The Design of *j*-DREW : a Deductive Reasoning Engine for the Web

Bruce Spencer

Institute for Information Technology – e-Business
National Research Council of Canada
46 Dineen Drive, Fredericton, New Brunswick, Canada E3B 5A3
Bruce.Spencer@nrc.ca, http://www.iit.nrc.ca

Abstract. *j*-DREW is an easily configured, powerful deductive reasoning engine for first order, clausal logic written in Java and well integrated with the Web. A programmer with the ability to manipulate recursive data structures, such as commonly taught in university computer science programs, will be able to reconfigure the reasoning engine of *j*-DREW using its application programmers interface (API). *j*-DREW uses powerful and efficient techniques developed for competitive automated theorem provers, such as discrimination trees, sound unification, subsumption and flatterms. It can be deployed as part of a larger Java system, on a server or, with its small memory footprint, on a client. Three prototypes for definite clauses are considered: several variants of Prolog, a backward chaining RuleML engine and a proposed forward chaining deduction engine that interfaces to the Java 1.1 event model.

1 Introduction

j-DREW is a powerful, highly configurable reasoning engine for definite clauses or rules, with various deployment venues possible, including use on the Web. More important, *j*-DREW is an *idea* that includes a class of reasoning engines. The current paper deals only with definite clausal logic, since that logic expresses many examples and needs only semantic ordered resolution without factoring, essentially SLD resolution. The *j*-DREW components faciltate development of new engines for special purposes, freeing programmers from the complexity of handling the binding and unbinding of variables, yet offering good to very good performance. Choosing this flexibility results in removing the possibility of compilation.

The motive for developing *j*-DREW came from teaching Prolog programming, conventional programming and object oriented programming for many years. Most programmers with one or two years training can readily build recursive data structures, including trees. They can also perform more complex operations that involve adding and deleting from a tree. Most students of Prolog have no difficulty understanding the Prolog goal tree. But it is very uncommon to ask students to write programs in C, C++ or Java that build and manipulate Prolog goal trees, probably because the task is complicated by a number

of subtle points involving variable bindings: creating unifying substitutions, applying substitutions to some part or to all of the tree, composing substitutions, creating fresh variables, performing occurs-check during unification, unbinding of variables during backtracking, and controlling the search. Courses that cover these topics at the programming level are specialized courses on implementations of logic systems, not general data structures or artificial intelligence courses. However, if one restricts the discussion from first order logic and allows only propositional logic, avoiding all issues of handling variables, the propositional Prolog engine is no more than a backtracking search, shown in Figure 1, and can be presented in a data structures classroom with a reasonable chance of achieving comprehension by the better students.

```
%Let initialGoal be the given goal, initially open
Tree proofTree = new Tree(initialGoal);
Stack choicePoints = new Stack();
while(true){
  if(proofTree.hasNoOpenGoal())
     halt('success');
  else {
     Goal g = proofTree.selectOpenGoal();
     g.createMatchingClauseList();
     if(g.hasMoreMatchingClauses()){
        Clause c = g.nextMatchingClause();
        GoalList children = c.bodyOfClause();
        g.addChildren(children);
        choicePoints.push(g);
     }
     else {
        chronologicalBacktrack();
     }
  }
}

void chronologicalBacktrack(){
   while(!choicePoints.empty()){
      Goal g = choicePoints.pop();
      g.removeChildren();
      if(g.hasMoreMatchingClauses())
         return; %exit
         %because there is something to try
   }
   %no proof can be found
   halt('failure')
}
```

Fig. 1. Propositional definite clause reasoner with backtracking search

To understand the code in Figure 1, let Tree be the class of proof trees, where each proof tree can tell that it either has no open goal remaining, or can provide a deepest open goal. An intial open goal is provided when the tree is created. Each goal maintains its own list of those definite clauses having heads that match it; a goal can tell if it has more matching clauses, and can provide the next matching clause on demand. The body of that chosen matching clause is a list of goals, and these are added to the tree as the children of the goal; then the goal is no longer open, and these children are new open goals. For backtracking, choicePoints is a stack of goals in the proof tree. A goal is pushed onto this stack whenever a goal chooses its children. Backtracking is forced if some selected goal has no matching clauses, and then some earlier choice of a goal list for a goal must be undone. The earlier goals are on the stack in reverse chronological order – the top of the stack is the most recent choice.

