Investigation of the Rule Modeling Capabilities in Huron

Jacob L. Cybulski
Amdahl Australian Intelligent Tools Programme
La Trobe University, Bundoora, Vic. 3083, Australia
Phone: +61 3 479 1270, Fax: +61 3 470 4915, Email: jacob@latcs1.lat.oz.au

Abstract

This report investigates the scope and limitations of Huron's most powerful feature - its rule system. Huron's ability to capture programming solutions in rules is contrasted with those of the most popular representative of rule-based languages - Prolog. Programmer's productivity in writing rule systems in Huron is assessed by setting a case study involving solving a small problem in both Huron and Prolog and comparing the efficiency of programming tasks during the entire process.

1. Introduction

Definitions. Rule-based systems [Harmon and King 1985], originated from conventional decision-support systems, such as Decision Tables, and were popularised through the Artificial Intelligence research and development of Expert Systems. Rule-based systems' behaviour can be described by a number of *rules* and *facts* of a very simple format (Ref. Table 1).

Production	Logical	Facts
if condition-1 **and** condition-2 **and** ... condition-M **then** action-1, action-2, ... action-N.	**if** evidence-1 **and** evidence-2 **and** ... evidence-M **then** conclusion-1, conclusion-2, ... conclusion-N.	evidence-1. evidence-2. ... evidence-M.

Table 1 - Rules and facts

In a production system, rules describe a set of partial behaviours activated upon the system reaching certain specific states of computation. In such a system, once a set of conditions is satisfied, the rule's context is thus determined, leading to the rule

firing, which causes all the actions associated with the rule to be performed. The actions may in turn trigger other rules in the system, thus, propagating the control throughout the rule database.

In a logical system, rules are used to generate a set of possible conclusions from the factual evidence stored in its database. Such derived conclusions may in turn be used as new evidence in the reasoning process (forward chaining). The reasoning cycle may be reversed to prove a hypothesis to be correct. In such a case the rules are used to identify the contributing evidence, the factual evidence terminates the process with the success, otherwise the system will search for a way of deriving it from other rules in the system (backward chaining).

Modern rule-based systems, such as Prolog, can usually allow both interpretations of their rules, i.e. procedural (production) or declarative (logical).

Background. Huron [Huron 1991] is a commercial programming environment, embedding a rule-based language for the specification of program activities. Due to the product's novelty, the language, its structure, expressive power and applicability are virtually unknown. Furthermore, the rules were designed to support the database and user-interface manipulations, hence, they are quite unique in their form and operation. Therefore, it was suggested to assess Huron features by comparing its rule sub-system against one of the most popular representatives of the rule-based products, Prolog [Clocksin and Mellish 1987]. Some aspects of this evaluation were based on qualitative analysis of both products properties, others relied on experimental implementation of a small system in both Huron and Prolog.

Experiment. The crux of the experiment was an implementation of a Naive Banking System (NBS - refer appendix A) providing the hypothetical bank (NBA) customers with a suite of simple services. NBS was to allow its customers to open and close loan, term deposit or saving (with or without overdraft) accounts, which could be owned either individually or jointly with other bank customers. Typical account transactions were to include deposits, withdrawals, transfers, and balances. On the monthly basis the system would send its customers a statement of all transactions performed in that period of time. Interest was to be calculated monthly on the minimum monthly balance (or maximum liability for loan and overdraft accounts). At the end-of-month processing the bank was also required to perform a number of integrity and audit checks to eliminate any possibility of tampering with its accounting and computer systems.

The system was to be designed with rule systems in mind and subsequently implemented in both Huron and Prolog.

