers exist for arbitrary distributed programs. Therefore, we must initially require that the programmer specify how a given program is to be distributed. However, the programmer should not have to define the distribution of the program when designing the logic. Instead, the programmer should write the entire program first and then specify concurrency without changing the program's semantics.

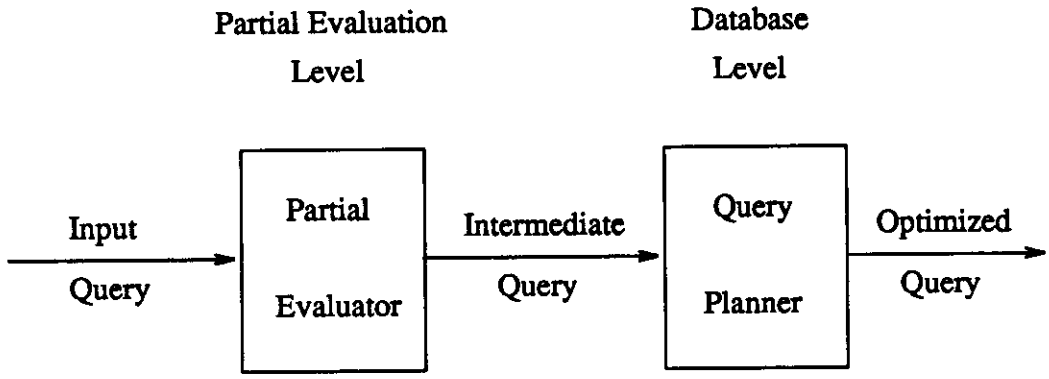
We intend to provide two interfaces to distributed Log(F) to facilitate the composition of distributed programs. First, we provide simple annotations which allow the programmer to indicate transducers which are to reside on remote processors. Second, we intend to provide a graphics interface to Log(F). Programmers will select transducer icons from a toolbox and connect them together to form larger transducers. Process boundaries will be indicated by surrounding portions of the resulting graph with boxes to indicate that that portion is to be run on one processor. Boxes will be annotated to indicate which processor they should be run on.

Ultimately, we will not require the programmer to specify how to distribute his program; we will only require that the programmer assign weights to each of the elementary transducers used in his program. Weights will be a function of the computational expense for each input element as well as the ratio of input elements to output elements. Weights of larger transducers will be determined by appropriately combining the weights of the elementary transducers that compose them. These weights will be used by the compiler to determine how best to distribute the program.

5.8. Program/Query Optimization (Lewis Chau, Cliff Leung, Dick Muntz, Stott Parker)

Partial evaluation is a special kind of program transformation for the purpose of optimization. This optimization is accomplished mainly via instantiation of parameters of a program by propagating values for top-level formal arguments through the program (execution of the unification at compile time), and reduction of the number of logical inferences by opening calls. It is similar in many ways to macro expansion.

A query that is a Prolog goal can be executed in many different ways. *Two-level optimization* is introduced as a means of finding a way to execute the query that is computationally as efficient as possible. *First-level optimization* applies partial evaluation as an alternative to compiling queries. With this approach, the partial evaluation system expands an input Prolog query, generating a conjunction of calls to the extensional database (EDB) intermixed with calls to built-in predicates. Redundant goals can be eliminated to some extent at this level. The partial evaluation system then transmits conjunctions of calls which can be the subject of further optimization by a database system. *Second-level optimization* (query planning) then optimizes a query at the database level. Its purpose is to analyze and improve queries based on straightforward information about the Prolog program underlying the query and the EDB itself. See [23].



Two-level Query Optimization

We are interested in applying partial evaluation to query optimization. A knowledge base is composed of a set of rules and ground facts. We can treat the set of rules simply as a logic program and the set of ground facts as a conventional relational database. Answering a query is equivalent to partially evaluating the query by an interpreter (partial evaluator), and executing the resulting conjunction of calls to the database. Optimization can be applied both to the partial evaluation level and to the database level. Currently, we have implemented a partial evaluator for full Prolog programs.

The output of partial evaluation is a conjunction of calls to the extensional database intermixed with calls to builtin predicates. In second level optimization, our major concern is the design of a query plan such that the resulting query will be executed more efficiently. In Prolog, the ordering of clauses in a program, and the ordering of goals in the right-hand side of a clause, is important control information that helps to determine the way a program is executed. This control information permits generation of an efficient query plan. The control information that is critical

to the query planner is described in [23]. Currently, we are applying the two-level optimization technique to the Tangram Stream Processor.