Note that while this is essentially SLD resolution, there is an explicit data structure constructed which is different from the list of goal clauses created by SLD resolution. It is also different, perhaps bulkier, than the reasoning procedure in most Prolog implementations, such as *establish* in [5]. That function builds an implicit tree, because a Prolog programmer's objective is to generate answers rather than to create proof trees.

The idea that came to be j-DREW was to provide programming tools in the form of a Java application programmers interface (API), that would address the problems mentioned: that of complicating the previous explanation by combining it with an explanation of variable bindings. The j-DREW API deals with the variable bindings "behind the scenes," so that the code in Figure 1 is a definite clause reasoner for first order logic as well.

In the following sections, the ideas behind the j-DREW architecture are explained, organized according to (1) the basic components, some of which were derived from competititve theorem provers, (2) the programmer's interface to the various configurations possible, and (3) some deployed and designed prototypes. Lastly, comparisons with other reasoning systems, and future prospects are offered.

2 Basic Components

We use standard definitions [2] for predicate symbol, function symbol, atom, literal, substitution, unifier and most general unifier. In this paper, lower case letters $p, q, r, ...$ (and with subscritps) refer to predicate symbols or to whole atoms, and letters $f, g, h, \ldots$ (and with subscritps) refer to constants and function symbols, while upper case letters $X, Y, \ldots$ (and with subscritps) refer to variables. A definite clause is an ordered disjunction of literals, the first of which, called the head, is positive and the rest are negative, *e.g.* $p \vee \neg p_1 \vee \ldots \vee \neg p_n$.

At its kernel, j-DREW is a theorem prover for definite clauses of first order logic. The data structure for definite clauses is important since it affects the efficiency of key operations, such as unification. The negation signs of literals in definite clauses are not stored, because they are known from their positions in

the clause: the second through last literals are negative. Because symbols can have long names and can occur in many parts of a formula, it makes sense to store these names in one place. A symbol table is an array of print names and arities of symbols. For instance in the atom $p(f(g_1), h, h(g_2))$, p has arity 3, f has 1, and both g_1 and g_2 have arity 0. There are two different symbols h: one with arity 0 and the other with arity 1. Sometimes symbols are juxtaposed with the arity, as $h/0$ and $h/1$, to distinguish them. A suitable symbol table for this atom appears in Figure 2.

Atoms and clauses are represented as flatterms[3], which for most applications are more space efficient than linked lists or pointer-based trees of symbols. Our flatterm type is a pair of arrays: symbol and length, both composed of short integers. The length array is not strictly needed, but simplifies the code at the cost of doubling the storage requirement, which so far has not presented a problem. Each predicate symbol, function symbol and variable in the formula has a position in these arrays. We deal with variable free formulas first. The first array of the flatterm, symbol, contains the symbol table index for predicate or function symbol in this position. The second array, length, contains the length of the subformula (subterm) beginning at this position. For example, in Figure 2, the subterm g_2 begins at position 6, so the flatterm's symbol array contains in position 6 the symbol index for g_2, which is 4. Since g_2 is a subterm of length 1, there is a 1 in the flatterm's length array for position 6. Meanwhile $h(g_2)$ is a subterm of length 2 beginning in position 5.

Formulas with variables can also be represented with flatterms. In j-DREW we are interested in clausal logic where variables have scope that is across a clause and no further. The variables in each clause are numbered according to their first appearance with negative indices starting at -1, and there is no confusion with variables from other clauses. All occurrences of the same variable in a flatterm have the same negative number in the symbol array. For example, Figure 3 shows the flatterms for $p(f(h(X)), h(Y), f(X), Y)$ in which -1 is the index for X and -2 is the index for Y. These are always numbered according to their appearance in left-to-right order.