2. Rule-Based Design of NBS

Database. The NBS system consists of a database of customers (Cust), accounts (Acc) and their types (Def), an interest rate table (Int), and a transaction file (Trans). The customer and account files are correlated to allow bank customers to jointly own accounts (Owns). The records of these six files are defined in Table 2. The bank can track its cash, cheque, and interest/charge transactions by designating special-purpose accounts numbered 1, 2, and 3 in the Acc file.

```
Int =                                                  Interest rate table
    Int_Bracket : LongInt,                                Interest bracket
    Int_Rate : Float.                                        Interest rate

Cust =                                                        Customer file
    Cust_No : LongInt,                          Customer identification number
    Cust_Surname : Char[40],                                 Customer surname
    Cust_First : Char[20],                                 Customer first name
    Cust_Street : Char[40],                              Street number and name
    Cust_City : Char[20],                                           City name
    Cust_State : Char[3],                                          State code
    Cust_PCode : Char[4].                                           Post code

Def =                                                       Account type table
    Def_No : LongInt,                                      Account type number
    Def_Name : Char[20],                                       Type trade name
    Def_Type : Char[10],                                   Type reference name
    Def_Min_Bal : Float,                  Account minimum permissible balance
    Def_Terms : LongInt.                                 Minimum deposit term

Acc =                                                         Account file
    Acc_No : LongInt,                     Account number - 1,2,3 are reserved
    Def_No : LongInt,                                             Account type
    Acc_Name : Char[20],                                          Account name
    Acc_Open_Date : Date,                                 Account opening date
    Acc_Open_Time : Time,                                 Account opening time
    Acc_Close_Date : Date,                                Account closing date
    Acc_Close_Time : Time,                                Account closing time
    Acc_Prev_Bal : Float,                 Balance at the end of previous term
    Acc_Min_Bal : Float,                            Minimum balance this term
    Acc_Curr_Bal : Float.                             Current account balance

Owns =                                                  Account ownerhip file
    Owns_CustNo : LongInt,             Customer (jointly) owning an account
    Owns_AccNo : LongInt.                                        Owned account

Trans =                                                       Transaction file
    Trans_Date : Date,                                        Transaction date
    Trans_Time : Time,                                        Transaction time
    Trans_Ref : Char[20],                                       Reference note
    Trans_From : LongInt,           Account from which funds are transfered
    Trans_To : LongInt,             Account into which funds are transfered
    Trans_Amount : Float.                                    Transfered amount
```

Table 2 - NBS data dictionary

Name	File	Def	Acc	Cust	Owns	Int	Trans
File	Create Destroy	Create Destroy	Create Destroy	Create Destroy	Create Destroy	Create Destroy	Create Destroy
Access	Get Find	Get Find	Get	Get Find	Get Find	Get Find	Get
Update	Put Append Update Delete	Add Delete	Open Append Trans Close Delete	Add Update Delete	Add Delete	Add Delete	Add
Tests	Test	Test	Test	Test	Test	Test	Test
		File	File Def	File	Acc Cust	File	File Acc

Table 3 - NBS rule packages and their dependencies ( )

Maintenance. Each of the files can be accessed by a maintenance package, based on the generic *file* package, of rules aiding file manipulation and preserving its integrity constraints (Cf. Table 3), e.g. creating and destroying a file, getting, finding, adding, appending, replacing and deleting a record, and testing the package itself.

Not all of the functions were needed in each of the six file maintenance modules, in other cases the package was extended by additional functions. The Acc package has the most significant differences from File. It allows adding new accounts to the file by "opening" them. It also facilitates "deleting" records from the file providing they are first "closed". Account transactions "Trans" cause updates to the account balance.

Transactions. Once the individual account operations are implemented the transfer of funds between two different accounts can be easily accomplished (via Acc Trans operation). Addition of records to the transaction file causes updates to the relevant customer accounts. Transfers of cash, cheque, interest or charge into the customer account also affect one of the special bank accounts, i.e. CASH, CHEQUE and CHARGE, this approach provides a greater uniformity of banking transactions.

Audit. NBS limits its audit checks to balancing the entire accounting system (which must balance to 0) and checking monthly transaction files against individual account balances (previous balance plus all transactions must equal current balance).

3. Prolog and its Implementation

Prolog. Prolog (Programming in Logic) is one of the most popular rule-based programming systems. It is available on a number of platforms ranging from mainframes (e.g. IBM Prolog or IF Prolog), through minis and workstations (e.g. Quintus Prolog, NU Prolog or Prolog from BIM) to personal computers (e.g. Arity Prolog or LPA Prolog), there are even public domain versions of the language aimed at educational market (e.g. Open Prolog for Apple Macintosh or PD Prolog for IBM PCs). The Prolog notation introduced in this section is based on Edinborough Prolog, currently being considered as a possible ANSI standard. Different implementations of Prolog, however, have considerable extensions providing access to relational databases (e.g. Oracle, Ingress, DB2), windowing systems and interface toolkits (Sun Windows, X with OSF/Motif, Windows or Mac Interface), allowing interactive editing and debugging, interfacing foreign language, configuration managements, adding functions and utilities, etc. All code described in this report was developed using Macintosh Open Prolog.