5.9. The Synopsis of Database Responses (Chung-Dak Shum, Dick Muntz)

Conventional responses in database systems, usually given as lists of atomic objects, although sufficient to serve the purpose of conveying information, do not necessarily provide efficient and effective communications between a user and the system. Recently, new notions of answers to queries have been receiving more research interest. For example, in [45], an answer to a query is expressed in terms of both atomic facts and general rules; in [29], intensional descriptions or concepts are being used as part of an answer. This latter notion of answers is particularly helpful when the number of entities or objects which satisfy the query is very large.

In [29], the answer to a query is expressed not as a set of individuals, but as a set of concepts or predicates, whose extensions may not be explicitly represented. Now suppose that we have retrieved a set of individuals as the answer to a conventional database query, and we want to re-express the answer in terms of a set of concepts. Those concepts must, of course, be pre-defined; otherwise, the user may not be familiar with them and the answer in terms of those concepts thus will not make too much sense. One of the immediate drawbacks to such an approach of expressing answers is that the extensions of the pre-defined concepts often do not satisfy the query conditions as a whole. As a result, we cannot express answers the way we want except in very rare cases.

We consider expressions for answers in terms of concepts and individuals [80]. Exceptions within individual concepts are allowed. Two criteria are defined as measures of the *goodness* of such expressions: (i) minimizing the total number of terms; (ii) minimizing the number of exceptions. Expressions satisfying these two criteria are called optimal expressions. We have shown that, under a strict taxonomy of concepts, any two optimal expressions for an extensional answer share the same set of terms. The inductive proof also leads to an algorithm for obtaining such expressions. Generalizing the strict taxonomy of concepts to a join-semilattice of concepts eliminates the term uniqueness property and also makes the problem of finding an optimal expression intractable. The problem under multiple taxonomies, although it involves a restricted type of join-semilattice, remains intractable.

One of the motivations behind our interest in different forms of answers is their *conciseness*. However, if there is a large number of individuals within a concept and approximately half of the individuals satisfy the query. Using an expression of concepts and individuals, no longer are we able to express our answer concisely. If we insist on concise answers, one possible 'solution' is to sacrifice *preciseness* for conciseness [81]. For each concept, we associate a count of its individuals and a count of qualified individuals which satisfy the query and refer to them, collectively, as a quantified concept. An aggregate or imprecise response is just an expression of quantified concepts. We study the tradeoff between conciseness and preciseness. Conciseness is measured by the length or the number of quantified concepts in an expression, and preciseness is measured by the entropy or the amount of uncertainty associated with the expression. Given its length, an expression with the minimum amount of entropy is considered optimal. Under a one-level taxonomy with the same cardinalities for all leaf concepts, the problem of finding an optimal expression can be solved inexpensively. An efficient heuristic is also proposed for the

general one-level taxonomy. For a taxonomy of more than one level, an algorithm is suggested. Although it is does not always lead to an optimal expression, it avoids the combinatorial explosion associated with the problem and appears to lead to good solutions.

Our work on imprecise responses is closely related to

- Statistical Databases
- Categorical Databases

Studies on statistical database management systems [40] suggests the definition summary tables in databases which are physically stored and maintained as redundant data as well as the original global database. Making use of such summaries, a large class of queries can be answered without extensively accessing data from the global databases. Currently, we are interested in the representations of such summaries under the general context of information abstraction.

6. Tangram Industrial Strength Prolog

Prolog goes a long way toward providing the kind of declarative modeling functionality we desire, and has the added benefit that it has a strong connection (from its logical foundation) with relational database systems. Still, to provide the 'industrial strength' environment we require for Tangram, a number of extensions to Prolog must be made:

- Module management
- Prolog database management
- Object management

Tangram Prolog is an extended Edinburgh-style Prolog system, augmented with a development environment and modules of low-level primitives for the functions listed above. Modules permit selective access to subsets of these primitives to individual Prolog processes.

6.1. Module Management (Tom Page, Dick Muntz)

The concept of reducing software complexity through modularization is well known and essential. Conventional languages have employed procedures and abstract data typing techniques to achieve modularity. Program modules can be constructed independently and composed to form larger systems. Access to a module is available only via its published interface. Internal data structures and procedures are invisible outside the module. The design and implementation process can be facilitated by transparently replacing initial, simple implementations of data structures or services with more sophisticated versions which maintain the same well-defined interface.