Symbol Table

	name	arity
1	p	3
2	f	1
3	g_1	0
4	g_2	0
5	h	0
6	h	1

Flatterm for $p(f(g_1), h, h(g_2), g_1)$

	symbol	length
1	1	7
2	2	2
3	3	1
4	5	1
5	6	2
6	4	1
7	3	1

Fig. 2. Symbol Table and Flatterm

Flatterm for left
$p(f(h(X)), h(Y), f(X), Y)$

	symbol	length
1	1	9
2	2	3
3	6	2
4	-1	1
5	6	2
6	-2	1
7	2	2
8	-1	1
9	-2	1

	position	side
-1	6	right
-2	5	right

Flatterm for right
$p(f(W), h(f(g_2)), Z, Z)$

	symbol	length
1	1	8
2	2	2
3	-1	1
4	6	3
5	2	2
6	4	1
7	-2	1
8	-2	1

	position	side
-1	3	left
-2	5	right

Fig. 3. Flatterms and variable substitutions

While j-DREW 's variable handling strategy may appear obtuse to programmers of definite clause reasoning engines and Prolog system programmers, recall that one of the objectives of j-DREW is to be a common API for forward and backward reasoning, even allowing mixed search strategies, and to build explicit data structures. As such there is no attempt in the current prototype to take advantage of structure sharing between clause instances, used heavily in Prolog but often not used in theorem proving and in forward reasoners such as JESS [4].

A variable substitution in j-DREW is also represented as two arrays, position and side, of short integers. A substitution entry is created for each distinct variable in the flatterm. Substitution arrays are created only in response to a request for unification of two flatterms, called left and right, and a value given to a variable is always a subterm in one of the two flatterms. For instance, in Figure 3 the value given to X in the left atom is g_2. This is shown in the left substitution arrays in position -1 for X, where the values are 6 and right, indicating the subterm at position 6 in the right atom, which is g_1. Note that the result of applying either of given substitutions would be $p(f(h(g_2)), h(f(g_2)), f(g_2), h(g_2))$. Thus these substitutions unify the two flatterms.

Note that this binding strategy is called a local shallow binding, which is not usually used in backtracking search. Local bindings need no global list of variables and values, but variables are always referenced with respect to a clause. An advantage to local binding is that no large, sparsely populated array of global variables needs to be created. Although this is not an issue for backtraking search because the variables come and go in Last-In-First-Out (LIFO) fashion, it arises in other types of search. With shallow binding, a variable's current replacement is available by looking it up in a list that is always up to date, requiring only a shallow search for the value. With deep bindings, new replacements do not up-

date the the values in the existing replacements, so when looking up the existing value one or several searches may be required to see if any of its embedded variables have been bound. Deep bindings are usually used in backtracking search where new bindings are often undone in LIFO fashion.

j-DREW uses sound unificiation.

The main data structure in j-DREW is the DCTree, or definite clause tree. Each node in the tree contains a flatterm for the atoms in a definite clause, where the signs of these atom are defined implicitly by position, as mentioned before; the first atom is positive and others are negative. For simplicity, the second and subsequent atoms in flatterm for the node is called a goal list. Each atom in the goal list is called a goal, and has at most one child node. A goal has a child if some clause has been chosen to solve that goal. In that case, the instance of the definite clause in the child node is such that its first atom is usually identical to the atom for this goal. In some cases the child atom is an instance of goal atom, and then an operation to make them identical is scheduled to occur later as part of variable binding propagation, described next.

In j-DREW the unification process between two atoms, where one is a goal and the other is the head of a definite clause, is always performed between two different clauses. For each clause an array is created to store values, one for each of its variables. The unification algorithm determines these values, so that by replacing the variables by their values, the two atoms in question become identical, if this is possible, or the algorithm returns a failure notification if this is not. Each of these values is called that variable's *binding*. The process of creating a new flatterm with the variables replaced by their bindings, if any, is called *binding application*. Notice that since variables are clause-local, values in one flatterm are not automatically transfered to another flatterm. In j-DREW the only way that a variable's value can be propagated from one node in the tree to another is by using unification to create variable bindings, and then using binding application to create a new flatterm.

Propagation occurs this way: Suppose a variable's value has changed in an adjacent node, say a child C of some node N of interest, and that change should be propagated to N. See Figure 4. In this example, a segment of the flatterm for N is designated for the goal p_1. Also a segment of the flatterm for C is designated for the head q of a clause. At the moment p_1 and q are different atoms, but p_1 is an instance of q. A unification step between q is p_1 is performed, resulting in some variable bindings for N's flatterm. When these are applied, a new flatterm for N is created, and it becomes the new flatterm, replacing the previous one.

At some times in the search, described later, variable bindings are chosen and propagated, and later may be found not useful. Then the effect of variable binding should be removed. If these binding effects are always done and then undone in last-in first-out order, or LIFO, a stack of flatterms is a convenient structure. For this reason, the current version of j-DREW allows only LIFO changes to the variable bindings. Since this is consistent with backtracking search methods, it is a reasonable restriction. Thus each node has its own stack of flatterms.