Notation. Prolog rules take the form of logical clauses, they can be parametrized and are being traversed in the backward chaining order. Their form is as follows :-

```
fact(P1, P2, ..., PN).                                          (ii)
hypothesis(P1, P2, ..., PI) :-
   evidence-1(E11, E12, ..., E1J), ...,
   evidence-K(EK1, EK2, ..., EKL).
```

Facts. The core of the Prolog system consists of a database of rules and facts. A fact is a named record relating a number of ordered data fields. For instance, information about the bank customers and their addresses may be represented as follows :-

```
% Customer: CustNo, Last, First, Street, City, State, Zip          (iii)
cust(1, 'NBS', 'Pty. Ltd.', 'Great St.', 'Melbourne', 'VIC', '3001').
cust(2, 'Cybulski', 'Jacob', '1 Heap St.', 'Melbourne', 'VIC', '3001').
cust(3, 'Cybulski', 'Margaret', '1 Heap St.', 'Melbourne', 'VIC',
'3001').
cust(4, 'Reed', 'Karl', '11 Main Rd.', 'Eltham', 'VIC', '3456').
cust(5, 'Beitz', 'Andrew', '8/12 Hop Crt.', 'Brisbane', 'QLD', '4789').
cust(6, 'Golebiowski', 'Paul', '12/3 Rope Ave.', 'Sydney', 'NSW',
'2089').
cust(7, 'Smith', 'John', '1/8 Average St.', 'Mt Little', 'TAS', '1878').
cust(8, 'Black', 'Mary', '6 Bright Rd.', 'Footscray', 'VIC', '3245').
cust(9, 'Brown', 'James', 'Plenty Rd.', 'Yalubalu', 'QLD', '4081').
```

or the bank accounts' information (simplified) :-

```
% Account: AccNo, AccType, Balance                    (iv)
acc(1, savings, 100).
acc(2, savings, 200).
acc(3, overdraft, -100).
acc(4, loan, -5000).
acc(5, loan, -2300).
acc(6, term, 10000).
```

or account ownership :-

```
% Ownership: CustNo, AccNo                            (v)
owns(1, 3).
owns(2, 1).
owns(3, 1).
owns(2, 4).
owns(8, 2).
owns(8, 5).
owns(9, 6).
```

Queries. The facts may be queried on any of the record fields by providing a set of sample field values (numbers, quoted values or lower-case symbols). Once the query finds a matching fact, the result is returned via query variables (upper-case symbols or underlines which are nameless variables). Examples of queries based on the tables described above :-

```
% Find Karl Reed's address                            (vi)
?- cust(N, 'Reed', 'Karl', Street, City, State, Zip).

Street = '11 Main Rd.'
City = 'Eltham'
State = 'VIC'
Zip = '3456'
```

```
% Find the first names of all customers named Cybulski  (vii)
?- cust(_, 'Cybulski', First, _, _, _, _).

First = 'Jacob'

First = 'Margaret'
```

The more complex queries are constructed by joining a number of sub-queries and relating them with common variables, e.g.

```
% Show the types and balances of all accounts owned by Queenslanders (viii)
?- cust(CustNo, _, _, _, _, 'QLD', _),
   owns(CustNo, AccNo),
   acc(AccNo, AccType, Bal).

CustNo = 9
AccNo = 6
AccType = term
Bal = 10000
```

Rules. Constructing a rule restricting the view of the database of facts permits greater selectivity of information presented to the user, e.g.