By contrast relatively little work has been done on modularization in logic programming systems. Tangram Prolog subdivides the name space of procedures so that each module has its own complete name space. Different parts of a system can be written without knowledge about the local names of other parts. Modules can be collected into libraries to group independent subsystems. Libraries are modules of modules which have their own published interfaces and hide the interfaces of internal modules.

6.2. Prolog Database Management (Tom Page, Dick Muntz)

There is considerable interest in combining database and logic programming technologies [66, 46, 90, 93, 94]. The motivation stems primarily from appreciation of the complementary benefits of the two technologies which were developed largely independently [92, 83, 76]. Our modeling environment requires more sophisticated interpretation of data than current database systems provide as well as efficient access to larger volumes of data than current Prolog implementations afford.

Many attempts at connecting Prolog with Relational DBMSs have been documented over the past few years [17, 94]. However, simply connecting the two systems via an interface is woefully naive [22]. Current Prolog implementations were designed to provide very fast unification over small atom spaces. The problem is that all data that is brought into the Prolog workspace becomes tightly intertwined in order to optimize unification, the performance bottleneck in Prolog. Atoms are not easily garbage collected or dropped from the workspace when they are no longer of interest. The volume of data in database applications quickly swamps current Prolog

system tables. The size of applications envisioned taxes the Prolog programming environment beyond its limits.

Our assertion is that the database should be viewed as a huge virtual workspace for Prolog. This workspace must cache units of data that are persistently stored in the database.

The workspace management replacement algorithm must handle three patterns of database access.

(1) Large flat relations may be accessed via the stream interface. Streams are sequences of terms that flow into transducers. An input stream can be discarded after processing by the transducer.

(2) Large complex objects may be integrated into the Prolog workspace by 'opening' them, and then operating upon them as though situated in memory. When they are 'closed', they are either discarded or rewritten in the database.

(3) Modules of code stored in the database may be brought into the Prolog workspace. Code in the modules may then be run until the modules are discarded.

Standard Prolog implementations lack the ability to deal with code or data as modules. Each clause is independent in Prolog; relationships or structure among clauses are not expressed but rather established operationally via inference [20]. While Prolog programs exhibit little locality when viewed from this perspective of extreme modularity, modules provide a logical bundle of code/data within which locality is expected. Thus, we propose a new organization for the Warren Abstract Machine [36,91] in which separate atom and functor tables are built for each module. If locality exists, program execution will cross module boundaries infrequently relative to intra-module unifications. Within each module, atom unification will be very fast at the expense of translating arguments across module boundaries.

6.3. Object Management (Tom Page, Dick Muntz)

One characteristic of a logic programming language is that the same call with the same arguments returns the same results in any context [28]. All of the information needed to perform an operation must be present in the arguments and not recorded in the state of the program. This makes it very difficult to achieve the important principle of data abstraction in a logic programming language. We must have a way to represent persistent state information in Prolog, especially if we are to provide a transparent database model.

Tangram Prolog will provide an object-oriented programming module which adds the notion of persistent objects to the language. Objects may be managed by the database and become active when they are addressed with messages which they support. Name binding issues with respect to inherited methods will be explored.

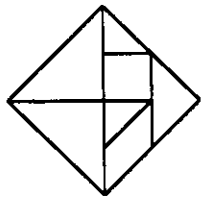
7. Future Projects

The list of open projects is vast. For example, translation is a pervasive problem. The multitude of differing tool input/output formats requires good translation tools, and mapping among different kinds of models is an important related problem. The SARA project at UCLA has dealt heavily with translation issues, and it seems likely that some SARA tools can be adapted here.