Returning to the example, before the old flatterm for N is discarded, it is pushed onto N's stack of flatterms. Then the new flatterm is create and becomes N's flatterm. Subsequently, more variable bindings may be propagated to N from the child D. If later it becomes necessary to undo both sets of variable bindings, it is a simple matter to pop the stack until the appropriate flatterm is reached.

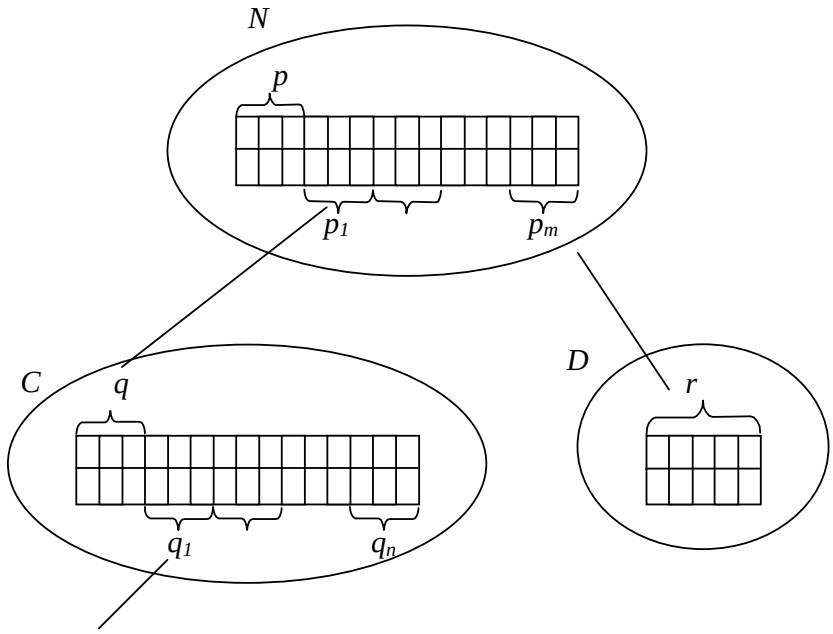

Fig. 4. Nodes in a DCTree, showing flatterms. C and D are children of N.

j-DREW uses techniques from Java where these simplify its work, making it easier for Java programmers to incorporate j-DREW into their code. In the variable propagation example, a Unifier object is created for a specific goal pair of nodes in the DCTree. This object is capable of creating new flatterms for either of these nodes. A Unifier also tells whether or not it is possible to unify these two atoms. The programmer can ask the Unifier whether unification is possible, without actually creating the instances; in some cases it might be important to separate these tasks in case one is only interested only in the first and does not want to invest time to do the second. Also the programmer can ask to have only one of the new flatterms created, allowing one-way unification, useful for finding instances of a given literal.

Common in Java programs is the use of Iterators. An Iterator is an object that gives access to each of a sequence of other objects, through a simple interface. The method `hasNext()` returns a boolean result, telling whether or not there is another object in the sequence that has not yet been given, or visited. The

method `next()` returns that next object, and advances the Iterator so that subsequent calls to `next()` return subsequent elements of the sequence. In the case of j-DREW once a goal is identified, an Iterator of Definite Clauses can be created, where each Definite Clause in the sequence is an instance of some given clause, chosen so that its head is an instance of the goal. These clauses give us the goal lists that can later be attached to the goal's occurrence in the DCTree. One can see that the code in Figure 1 uses the idea of an Iterator, in the calls to methods `hasMoreMatchingClauses()` and `nextMatchingClause()`. These are choices that arise in the search, and it is also an example of the one-way unificiation previously mentioned.

Consider further the task of creating all of instances of the given clauses whose head is an instance of a given goal. Clearly one could just consider each of the given clauses in turn, attempting to unify each head with the goal. But something much faster is possible, based in McCune's discrimination trees. Given the usual prefix-ordered string representation of a set of terms or atoms, a discrimination tree is obtained if one combines prefixed as much as possible. An *imperfect* discrimination tree results if one first replaces all variables by a new symbol, say $*$. See Figure 5 for the imperfect discrimination tree arising from the clause heads $p(f(g_1), h, h(g_2), g_1)$ and $p(f(h(X)), h(Y), f(X), Y)$. In the resulting tree structure, each internal node contains either $*$ or a function or predicate symbol that occurs somwhere in the clause heads, or a $*$. If two atoms have the same symbols in positions 1 to n, but a different symbol in position $n+1$, then in the discrimination tree the the node corresponding to the first n positions will have (at least) two children.

