Knowledge Bus: Generating Application-focused Databases from Large Ontologies

Brian J. Peterson
SAIC
peterson@cs.umbc.edu

William A. Andersen
US Department of Defense and
Department of Computer Science,
University of Maryland, College Park
andersen@cs.umd.edu

Joshua Engel
SAIC
engel@cs.umbc.edu

Abstract

As well as being useful for semantic integration of information systems, large shared ontologies may be applied as specifications for new systems. In this paper we discuss the design of Knowledge Bus, a system which generates information systems - databases and programming interfaces - from application-focused subsets of a large ontology (Cyc). The generated systems are semantically interoperable at the level of their shared representations. In addition, constraints and axioms contained in the ontological specification are encapsulated in the generated systems by an object-oriented (Java) API. This API makes the databases easily accessible to application programmers while preserving the intended semantics of the ontology.

1 Introduction

The use of a shared vocabulary or ontology to enable semantic interoperability of existing databases and other software is gaining acceptance. Even greater benefits can be achieved by using the same ontologies as formal specifications for new systems. The costs of reverse-engineering the semantics of legacy systems and expressing those semantics in terms of an ontology are high. By building information systems from a formal foundation which is shared across many such systems, we hope to reduce the effort involved in ensuring interoperability between them.

A fundamental problem in building cooperative information systems is that it is generally difficult to understand the semantics of any system you did not build. The use of large, general ontologies has been proposed as a solution to this problem. By mapping the semantics of independently developed components to concepts in an ontology, the hope is to use the ontology, along with appropriate middleware, as an interlingua enabling communication between the systems.

Rather then attempting to reverse-engineer a shared model of the databases, we propose an alternate solution: generate the databases (including APIs) directly from focused subsets of a large, general purpose ontology. Because these systems are derived from a common representation, interoperability at the semantic level can be achieved without an expensive level of hand-crafting. Extracting only the subset of the ontology needed to support representation and reasoning in a focused application domain enables the resulting systems to be much smaller, more efficient, and more manageable than if the entire ontology were present in each system.

Using traditional database construction techniques, the cost of programming and maintaining a complex schema and thousands of types, rules (views), and integrity constraints would be prohibitive. This process of automatic generation will make the generated systems easier to maintain.

For the past two years, we have been developing a prototype system, called Knowledge Bus, to explore this approach to constructing information systems. Knowledge Bus has been used to develop databases for the Department of Defense which are now in operational use in complex decision-support applications.

We start with a very large ontology (Cyc) which uses a first-order based language (CycL) with an expressiveness which subsumes relational and object-oriented models. The system for generating databases consists of four major components: The sub-ontology extractor identifies a domain-relevant section of the ontology. The logic program generator takes the extraction and translates it into a logic program which can be evaluated by a deductive database query engine such as XSB ([Sag96]).

Proceedings of the 5th KRDB Workshop
Seattle, WA. 31-May-1998
(A. Borgida, V. Chaudhuri, M. Staudt, eds.)

http://sunsite.imforkatic.rwth-aachen.de/Publications/CEUR-WS/Vol-10/

The API generator takes the logic-based model and exposes it to application developers as an object model through strongly typed object-oriented APIs (implemented in Java). The runtime system supports access to the generated databases.

2 Cyc and XSB

We chose to start with Cyc ([Guh94]) as our ontology because of its large size and broad coverage: 500,000 axioms (rules and assertions) and 40,000 constants. Whatever its merits as an ontology, Cyc is not a good choice for a database system. First, it lacks the ACID properties one would expect from an industrial-strength DBMS. More seriously, we did not view the Cyc inference engine as an appropriate query processor for a database system because it is based on classical first-order semantics. We chose the XSB system ([Sag96]) as the basis for the underlying database because it evaluates logic programs according to the Well-Founded Semantics (WFS) ([Gel91]) and has a fast and efficient inference engine. XSB has interfaces to commercial RDBMS systems, which will enable us to build transaction and recovery services in future implementations.