Several important projects that we have begun to address are listed here:

- Estimation of time/space requirements to find solutions of models
- Automated use of tools/testbeds given an experiment
- Automated sensitivity analysis
- Automated explanation
- Induction of analytic models from behavior (learning)
- Paradigm systems for
  - Production Systems/Triggers
  - Petri nets

> Those who sell electronic gadgetry would have us believe that the computer age will be a new era for scientific thought and humanity; they might also point out the basic problem, which lies in the construction of models.
>
> – Rene Thom [88]



References

1. Abelson, H. and G. Sussman, *The Structure and Analysis of Computer Programs*, pp. 242-292, MIT Press, Boston, MA, 1985.
2. Adiba, M. and N.B. Quang, "Historical Multimedia Databases," *Proc. Twelfth Intnl. Conf. on Very Large Data Bases*, pp. 63-70, Kyoto, Japan, 1986.
3. Allen, J.F., "Maintaining Knowledge about Temporal Intervals," *CACM*, vol. 26, no. 11, pp. 832-843, November 1983.
4. Allen, J.F., "Towards a General Theory of Action and Time," *Artificial Intelligence*, vol. 23, pp. 123-154, 1984.
5. Atzeni, P. and D.S. Parker, "Set Containment Inference and Syllogisms," Technical Report CSD-880022 , UCLA Computer Science Dept., June 1987, issued March 1988. To appear, *Theoretical Computer Science*
6. Bancilhon, F., T. Briggs, S. Khoshafian, and P. Valduriez, "FAD, a Powerful and Simple Database Language," *Proc. Thirteenth Intnl. Conf. on Very Large Data Bases*, Brighton, England, 1987.
7. Batory, D.S. and T.Y. Leung, "Implementation Concepts for an Extensible Data Model and Data Language," Tech. Report TR-86-24, Dept. of Computer Sciences, Univ. of Texas at Austin, Austin, TX 78712, 1986.
8. Batory, D.S., "A Molecular Database Systems Technology," Tech. Report TR-87-23, Dept. of Computer Sciences, Univ. of Texas at Austin, Austin, TX 78712, 1987.
9. Batory, D.S., T.Y. Leung, and T. Wise, "Implementation Concepts For an Extensible Data Model and Data Language," *ACM Trans. Database Systems*, to appear.
10. Becker, R.A. and J.M. Chambers, *S: An Interactive Environment for Data Analysis and Graphics*, Wadsworth, Inc., Belmont, CA, 1984.
11. Beeri, C., S. Naqvi, R. Ramakrishnan, O. Shmueli, and S. Tsur, "Sets and Negation in a Logic Database Language (LDL1)," *Proc. Sixth ACM Symp. on Principles of Database Systems*, pp. 21-37, San Diego, March 1987.
12. Ben-Zvi, J., "The Time Relational Model," Ph.D. Dissertation, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, 1982.
13. Bennett, J.S., L. Creary, R. Engelmore, and R. Melosh, "SACON: A knowledge-based consultant for structural analysis," Technical Report HPP 78-23, Computer Science Dept., Stanford University, 1978.
14. Berson, S., E. de Souza e Silva, and R.R. Muntz, "An Object-Oriented Methodology for the Specification of Markov Models," Technical Report CSD-870030, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, July 1987.
15. Bic, L. and R.L. Hartmann, "AGM: A Dataflow Database Machine," Technical Report, Dept. of Information and Computer Science, Univ. of California at Irvine, February 1987.
16. Bobrow, D.G., "If Prolog is the Answer, What is the Question?," *Proc. Intnl. Conf. on Fifth Generation Computer Systems*, pp. 138-145, ICOT, Tokyo, November 1984.
17. Bocca, Jorge, "On the Evaluation Strategy of EDUCE," in *Proceedings SIGMOD 1986*, pp. 368-378, 1986.

18. Bolour, A., T.L. Anderson, L.J. Dekeyser, and H.K.T. Wong, "The Role of Time in Information Processing," *ACM SIGMOD Record*, vol. 12, no. 3, pp. 27-50, 1982.

19. Borning, A., "The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory," *ACM Transactions on Programming Languages and Systems*, vol. 3, no. 4, October 1981.

20. Brodie, Michael L. and Matthias Jarke, *On Integrating Logic Programming and Databases*, pp. 40-62, Computer Corporation of America, Cambridge, Massachusetts.

21. Bruynooghe, M., G. Janssens, A. Callebaut, and B. Demoen, "Abstract Interpretation: Towards the Global Optimization of Prolog Programs," *Proc. Fourth International Symposium on Logic Programming*, pp. 192-204, IEEE Computer Society, 1987.

22. Ceri, Stefano, Georg Gottlob, and Gio Wiederhold, "Interfacing Relational Databases and Prolog Efficiently," in *Proceedings 2nd Expert Database Systems Conference*, pp. 141-153, 1986.