(ix)

```
% Rule allowing the view of state accounts
view_state_accounts(State, AccType, Bal) :-
    cust(CustNo, _, _, _, _, State, _),
    owns(CustNo, AccNo),
    acc(AccNo, AccType, Bal).

% Show the types and balances of all accounts owned by Queenslanders
?- view_state_accounts('QLD', AccType, Bal).

AccType = term
Bal = 10000
```

Prolog rules are also used for validating passed parameters, e.g. checking term deposit maturity for withdrawals :-

(x)

```
% Rule checking whether a term deposit is mature for withdrawal
acc_term_transaction_invalid(TermSoFar, MaturityTerm, Amount) :-
    Amount < 0,
    TermSoFar =< MaturityTerm.

% Check the transation on the term deposit
?- acc_term_transaction_invalid(10, 24, -100).
success
```

or performing calculations, e.g. calculating the interest on deposit :-

(xi)

```
% Rule calculating interest payment over one period
calc_int_pay(PrevBal, Int, NewBal) :-
    NewBal is PrevBal * (1 + Int).

% Calculate account's balance after interest
?- calc_int_pay(1000, 0.10, NewBal).

NewBal = 1100
```

A rule may be viewed not only as a set of evidence to support the hypothesis, but also as procedures consisting of a sequence of evidence-checking steps leading to either proving or disproving the hypothesis. Procedures are useful in organising the control of Prolog programs and providing algorithmic solutions to some types of problems. For instance, a sequence of steps leading to the successful completion of the funds transfer between two bank accounts may be implemented in the following rule :-

(xii)

```
% Rule allowing transfer of funds between two bank accounts
acc_transfer_transaction(Time, From_Acc, To_Acc, Amount) :-
    Withdrawal is -Amount,
    acc_trans(From_Acc, Time, Withdrawal),
    acc_trans(To_Acc, Time, Amount),
    trans_log(Time, From_Acc, To_Acc, Amount).

?- acc_transfer_transaction('92/04/15 15:04:11', 2, 3, 500).
success
```

Control. Every time the rule finds a partially matching evidence, Prolog will attempt to assign (*unify*) the rule variables to the values improving the match. Such assignment, however, is not committed until the entire chain of reasoning is completed and the top-most hypothesis is found to be true. If the rule cannot be supported by any evidence in the database, Prolog will attempt to backtrack to the point of the nearest available branch in reasoning and rollback all of its variable assignments, subsequently restarting the whole process anew (*resolution*). (Such control mechanism allowed finding only the Queensland customers of the bank in ix.)

User Interface. Prolog standard does not specify a requirement for elaborate man-machine interaction, which is limited to the simple read and write from and to a file (including the standard IO). Most of the implementations, however, do have an external library support, thus, giving the language sophisticated graphical interface, e.g. NU-Prolog supports CURSES and X, interface to X/Motif is also provided in IF/Prolog, Quintus Prolog, Prolog from BIM and many others, Prolog from BIM also works under SUN Windows, Arity Prolog takes advantage of Microsoft Windows, LPA Prolog supports Macintosh Toolkit, etc. Many of the Prolog versions working under a proprietary windowing system, are also supplied with the specialised interface building kit.

Summary. Rule-based systems, and Prolog in particular, allow easy encoding and storing of facts and rules (i-v) and their subsequent querying (vi-viii). The rules may be used for validating input arguments (x), calculating formulae (xi), creating database views (ix), and implementing procedural algorithms (xii). Prolog also allows what-if analysis of query data, based on its nested transaction model, i.e. unification and resolution. Majority of commercial releases of Prolog come with the sophisticated user interface building kits.

4. Huron and its Implementation

Huron. Huron is an Amdahl's programming environment encompassing a number of inter-dependent subsystems which provide the following services: database management, transaction monitoring, a rule/data driven programming language, foreign language interfacing, etc.

Notation. Huron provides a sophisticated visual environment allowing viewing and manipulating its data tables and rules via screen forms, thus, simplifying the language syntax, but at the same time somewhat complicating the user interaction.

Facts. Huron stores its facts in a database of relational tables (Cf. example i). Tables are collections of records, of which fields conform to a limited set of data types. A table definition may include a specification of a primary key (possibly compound of several fields), the order of records in the table, required or optional fields, field default values, etc. To increase the efficiency of table access, tables may be parametrised and referenced with other tables. Rules can be attached to a table to act as triggers and validators on access.