2.1 Cyc

Cyc is a FOL-based ontology which maintains its axioms in conjunctive normal form (CNF). Its features include a context hierarchy, an event calculus and sub-abstraction temporal model, and an argumentation-based nonmonotonic reasoning formalism. Reference [Guh94] for more details on Cyc.

2.1.1 The structure of the Cyc ontology

The Cyc ontology is organized into a set of contexts, called *microtheories* or *Mts*. Each Mt consists of a set of axioms and a set of domain assumptions[1]. Mts are related in Cyc by the predicate `genlMt`, which imposes a lattice on the Mts in the Cyc KB[2]. If `A` and `B` are Mts and `genlMt(B,A)` then all axioms true in `A` are also true in `B`.

Concepts or types in the Cyc KB are called *collections*[3]. Subsumption in Cyc is explicit and is indicated by the relation `genls`. Each constant in the Cyc KB is a member of one or more collections. Membership is indicated by the relation `isa`. Some Cyc collections (e.g., `CycSystemInteger`) denote "primitive" types. The primitive types ground out the representation of Cyc concepts into data types directly representable on the underlying hardware.

Relations and functions in Cyc are strongly typed. That is, for a k-ary predicate $p: T_1 \times \ldots \times T_k$ and an instance of the predicate $<a_1, \ldots, a_k>$, the following must be provable: $\texttt{isa}(a_i, T_i)$, $1 \le i \le k$, where the a_i are terms and the T_i are collections (types) in the KB. Functions are defined analogously.

2.1.2 Nonmonotonic reasoning in Cyc

Cyc's non-monotonic formalism is centered around an argumentation axiom, where φ is considered to be true if for each argument concluding $\neg\varphi$ there is an argument for φ which is preferred. Preference for one argument over another can be determined by various methods. One such method is to see if an argument relies on an invalid default inference, which can be establish using Cyc's `ExceptWhen` and `ExceptFor` predicates. `ExceptWhen(`φ`,` ψ`)` declares that ψ can be used in an argument except when φ can be proven. `ExceptFor(`τ`,` ψ`(x))` is similar, saying that ψ can be used in an argument except for `x` bound to term τ.

2.1.3 Cyc's temporal model

Cyc's event calculus temporal model is based on the `holdsIn` predicate, where `holdsIn(E,`φ`)` has the intended interpretation that φ is true during the time bounds of `E` (`E` is something with temporal extent, like an event, a person, or just a time interval). The sub-abstraction mechanism works by relating two constants together, asserting that one is a temporal portion of the other. So `FredTheStudent` would be a sub-abstraction of `Fred`, representing `Fred` from 1990 to 1996.

2.2 XSB

XSB is a Prolog-based logic programming system being developed at the State University of New York, Stony Brook. It evaluates according to the WFS using tabling and an SLG-WAM ([Sag96]) implementation, and has an impressive speed, coming within a small factor of commercial Prolog systems ([Sag96], [Swi94]).

2.3 WFS/XSB instead of FOL/Cyc

We believe that classical first-order semantics is inappropriate for a deductive database because of its computational complexity, inability to define desired relations (like the transitive closure of an arbitrary relation), and because of its treatment of negation. [Pry88] and [She88] discuss various semantics for negation; [Aho79], and [Imm89] discuss the expressiveness and computational complexity of classical semantics and first-order logic.

Our goal is to use the WFS because it has a polynomial data complexity, it can define desired relations (like transitive closure), it treats negation in a way that we con-

[1] Domain assumptions are axioms which constrain the meaning of all axioms occurring in an Mt. If φ is a domain assumption of Mt `C`, then, for all axioms ψ in `C`, ψ is equivalent to $\varphi \rightarrow \psi$ when "lifted" from or removed from the context of `C`. We currently make no use of this information.

[2] Any subset of the Mts in the Cyc KB is guaranteed to have a maximal element (the Mt `BaseKB`) and a minimal element (the Mt `BrowsingCntxt`).

[3] We will use the terms "collection" and "type" interchangeably.

sider appropriate for a deductive database, and it has equivalence results with major non-monotonic formalisms ([Abi95], [Gel91], [Pry88], [Pry91], [She88]). We decided on XSB because it is robust, dependable, and fast ([Swi94]).