23. Chau, L., "Two-Level Query Optimization," Draft, UCLA Computer Science Dept., July 1987.

24. Chau, L., "Functional Grammars and Stream Pattern Matching," Draft, UCLA Computer Science Dept., March 1988.

25. Chou, C., "Relaxation Processes: Theory, Case Studies and Applications," Report CSD-860057 (M.S. Thesis), UCLA Computer Science Dept., 1986.

26. Clark, K. and S. Gregory, "Notes on the Implementation of PARLOG," *J. Logic Programming*, vol. 2, no. 1, pp. 17-42, 1985.

27. Clifford, J. and D.S. Warren, "Formal Semantics for Time in Databases," *ACM Transactions on Database Systems*, vol. 8, no. 2, pp. 214-254, June 1983.

28. Conery, J. S., "Object Oriented Programming with Horn Clause Logic," Draft, University of Oregon, July 1987.

29. Corella, F., "Semantic Retrieval and Levels of Abstraction," in *Expert Database Systems*, ed. L. Kerschberg, Benjamin-Cummings, New York, 1985.

30. Cousot, P. and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," *Conf. Rec. 4th ACM Symp. on Princ. of programming languages*, pp. 238-252, 1977.

31. Dantzig, G., "Mathematical Programming and Decision Making in a Technological Society," Tech. Report SOL 82-11, Systems Optimization Laboratory, Dept. of Operations Research, Stanford Univ., August 1982.

32. DeGroot, D. and G. Lindstrom, *Logic Programming: Functions, Relations, and Equations*, Prentice-Hall, 1986.

33. Dean, T.L. and D.V. McDermott, "Temporal Data Base Management," *Artificial Intelligence*, vol. 32, pp. 1-55, 1987.

34. Debray, S. K. and D. S. Warren, "Automatic Mode Inference for Prolog Programs," *Proc. Third International Symposium on Logic Programming*, pp. 78-88, IEEE Computer Society, 1986.

35. Dolk, D.R., "A Generalized Model Management System for Mathematical Programming," *ACM Trans. Math. Software*, vol. 12, no. 2, pp. 92-126, June 1986.

36. Gabriel, John, Tim Lindholm, E.L. Lusk, and R.A. Overbeek, "A Tutorial on the Warren Abstract Machine for Computational Logic," ANL-84-84, pp. 1-52, Argonne National Laboratory, Argonne, Illinois, June 1985.

37. Gale, W.A., "REX Review," in *Artificial Intelligence & Statistics*, ed. W.A. Gale, Addison-Wesley, 1986.

38. Golshani, F., "The Basis of a Dataflow Model for Query Processing," *Proc. Eighteenth HICSS*, Honolulu, January 1985.

39. Gregory, S., *Parallel Logic Programming in PARLOG: The Language and its Implementation*, Addison-Wesley, Reading, MA, 1987.

40. Hebrail, G., "A Model of Summaries for Very Large Databases," *Proc. of the 3rd Int. Workshop on Statistical and Scientific Database Management* , Luxembourg, 1986.

41. Helm, R. and K. Marriott, "Declarative Graphics," *Proc. Third Intnl. Conf. on Logic Programming*, pp. 512-527, Springer-Verlag, London, 1986.

42. Hoffmann, C.M. and J.E. Hopcroft, "Simulation of Physical Systems from Geometric Models," Preprint, Dept. of Computer Science, Cornell University, Ithaca, NY, 1986.

43. Huntsinger, R., "On Constraint-Oriented Environments for Continuous System Simulation," CSD-880020, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, March 1988.

44. Huntsinger, R., "Representation Transformation in Constraint Satisfaction Systems," CSD-880018, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, March 1988.

45. Imielinski, T., "Intelligent Query Answering in Rule Based Systems," *J. Logic Programming*, vol. 4, no. 3, September 1987.

46. Jarke, Matthias, Jim Clifford, and Yannis Vassiliou, "An Optimizing Prolog Front-End to a Relational Query System," in *Proceedings ACM SIGMOD*, pp. 296-306, 1984.

47. Joyce, J., G. Lomow, K. Slind, and B. Unger, "Monitoring Distributed Systems," *ACM Trans. Computer Systems*, vol. 5, no. 2, pp. 121-150, May 1987.

48. Kahn, K., "A Primitive for the Control of Logic Programs," *Proc. Symp. on Logic Programming*, pp. 242-251, IEEE Computer Society, Atlantic City, 1984.