(i)

No	Last	First	Street	City	State	Zip
1	NBS	Pty. Ltd.	Great St.	Melbourne	VIC	3001
2	Cybulski	Jacob	1 Heap St.	Melbourne	VIC	3001
3	Cybulski	Margaret	1 Heap St.	Melbourne	VIC	3001
4	Reed	Karl	11 Main Rd.	Eltham	VIC	3456
5	Beitz	Andrew	8/12 Hop Crt.	Brisbane	QLD	4789
6	Golebiowski	Paul	12/3 Rope Ave.	Sydney	NSW	2089
7	Smith	John	1/8 Average St.	Mt Little	TAS	1878
8	Black	Mary	6 Bright Rd.	Footscray	VIC	3245
9	Brown	James	Plenty Rd.	Yalubalu	QLD	4081

Queries. Simple queries or updates may be issued from the Table Browser, Single Occurrence Editor and Table Editor utilities. The utilities allows visual inspection of any individual table, selection of records according to the criteria imposed on the record fields (Cf. example ii), adding and modifying displayed records. The more complex queries and updates, however, have to be entered via rules.

```
SELECT STATE = LIKE 'V'
```

(ii)

No	Last	First	Street	City	State	Zip
1	NBS	Pty. Ltd.	Great St.	Melbourne	VIC	3001
2	Cybulski	Jacob	1 Heap St.	Melbourne	VIC	3001
3	Cybulski	Margaret	1 Heap St.	Melbourne	VIC	3001
4	Reed	Karl	11 Main Rd.	Eltham	VIC	3456
8	Black	Mary	6 Bright Rd.	Footscray	VIC	3245

Rules. Unlike Prolog, Huron rules have a procedural semantics, similar to that of production systems. The rules consist of a number of components, i.e. their definition, conditions, actions, exceptions, Y/N quadrant and action sequence numbers (Cf. example iii).

Each rule is defined by its name, formal arguments and local variables. Huron variables, whether local, global or formal, can only store values of the simplest type, and the more complex data structures have to be handled via tables. Any local variable becomes globally accessible in the rules called from the rule defining that variable (dynamic scoping).

The rule arguments and globals may then be assessed by a number of conditions organised in a simple "if-then-else-if" structure. Each of the conditions must be either a simple relation between two arguments or be a call to the logical rule. More complex conditions constructed with the logical operators are not allowed. Also the "else" clauses are not permitted unless they are a continuation of the last "if".

(iii)

DEF	JLC_ACC_OPEN(TYPE, NAME, BAL); _ LOCAL NOW_DATE, NOW_TIME;		
CONDS	_ JLC_INVALID_NAME(NAME); _ JLC_INVALID_BAL(TYPE, BAL)	Y N N Y N	Y&N Q

ACTIONS	_ NOW_DATE = $SYSTEM_DATE;
	_ NOW_TIME = TIME;
	_ JLC_ACC.DEF_NO = TYPE;
	_ JLC_ACC.ACC_NAME = NAME;
	_ JLC_ACC.ACC_CURR_BAL = BAL;
	_ JLC_ACC.ACC_OPEN_DATE = NOW_DATE;
	_ JLC_ACC.ACC_OPEN_TIME = NOW_TIME;
	_ INSERT JLC_ACC;
	_ COMMIT;
	_ GET JLC_ACC WHERE
	_ ACC_OPEN_DATE = NOW_DATE &
	_ ACC_OPEN_TIME = NOW_TIME;
	_ RETURN(JLC_ACC.ACC_NO);
	_ CALL ENDMSG('INVALID ACCOUNT NAME ' || NAME);
	_ CALL ENDMSG('INVALID ACCOUNT BALANCE ' || TYPE || BAL);
	_ SIGNAL ACCOPENFAIL;
		1
		2
			1
			2
			3
			4
			5
			6
			7
			8
			9
			A
			B
			1
			2
				SEQUENCES

EXCEPTS	_ ON INSERTFAIL JLC_ACC:
	_ CALL ENDMSG('FAILED TO INSERT ACCOUNT' || NAME);
	_ SIGNAL ACCOPENFAIL;
	_ ON GETFAIL JLC_ACC:
	_ CALL ENDMSG('FAILED TO RETRIEVE STORED RECORD' || NAME);
	_ SIGNAL ACCOPENFAIL;		

Actions can be either simple Huron statements or rule calls. They are arranged into a sequence activated by the condition found to be true. The logical sequence of actions has to follow their physical placement in the rule table. The number of

possible action sequences is limited by the screen size (around 8). The sequence length is limited by the number of 1 character sequence tags.