3 The sub-ontology extractor

The database specification process begins by extending the Cyc ontology with concept definitions for a particular application domain. The job of the sub-ontology extractor is then to extract from the ontology all of the concepts required to support representation and reasoning in the application domain.

3.1 Starting the process

We begin by specifying a single Mt, *C*, with respect to which extraction is performed[4]. From *C* we obtain a "seed" set, S_0, of collections which are defined in *C*. S_0 is obtained by syntactically examining the axioms present in *C* and gathering up any collections (instances of `Collection`) which appear.

Informally, the purpose of the extractor is to identify all axioms which might contribute to proofs about instances of concepts in the seed set S_0. The intuition is that the user has specified in *C* (and thus in S_0) all of the concepts which are of interest, and in not interested in (details of) concepts not explicitly specified in S_0.

3.2 Fixpoint relevance computation

The computation proceeds with the application of two operators, Π and Γ. The Π operator maps a set of collections to a set of predicates. The Γ operator maps a set of predicates to a set of collections. They are defined as follows:

$$\Pi(S)=\{p|p:T_1\times\ldots\times T_k \wedge \exists i[T_i\in S \wedge \neg primitive(T_i)]\}$$
$$\Gamma(P)=\{T|p:T_1\times\ldots\times T_k\in P \wedge \texttt{genls}(T_i, T)\}$$

Starting with the seed set S_0, we apply $\Pi(S_0)$ to obtain a set of predicates P_1 involving collections in S_0. We then apply $\Gamma(P_1)$ to obtain a (new) set of collections S_1, and so on until a fixpoint ($S^*=S_{i+1}=S_i$, $P^*=P_{i+1}=P_i$) is reached:

$$P_0=\varnothing$$
$$P_{i+1}=\Pi(S_i)$$
$$S_{i+1}=\Gamma(P_i)$$

At the end of the computation the sets S^* and P^* contain the sets of "relevant" collections and predicates, respectively. We then add to P^* the set of functions whose signatures include only collections in S^*. Finally, we collect the set of axioms in all Mts accessible from *C* which involve only predicates occurring in P^*.

Because the operators Π and Γ are monotonic, the algorithm is guaranteed to reach a fixpoint. However, the fixpoint may be infinite due to the presence in Cyc of "collection denoting functions" (CDFs). We prevent this condition by only considering ground CDF instances during the fixpoint computation. This guarantees that the fixpoint is reached after at most the size of the Cyc `genls` hierarchy.

3.3 Justification for the algorithm

One potential weakness of this approach is that it makes the (unfounded) assumption that all collections are disjoint. An example will serve to clarify this point. Consider the clause $\neg$`p(X,Y)`$\vee\neg$`q(Y,Z)`$\vee$`r(X,Z)`. Say the fixpoint computation has already identified `p:`$T_1\times T_2$ as relevant. Intuitively, we would also like `q:`$T_3\times T_4$ to be relevant to enable the inference of `r(a,Z)` given `q(a,Y)` for some constant `a`. If `genls(`T_2`,`T_3`)` then we have what we want by definition of the operator Γ. If however, T_2 and T_3 are unrelated by the `genls` relation, we have a problem - we have no way of knowing a priori if some constant, when substituted for `Y`, is not both of type T_2 and of type T_3. The current algorithm would not select predicate `q` unless there were an independent reason for doing so. To correct this we could incorporate constraint information in Cyc about the relative disjointness of collections. We plan to explore this improvement in future work.

The question remains - does this algorithm compute anything reasonable? This is partly an empirical issue. However the use of the fixpoint computation ensures that we have not left anything important out.