49. Kowalski, R.A., "Database Updates in the Event Calculus," Research Report 86/12, Department of Computing, Imperial College, London, 1986.

50. Kurose, J.F., K.J. Gordon, R.F. Gordon, E.A. MacNair, and P.D. Welch, "A Graphics-Oriented Modeler's Workstation Environment for the RESearch Queueing Package (RESQ)," *Proc. Fall Joint Computer Conference*, pp. 709-718, IEEE Computer Society #743, 1986.

51. LeDoux, C.H., "A Knowledge-Based System for Debugging Concurrent Software," Technical Report CSD-860060 (Ph.D. Dissertation), UCLA Computer Science Dept., Los Angeles, CA 90024-1596, March 1986.

52. Lee, R.M., H. Coelho, and J.C. Cotta, "Temporal Inferencing On Administrative Databases," *Information Systems*, vol. 10, no. 2, pp. 197-206, 1985.

53. Li, P-Y.P. and A.J. Martin, "The Sync Model: A Parallel Execution Method for Logic Programming," *Proc. Symp. on Logic Programming*, pp. 223-234, IEEE Computer Society, Salt Lake City, 1986.

54. Lindstrom, G. and P. Panangaden, "Stream-Based Execution of Logic Programs," *Proc. Symp. on Logic Programming*, pp. 168-176, IEEE Computer Society, Atlantic City, 1984.

55. Lum, V., P. Dadam, R. Erbe, J. Guenauer, P. Pistor, G. Walch, H. Werner, and J. Woodfill, "Designing DBMS Support for the Temporal Dimension," *Proc. ACM SIGMOD Conference on Management of Data*, pp. 115-130, June 1984.

56. Mackinlay, J., "Automating the Design of Graphical Presentations of Relational Information," *ACM Transactions on Graphics*, vol. 5, no. 2, pp. 110-141, April 1986.

57. McCrosky, C.D., J.J. Glasgow, and M.A. Jenkins, "Nial: A Candidate Language for Fifth Generation Computer Systems," *Proc. ACM'84 Annual Conference*, pp. 157-166, San Francisco, October 1984.

58. Melamed, B., "The Performance Analysis Workbench: An Interactive Animated Simulation Package for Queueing Networks," *Proc. Fall Joint Computer Conference*, pp. 729-740, IEEE Computer Society #743, 1986.

59. Mellish, C. S., "Abstract Interpretation of Prolog Programs," *Proc. Third Intnl. Conf. on Logic Programming*, pp. 463-474, Springer-Verlag, London, 1986.

60. More, T., "Axioms and Theorems for a Theory of Arrays," *IBM J. Res. Develop*, vol. 17, no. 2, pp. 135-175, 1973.

61. More, T., "The Nested Rectangular Array as a Model of Data," *Proc. APL79*, pp. 55-73, May 1979.

62. Naish, L., "All Solutions Predicates in Prolog," *Proc. Symp. on Logic Programming*, pp. 73-77, IEEE Computer Society, Boston, 1985.

63. Narain, S., "LOG(F): A New Scheme for Integrating Rewrite Rules, Logic Programming and Lazy Evaluation," Technical Report CSD-870027, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, 1987.

64. Narain, S., "LOG(F): An Optimal Combination of Logic Programming, Rewrite Rules and Lazy Evaluation," Ph.D. Dissertation, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, 1988.

65. Page, T.W., "Prolog Basis for a Data-Intensive Modeling Environment," Dissertation Prospectus, UCLA Computer Science Department, Los Angeles, CA 90024-1596, March 1988.

66. Parker, D.S., M. Carey, F. Golshani, M. Jarke, E. Sciore, and A. Walker, "Logic Programming and Databases," in *Proceedings First International Workshop on Expert Database Systems*, Kiawah Island, SC, October 1984. (also in *Expert Database Systems*, L. Kershberg, ed. 1985).

67. Parker, D.S. and M. Matsuo, "Incompleteness in Conceptual Modeling," *Proc. Advanced Database Symposium*, Tokyo, August 1986.

68. Parker, D.S., "The Modulus Knowledge Representation System," Draft, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, 1987.

69. Parker, D.S., "Partial Order Programming," Technical Report CSD-870067, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, 1987.

70. Parker, D.S. and R.R. Muntz, "A Theory of Directed Logic Programs and Streams," Technical Report CSD-880031, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, April 1988.