During execution, one of the evaluating rules or a database operation may signal an exception, e.g. database transaction failure, integrity or security violation, rule failure, or an application-specific exception. All exceptions may be captured by rules, examined and responded to in an appropriate manner.

As in Prolog, rules may be structured to act as validators, almost devoid of their procedural component, e.g. a rule checking the account maturity for a fund withdrawal (Cf. rule 3x) :-

```
JLC_TERM_TRANS_INVALID(TERMS_SO_FAR, MATURITY_TERM,
AMOUNT);
_
```

```
_ AMOUNT >= 0;                          Y   N   N
_ TERMS_SO_FAR > MATURITY_TERM;             Y   N
```

```
_ RETURN('Y');                                  1
_ RETURN('N');                          1   1
```

(iv)

or alternatively be used to perform purely functional activities, e.g. a rule calculating interest on the account deposit (Cf. 3xi) :-

```
JLC_CALC_INT_PAY(PREV_BAL, INT);
_
```

```
_ RETURN(PREV_BAL * (1 + INT));         1
```

(v)

the procedural aspects or Huron rules is clearly visible in previous examples (e.g. iii).

One of the serious limitations of Huron rules is their inability to specify actions to be performed before checking the condition, resulting in unnecessary fragmentation of rules. In certain cases this limitation could be circumvented by adding validation or trigger rules to the database itself rather than expanding the action precondition (e.g. checking the status of accounts before deletion should in fact be implemented as a validation rule in the account file rather than additional check in the top-level command-rule). Another possible solution to the rule rigid structure is to rely on its capability to rise and capture exception signals, so that it is the procedural components of the rule which determine the sequence of actions embedded in the exception sections.

Control. Huron rules are usually triggered by explicit calls rather than pattern-matching as in Prolog. Data driven processing, however, is still possible by maintaining derived fields, attaching validation and trigger rules to the database and screen tables, and automatic validation of cross-referenced tables. Although Prolog-like backtracking and associated variable unification are not available in Huron, rule processing may be controlled by traditional database transaction monitoring principles, i.e. transaction commitment and rollbacks.

User Interface. Huron user interaction can be organised via screen tables, menus and function keys. This area of Huron functionality wasn't extensively tested in the NBS experiment.

Summary. Huron facts are kept in a relational database system (i-ii), its rules are based on production rules, and thus, have a procedural semantics. The rule's components include its definition, conditions, actions and exceptions (iii). The rule structure, although rigid, allows the construction of logical and functional programs (iii-v). Huron permits either explicit procedural or implicit data-driven control, its exception handling and transaction monitoring enhance Huron control strategy.

5. Comparison and Discussion

Although both Huron and Prolog lean towards problem solving by rules, they do, however, represent two quite different approaches to rule-based modeling. This section compares the two methodologies (Cf. table 4).

Feature	Huron	Prolog
Rules		
Head	Procedure	Goal
Arguments	By-Value	By-Unification
Results	By-Return	By-Unification
Local Vars	Yes	Yes
Conditions	Simple Expressions	Subgoals
Actions	Calls	Subgoals
Exceptions	Yes	No
Logic	If-Then-Else-If	Or-And
Goal-Driven	Explicit	By-Match
Data-Driven	Event Rules	No
Foreign	Yes	Yes
Variables		
Arguments	Yes	Yes
Local	Yes	Yes
Global	Yes	No
Assignment	Procedural	Unification
Scoping	Dynamic	Lexical
Data Types		
Numbers	Yes	Yes
Symbols	Yes	Yes
Strings	Yes	Yes
Structures	No	Yes
Lists	No	Yes
Sets	Tables	Fact Collections
Tuples	Records	Facts
Fields	Flat	Structured
DB Operations		
Store	Insert	Assert
Retrieve	Get	By-Reference
Delete	Delete	Retract
Update	Replace	Retract/Assert
I/O Operations		
Read	No	Read
Write	MSGLOG	Write
Forms	Screen Tables	No
Utilities		
Maths	Yes	Yes
Strings	Yes	Yes
Dates	Yes	Limited
Time	Limited	Limited
Control		
Recursion	Yes	Yes
Loops	Forall	Fail-Retry
Meta	Yes	Yes
Transactions	Fail-Committ	Backtracking