The discrimination tree is used to find atoms that are likely to unify with a given goal atom. At each leaf of our discrimination tree we store the clause whose head corresponds to the path from the root. Our goal is to hasten the search for clauses whose head closely matches a given query atom. Once found, since the discrimination is imperfect, a later unification step is needed, so the matching process is called pseudo-unification. One can find all atoms likely to be unifiable with a given query atom by traversing the discrimination tree downward from the root, following the branches that correspond to the symbols in the query. What follows is a general description; we refer the reader to McCune's exposition for details [6]. When a $*$ is found in the tree, then one skips over one symbol in the query, and, if present, over its arguments. For example, looking for generalizations of $p(f(h(f(g1))), h(g_1), f(g_2), h)$, one enters the discrimination tree from the root, following the p, f and h branches until $*$ is found in the tree. One skips over the g_2 in the query, to account for one subterm skipped. Thus $p(f(h(X)), h(Y), f(X), Y)$ will eventually be found as a generalization of $p(f(h(f(g_1))), X, Y, Z)$.

The query atom also has its variables replaced by $*$. When one encounters a $*$ in the query during matching, one skips down all branches in the discrimination tree that correspond to one subterm. This gives a possibly large number of points in the tree to explore next. For instance, if one searches for all of the heads pseudo-unifiable with $p(f(*), *, *, *)$, one would follow the discrimination tree

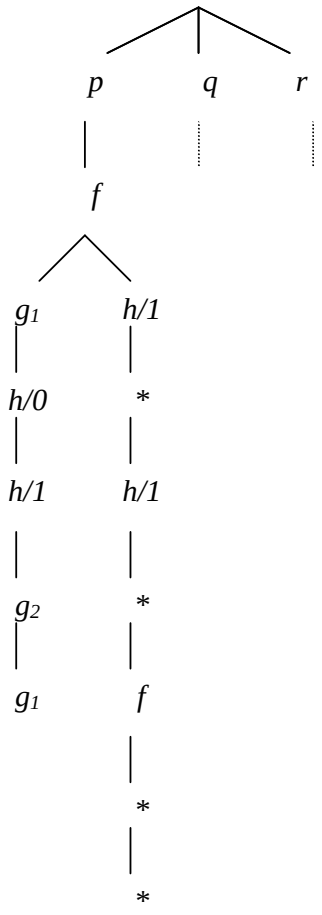

Fig. 5. Discrimination Tree containing $p(f(g_1), h, h(g_2), g_1)$ and $p(f(h(X)), h(Y), f(X), Y)$

from the root to p, then f, and would then follow two branches that correspond to the $*$ at the current position in the query goal: over the g_1 to the $h/0$ down the left branch, and over both the $h/1$ and the $*$ to the lower $h/1$ down the right branch. Eventually both leaves would be identified as pseudo-instances of $p(f(*), *, *, *)$.

The use of discrimination trees effectively gives every argument indexing, unlike some Prolog implementations that use only first argument indexing. This is important if searching for RDF triples [7] in the Semantic Web, as a query might be set up to find all RDF triples with a certain second or third argument, without specifying the second argument.

3 Programmer's Interface to *j*-DREW

The reasoning procedure in Figure 1 gives only the first solution to a goal. We advocate to use an iterator to generate a series of solutions to a goal, where the state of the computation is stored in the iterator. Thus there is not necessarily just one top-level query that can be asked, but rather for each goal that one wants to solve, a separate tree-producing iterator object is created to generate solutions on demand. Several of these iterators can be active at once. The code in Figure 6 provides hasNext() and next() functions. Notice that hasNext() does all the work. The chronologicalBactrack function is unchanged.