The algorithm has several advantages over the approach of [Swa96]. First, it does not rely on heuristics to identify relevant concepts. Second, their work applied only to an ontology with taxonomic structure (no axioms or relations). Our method is general in that it can be applied to arbitrary KBs (such as one based on description logic) which share the basic subsumption and typing features of Cyc.[5]

4 The logic program generator

The goal of the logic program generator is to translate the extracted sub-ontology into a logic program that can act as an efficient query evaluator. We decided that the original first-order logic (FOL) based language with classical semantics was not appropriate; our goal is to use a logic programming language with the Well-Founded Semantics (WFS). We were unable to design a translation to the WFS (due to our chosen temporal model, explained be-

[4] It is always possible to supply a single context. If multiple contexts are specified, then a single temporary context is constructed which is the union of the specified contexts.

[5] Note that our use of Cyc contexts is not essential but only serves as an additional focusing mechanism. The algorithm could be applied to KBs with no such concept of context or to KBs with alternate context mechanisms.

low), instead settling for Stratified. The resulting translation is evaluated with the XSB system. Our goal is that the logic program resulting from the translation would have a bounded term size. In our first version, to ensure a finite domain we removed function symbols and the rules that used them.

4.1 Basis of the Translation

The idea behind the translation is straight forward: keep the horn CNF and do what you can with the non-horn clauses. The non-horn clauses are not completely lost to us, since we can use them as integrity constraints on the database. We can also convert the clauses with more than one positive literal into rules with negative antecedents. For example, the clause $p \wedge q \wedge \neg r$ could be treated as

```
ic1 :- r,¬p,¬q.
p   :- r,¬q.
q   :- r,¬p.
```

where `ic1` is used as an integrity constraint on the database.

We find that in these cases (where a negative dependency is generated) it is often desirable to chose one rule form and dismiss the others. This is because, in the absence of other information, the WFS will assign 'unknown' as a truth value to the literals involved . In the example, if the rule concluding `q` is removed, then the other is able to conclude `p` (given `r` and the absence of a `q`). This provides part of the inferential capability of the original non-horn clauses, and may be closer to what the person intended when they added the original rule to Cyc: A rule with negative antecedents is converted to the non-horn CNF by Cyc. So if the rule was originally $r \wedge \neg q \rightarrow p$, then the person may have intended this rule as a way of concluding `p`, and not really as a way of concluding `q`.

Because Cyc converts input rules into CNF clauses, the only source of negative subgoals in our rules is from this stage of the translation.

In our initial application of this process (generating a deductive database from Cyc), the extraction from Cyc identified 1531 collections, 1267 predicates, and 5532 CNF clauses as relevant to our subject domain. Of these 5532 clauses, only 128 had no positive literals and 84 had 2 or more (about 4% of the extracted clauses). A horn clause could conclude a literal which we consider to be inappropriate for a rule. For example, a rule concluding `X<Y` is not very interesting for a rule, but fine for an integrity constraint. There were 653 horn clauses which we identified as constraints. The remaining 4667 (horn) clauses we used as rules.

4.2 Stratification

Because of the limitations imposed by our temporal model (see below), the rule base needs to be stratified. To accomplish this, we used a few heuristics for breaking negative cycles, but the process was still interactive. The heuristics were straight-forward, removing rules which conclude a literal that commonly appears as a subgoal (like `isa` and `genls`).

In our first application, by this time in the translation process there was 5848 rules. To stratify this rule base, 352 were removed (307 by heuristics).

4.3 Translating the Temporal Model

We settled on a simplified temporal model designed to support a historic database and primarily point queries. We treat time as integer-like with two distinguished time points, `start` and `now`, where `start` comes before all other (named) time points and `now` comes after all other (named) time points. Time points are represented by integers, counting the number of milliseconds around the epoch January 1, 1970[6]. A negative number is the number of milliseconds before the epoch. To simplify the algorithms supporting our temporal model, we represent temporal intervals as closed-open intervals over the integers, so that if we say that a fact is true from 5 to 10, it is true over the interval `[5,10)`. This means that a temporal interval with no extent is not possible, and a time point is equivalent to the interval `[x,x+1)`.

4.4 Temporal Translation Of Assertions

To capture the assertion `holdsIn(E,p(a))`,we add two arguments to the `p(a)` literal representing the start and end times that it holds: `p(a,s,e)`, where `s`/`e` are the start/end times for `E`. This breaks the connection with the temporal object `E`, but simplifies the syntax and makes it more appropriate for the indexing schemes of XSB.