Table 4 - Huron and Prolog features

Notation. Huron enforces the procedural interpretation of rules, which provides a natural extension of commercial languages like Cobol or C, thus becoming an attractive alternative to commercial programmers. The rule arguments, the logic of its conditions, the way it returns results, or accesses data tables, all follow the 3GL approach to programming. Prolog on the other hand has the logical foundation of its rule construction, forcing unusual design and programming methods, such as, pattern-matching, unification and resolution, all, requiring extensive training.

Data. Both systems encode their facts as relational tables. Huron, though, provides a flexible programming interface to its database and other standard database formats, it also offers an explicit data manipulation language. Prolog database, however, is usually RAM-based and it can be freely accessed by implicit reference. Such an approach to data storage is of very limited use for commercial applications. Nevertheless, some of the Prolog systems are extended to handle any relational database systems (e.g. Oracle or Ingress). As a result of Huron's strong database orientation, the set of available data structures are limited to relational tables and their primitive fields, Prolog systems offer a much richer collection of data types.

Rules. Prolog rules are very compact and uniform in their structure. Every rule sets to accomplish a specific goal and decomposes its logic into a number of subgoals. Each of the rule subgoals can be any Prolog logical expression combining the features of both conditions and actions, thus, simplifying the rule logic. Huron rules on the other hand, has a very rigid form, including the definition, conditions, actions and the powerful exceptions. The rule conditions are very crude allowing only the rule calls or the simple logical relations between variables and fields. Although, the conditions and actions give an impression of a decision table structure, the conditions can only emulate a simple if-then-else-if logic and the actions cannot be sequenced in arbitrary fashion but have to follow their physical order in the rule table. Rule size is also severely limited by the screen restrictions.

Control. Prolog control strategy is based on unification, resolution and backtracking, and although very sophisticated, it is quite difficult to master by non-experts. Huron offers a more traditional approach to controlling the program execution, familiar to those who use 3GLs, i.e. explicit rule calls with the transaction monitoring extensions. At the same time, Huron allows attaching rules directly to data tables, thus, providing powerful data-driven programming techniques existing in those rule-systems which support forward chaining, but not in Prolog.

Utilities. Both languages supply rich libraries of utility functions. Standard Prolog does not specify the utilities necessary to implement applications, however, majority of the commercial systems contain huge libraries of predicates and functions for dates, times, file manipulations, user-interface, sorting, etc. Open Prolog, which was used throughout the experiment, is a public domain system, and thus, had a bare library system allowing the minimum of the functionality. Conversely, Huron is a production business system offering rich library of functions.

Productivity. Throughout the course of experimental programming it was determined (Cf. table 5) that both Prolog and Huron solutions to NBS were of similar complexity and both required the similar number of rules (65 and 60 rules respectively). Yet, Prolog program required a much less time to implement than that in Huron, by nearly half the time (5 vs 10 hours).[1] The average programming productivity was 11 minutes per rule in Huron vs 5.5 minutes per rule in Prolog. Some of the reasons for the difference in programming productivity could be attributed to the Huron interface, which although user-friendly, was somewhat restrictive especially for the more experienced users.

	Huron				Prolog			
Task	**T**	**BD**	**AD**	**NR**	**T**	**BD**	**AD**	**NR**
Table Creation & Population	120	1	4	41	70	2	1	44
Account File Maintenance	100	4	2	5	60	1	1	7
Deposits & Withdrawals	365	5	5	14	185	3	3	14
Customer File Maintenance	not implemented				75	2	3	9
Interest Table Maintenance	not implemented				30	1	1	7
Account Types Maintenance	not implemented				25	1	1	9
Transfers	not implemented				90	3	2	7
Utilities	not needed				110	4	4	21
User Interface	165	3	7	6	not implemented			
Totals for Common Rules	585	5	5	60	315	4	4	65
Totals for Extra Rules	165	3	7	6	330	3	3	53
Totals for All Rules	750	5	7	66	645	4	4	118

Table 2 - Comparison of Huron vs Prolog productivity

T - Time in minutes; BD, AD - Subjective difficulty before and after implementation in 0-10; NR - Number of rules written

[1] It should be noted that the author had much more experience in Prolog rather than Huron programming. Although, the Huron experience was limited to a 2 day intensive programming course, the author has an extensive experience in a variety of languages and rule-based systems, thus, the bias towards Prolog is not significant.