To use the solution iterator in Figure 6 provides the essential definition of an Iterator object, which we will call a SolutionIterator. To build and display the sequence of proof trees for the goal, one would create a object of type SolutionIterator and use the hasNext() and next() methods as follows:

```
Iterator solver = new SolutionIterator(topLevelGoal);
while (solver.hasNext()) print(solver.next());
```

4 Related and Future Work

Currently the j-DREW API provides a basis on which to build a system functionally equivalent to Prolog, except that its implementation is not compiled, it does not use structure sharing and it creates explicit solution trees. A parser has been added to j-DREW to handle RuleML [1]. But j-DREW was designed from the standpoint of a basic resolution engine, and this allows us to use the basic levels of j-DREW to provide an API for forward reasoning systems. On receipt of a new conclusion, the reasoning engine would trigger the creation of subsequent conclusions, starting a chain of activities.

In particular we plan to integrate j-DREW into the Java 1.1 Event architecture. In Java 1.1, a user-interface event, such as a button-click, is attached to an event handler object, called a listener. A particular method in the listener is called each time the user-interface event occurs. For our button example, the event handler is called a ActionEventListener, and each time the button is

```
boolean foundNext = false;
boolean failed = false;
boolean enteringFirstTime = true;
Tree proofTree = new Tree(initialGoal)
Stack choicePoints = new Stack();

boolean hasNext(){
  if(foundNext)
    return true;
  else if (failed)
    return false;
  if (not enteringFirstTime)
    if (not chronologicalBacktrack())
      failed = true;
  boolean succeeded = false;
  while(not failed and not succeeded){
    if(proofTree.hasNoOpenGoal()){
       succeeded = true;
    }
    else {
       Goal g = proofTree.selectOpenGoal();
       g.createMatchingClauseList();
       if(g.hasMoreMatchingClauses()){
          Clause c = g.nextMatchingClause();
          GoalList children = c.bodyOfClause();
          g.addChildren(children);
          choicePoints.push(g);
       }
       else if (not chronologicalBactrack()){
          failed = true;
       }
    }
  }
  foundNext = succeeded;
  return foundNext;
}

Object next(){
  if(not hasNext())
    throw new NoSuchElementException();
  foundNext = false;
  enteringFirstTime = false;
  return proofTree;
}
```

Fig. 6. Propositional definite clause reasoner as a Java Iterator

pressed, the ActionPerformed method in the ActionEventListener is called. That method contains the code that responds to the event.

Event-condition-action rules, also known as reaction rules, are commonly used in expert systems, agent systems and messaging systems, or wherever reactions to asynchronous events are required. A forward reasoning j-DREW system would be implemented as a Java EventListener, reacting to the addition of a new conclusion, by inferring new information and possibly raising new events as a result.

There is interest in buiding systems for the Semantic Web using two kinds of rule-based systems: deductive and reactive [8]. We are committed to using j-DREW not just with a backtracking search, but rather to allow possible any search that the programmer might find convenient for a given problem, and possibly allowing simultaneously several search techniques. It is still an open question how best to design the variable binding strategy to accomodate both backtracking and non-backtracking search.

References

1. Harold Boley and Said Tabet. RuleML Homepage, 2002. http://www.dfki.uni-kl.de/ruleml.
2. Chin-Liang Chang and Richard Char-Tung Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, New York and London, 1973.
3. Hans de Nivelle. Data Structures for Resolution. Technical report, Max Planck Institut für Informatik, 66123 Saarbrücken, Germany, 1999.
4. Ernest Friedman-Hill. Jess, the Expert System Shell for the Java Platform, 2002. http://herzberg.ca.sandia.gov/jess/.
5. David Maier and David S. Warren. *Computing with Logic: Logic Programming with Prolog*. Benjamin/Cummings Publishing Co., Menlo Park, CA, USA, 1988.
6. W. W. McCune. Experiments with discrimination-tree indexing and path indexing for term retrieval. *Journal of Automated Reasoning*, 9:147–168, 1992.
7. RDFCore Working Group (co-chairs Dan Brickley and Brian McBride). Resource Description Framework (RDF) / W3C Semantic Web Activity, 2002. http://www.w3.org/RDF/.
8. Michael Schroeder(msch@soi.city.ac.uk) and Gerd Wagner (G.Wagner@tm.tue.nl), editors. *International Workshop on Rule Markup Languages for Business Rules on the Semantic Web*. http://www.soi.city.ac.uk/∼msch/conf/ruleml/, Chia, Sardinia, Italy, June 14 2002. In conjunction with the First International Semantic Web Conference (ISWC2002).