The start and end arguments are handled specially with respect to unification in order to model time. Bound arguments unify if they are within the range given by `[s,e)`. Unbound arguments are bound to the maximum possible range. Thus, the fact `p(a,5,10)` satisfies the query `p(X,6,9)`. It also satisfies `p(X,6,E)`, binding `E` to 10. It does not satisfy `p(X, 11, E)` or `p(X,6,12)`.

To implement this, rules are generated which perform a binding pattern analysis on the temporal arguments of a call and uses a rule which handles the call appropriately. Assertions are made to extensional versions of the predicate. For example, if `p/1` is in the language, then the following rules are generated:

[6] This epoch is an artifact of using Java for the generated APIs.

```
p(X,S,E) :-
     nonvar(S),nonvar(E),p_ff(S1,E1),
     atOrBefore(S1,S),atOrBefore(E,E1).
p(X,S,E) :-
     nonvar(S),var(E),p_ff(S1,E),
     atOrBefore(S1,S).
p(X,S,E) :-
     var(S),nonvar(E),p_ff(S,E1),
     atOrBefore(E,E1).
p(X,S,E) :-
     var(S),var(E),p_ff(S,E).
p_ff(X,S,E) :- p_EDB(X,S,E).
```

where `atOrBefore` does a temporal comparison between its arguments, and `p_EDB` is the extensional version of p. The extracted rules with p as a head are translated with a `p_ff` head. The `_ff` suffix indicates that the temporal arguments are unbound. These rules are generated so that unbound temporal intervals of a call are bound to the full interval inferred – backtracking over subintervals is not supported. They also ensure that temporally bound calls are satisfied if an interval that contains the given one can be inferred.

Casting the query to one with unbound temporal arguments is required since there are not any special indexing structures (in XSB) to determine "subinterval unification".

4.5 Temporal Translation Of Horn Rules

Most of the rules in Cyc have no `holdsIn` component, instead relying on sub-abstraction for a temporal interpretation. We decided not to try to translate Cyc's sub-abstraction mechanism. We treat a rule like

$$p(X,Y) \wedge q(Y,Z) \rightarrow p(X,Z)$$

as:

```
p(X,Z,S,E) :-
    p(X,Y,S1,E1),
    q(Y,Z,S2,E2),
    tInter([(S1,E1),(S2,E2)],(S,E)).
```

where `tInter` binds `(S,E)` to the temporal intersection of the intervals in the first argument. This makes `p(X,Z)` true during the times that both `p(X,Y)` and `q(Y,Z)` are true. The rules are designed so that temporal arguments get bound to the largest range that can be established with the current bindings to the sub-goal temporal arguments.

4.6 Translation Of Rules With Negated Subgoals

We read the rule

$$p(X,Y) \wedge q(Y,Z) \wedge \neg r(Z) \rightarrow p(X,Z)$$

as "`p(X,Z)` is true during the times that `p(X,Y)` and `q(Y,Z)` are both true, but that `r(Z)` is not". This is realized with the translation into

```
:- tabled(r/3).

p(X,Z,S,E) :-
   p(X,Y,S1,E1),
   q(Y,Z,S2,E2),
   tInter([(S1,E1),(S2,E2)], (S3,E3)),
   tsetof((S4,E4), r(Z,S4,E4), NEG),
   tDiff([(S3,E3)],NEG,DIFF),
   member((S,E), DIFF).
```

`tDiff` binds `DIFF` to an array containing the portions of the intervals in the first argument which are not covered by intervals in the second. The `tsetof` built-in is an XSB construct, which behaves like Prolog's `setof`, but is used for tabled predicates[7]. The rule has to identify all of the intervals for which `r(Z)` is true in order to ensure that `(S,E)` does not overlap with some interval for `r(Z)`. The results of the `r(Z)` call are tabled so that they are not repeatedly computed (for each instantiation of `(S3,E3)`).