71. Parker, D.S., R.R. Muntz, and L. Chau, "The Tangram Stream Query Processing System," Technical Report CSD-880025, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, March 1988.

72. Parker, D.S., T. Page, and R.R. Muntz, "Improving Clause Access in Prolog," Technical Report CSD-880024, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, March 1988.

73. Sadri, F., "Three Recent Approaches to Temporal Reasoning," Research Report 86/23, Department of Computing, Imperial College, London, Nov. 1986.

74. Sagie, I., "Computer-Aided Modeling and Planning (CAMP)," *ACM Trans. Math. Software*, vol. 12, no. 3, pp. 225-248, Sept. 1986.

75. Scheifler, R.W. and J. Gettys, "The X Window System," *ACM Transactions on Graphics*, vol. 5, no. 2, pp. 79-109, April 1986.

76. Sciore, E. and D.S. Warren, "Towards an Integrated Database-Prolog System," in *Expert Database Systems*, ed. Larry Kerschberg, pp. 293-305, Benjamin/Cummings, Menlo Park, CA, 1986.

77. Segev, A. and A. Shoshani, "Logical Modelling of Temporal Data," Tech. Rep. LBL-22636, Computer Science Research Department, Lawrence Berkeley Laboratory, Mar. 1987.

78. Shapiro, E.Y., *Concurrent Prolog: Collected Papers*, MIT Press, Cambridge, MA, 1987.

79. Shoshani, A. and K. Kawagoe, "Temporal Data Management," *Proc. Twelfth Intnl. Conf. on Very Large Databases*, pp. 79-88, Kyoto, Japan, August 1986.

80. Shum, C-D. and R. Muntz, "Implicit Representation of Extensional Answers," *Proc. on 2nd International Conference on Expert Database Systems*, 1988.

81. Shum, C-D. and R. Muntz, "An Information-Theoretical Study on Aggregate Responses," Technical Report , UCLA Computer Science Dept., Los Angeles, CA 90024-1596 , 1988.

82. Sinclair, J.B. and S. Madala, "A Graphical Interface for Specification of Extended Queueing Network Models," *Proc. Fall Joint Computer Conference*, pp. 709-718, IEEE Computer Society #743, 1986.

83. Smith, J.M., "Expert Database Systems: A Database Perspective," in *Proceedings First Int. Workshop on Expert Database Systems*, 1984.

84. Snodgrass, R., "The Temporal Query Language TQuel," *ACM Transactions on Database Systems*, vol. 12, no. 2, pp. 247-298, June 1987.

85. Studer, R., "Modeling Time Aspects of Information Systems," *Proc. Second Intnl. Conf. on Data Engineering*, Los Angeles, CA, 1986.

86. Subrahmanyam, P.A. and J-H. You, "Conceptual Basis and Evaluation Strategies for Integrating Functional and Logic Programming," *Proc. Symp. on Logic Programming*, pp. 144-153, IEEE Computer Society, Atlantic City, 1984.

87. Takeuchi, A., "Affinity between Meta Interpreters and Partial Evaluation," Technical Report TR-166, ICOT, Tokyo, April 1986.

88. Thom, R., *Structural Stability and Morphogenesis*, Wiley, 1975.

89. Tsur, S. and C. Zaniolo, "LDL: A Logic-Based Data Language," *Proc. Twelfth Intnl. Conf. on Very Large Data Bases*, pp. 33-41, Kyoto, Japan, 1986.

90. Ullman, J., "Implementation of Logical Query Languages for Databases," in *Proceedings ACM SIGMOD Conference on Management of Data*, 1985.

91. Warren, David H.D., "An Abstract Prolog Instruction Set," Technical Report 309, SRI International, Menlo Park, CA 94025, October 1983.

92. Zaniolo, C., "The Representation and Deductive Retrieval of Complex Objects," in *Proceedings Very Large Data Bases*, pp. 458-469, Stockholm, Sweden, 1985.

93. Zaniolo, C., "Prolog — A Database Query Language for All Seasons," in *Expert Database Systems*, ed. Larry Kerschberg, pp. 219-232, Benjamin/Cummings, Menlo Park, CA, 1986.

94. Zaniolo, C., "Safety and Compilation of Non-Recursive Horn Clauses," *Expert Database Systems*, pp. 167-178, April, 1986.