Summary. Overall Huron was found to be a natural extension of traditional 3GLs, like C or Cobol. Huron's rule function was comparable to that of Prolog, although one assumes the Production other the Logical model. Being a commercial rather than research system, Huron has certainly better utility than Prolog, it has a rich development environment, access to commercial database systems, reporting facilities and a standard user interface building toolkit. The best features of Huron rules are the ability of exception handling, transaction monitoring and direct tables attachment, thus, providing data-driven processing. The greatest limitations were found in Huron user interface, rule structure, and the limitations imposed on the availability of rich data types.

References

Clocksin, W.F. and Melish, C.S., Programming in Prolog, 3rd Edition, Berlin, Heidelberg, Springer-Verlag, 1987.

Harmon, P. and King, D., Expert Systems: Artificial Intelligence in Business, New York, Wiley, 1985.

Huron for Application Developers, Amdahl Corporation, October 1991.

Appendices

A. NBS Requirement

The Naive Bank of Australia (NBA), the young and competitive bank of Australia, is developing a complete computer system to manage their retail operations. The system will provide its customers with a number of attractive bank products.

NBA customers will be able to either individually or jointly own a number of different accounts, i.e. savings, loan or term deposit accounts. Each account will allow its owners to perform deposits, withdrawals, transfers, balances. On the monthly basis the bank will also send its customers the statement of all transactions performed in that period of time. NBA saving accounts may be serviced by the bank tellers at any of the NBA branches, or via the cheques or ATM machines. At the NBA branch manager's discretion, the selected customers may also be granted a savings account overdraft. For all types of NBA accounts the interest is calculated monthly on the minimum monthly balance (or maximum liability for loan and overdraft accounts).

The interest rates are dictated by the international money market, but at this moment stands at 3% for savings up to $1,000, 10% up to $5,000, and 12% above $10,000. The loans and overdrafts are serviced at 9%, 12% and 15% for the liabilities of up to $5,000, $10,000 and $25,000, above that an interest of 17.5% is accrued. All customer transactions attract a standard bank fee which vary depending on the current level of government duties and taxes, at this moment of time is at 0.5% of the transaction value.

The bank keeps the transaction audit trails, and records the totals of its cash, cheque, and ATM transactions (which in fact are treated as special bank accounts). At the end-of-month processing the bank is also required to perform a number of integrity and audit checks to eliminate any possibility of tampering with its accounting and computer systems, thus, it checks that all NBA accounts balance, it detects suspect accounts and transactions, such as accounts with balances below their overdraft limits, loan accounts of which repayments increase the customer liability, accounts which are subject of great many small transactions which amount to big balances, accounts of which balance will not match the transaction amounts, etc.

NBA is seeking a contracting firm which would undertake the implementation of the prototype of its system in no more than 3-4 days. The prototype must be implemented using either 3GL (such as Cobol), 4GL (Ingres), rule-based system (e.g. Prolog), object-oriented system (C++) or some hybrid system (e.g. Nexpert). The final production will take in Amdahl Huron.

The contractors will have to design the NBS system in the spirit of the prototype tool they are going to use, and subsequently implement it in both Huron and the selected tool. It should be noted that both implementations should adhere to the original design. Nevertheless, those of the design options which prove to be impossible or too difficult to implement (Huron deficiency), or if Huron's platform could provide an improvement over the suggested design (Huron advantage), then the original design should be compromised and replaced with an altered version. Any such design decisions should be clearly documented.

The contractors should also produce the report which will compare Huron against the selected product of their expertise, but also against the principles of their design methodology and other tools which fit into such a design arena.