4.7 Problems With The Temporal Translation

We are currently restricted to stratified programs because of how we handle rules with negated subgoals. We have not been able to design an efficient temporal model for the WFS in general.

Binding the temporal arguments of a call to the maximal interval makes the program sensitive to goal orderings. For example, with the assertions `p(a,5,10)` and `q(a,0,20)`, the query

```
p(X,S,E), q(X,S,E)
```

succeeds, but

```
q(X,S,E), p(X,S,E)
```

fails (since `p` is not true during the entire interval of `[0,20]`). This requires a standard query format for handling compound queries with unbound temporal arguments:

```
q(X,S1,E1), p(X,S2,E2),
tInter([(S1,E1),(S2,E2)], (S,E)).
```

so that `(S,E)` provides the temporal context for the answer `X=a`.

A further limitation of our current implementation is that no effort is made to merge conclusions with overlapping temporal intervals which are otherwise unifiable. So if the program can conclude `p(a,0,6)` and `p(a,5,10)`, it will not necessarily conclude `p(a,0,10)`. We did not feel justified in adding this (and significantly decreasing performance) since the queries made to the system are either point queries or the temporal arguments are unbounded (and all answers are returned). Given the type of temporal queries that are asked, the standard format for asking compound queries, and the way that the rules are translated (explained above), a call with a bound non-

[7] The `tabled` directive tells the XSB compiler that the given predicate should be tabled. A tabled predicate has the answers to its calls cached in a table. Reference [Swi94] for more details on XSB's tabling.

point temporal interval is not made. This means that the only kind of rule that is dependent on temporal merging is one that tests the interval returned from a call (like testing for a length or for specific boundary points), and these kinds of rules are (so far) infrequent for the domains we are interested in. So far this has not proven to be a problem.

4.8 Adding Assertion Identifiers

In Cyc, literals can appear as arguments to other literals. We represent this by adding a third argument during the translation which is the identifier for the literal.

A rule which uses a predicate which takes a literal as an argument (which can be determined by the type signature of the predicate) is transformed so that it takes an assertion identifier which is de-referenced to the appropriate assertion. For example, the Cyc rule

```
if p(X, q(Y)) then r(X,Y)
```

is translated as

```
r(X,Y,S,E,A) :-
        p(X,AID,S,E,A1),
        q(Y,_,_, AID).
```

4.9 Translating Cyc's Default Logic

No attempt has been made to duplicate Cyc's argumentation-based non-monotonic formalism; instead, the `ExceptWhen` and `ExceptFor` assertions are translated by adding negated subgoals to the rules that they act on. For example,

`ExceptWhen(p(Y),r(X,Y)→q(X,Y))`

modifies the rule so that it becomes

`r(X,Y)∧¬ p(Y)→q(X,Y)`

We then rely on the WFS to achieve a behavior consistent with major non-monotonic formalisms ([Pry91]).

4.10 The Resulting Dependency Graph

Our first application of the extraction and translation resulted in 1546 rules and 2547 ground facts[8]. The dependency graph of the rules has 162 strongly connected components: 133 of these were of length one (an immediately recursive rule), 22 of size 2, 1 each of size 3, 4, 5, and 6, and 1 of size 32.

4.11 Subgoal Reordering

In our initial application of this process, the resulting rules did not lend themselves to efficient evaluation with a left-to-right subgoal selection rule. Even with the tabling of XSB, the search space was just too large even for simple queries. This problem was effectively dealt with by profiling a set of example queries and reordering the subgoals of the rules involved. While we have not designed an algorithm to effectively reorder the subgoals, our experiences with manual reordering leads us to believe that we can. Most of the reordering involved preventing unbound subgoal calls and taking advantage of XSB's tabling mechanism, each of which can be accomplished by analyzing binding patterns.

4.12 Information loss

The purpose (as discussed earlier) of translating the extracted sub-ontology into a logic program interpreted under the WFS (with a finite domain) is to ensure a polynomial data complexity and a more appropriate semantics for negation. It is difficult to characterize what has been lost or gained in the translation because, which CycL is based on first-order logic, it has many extensions for which Cycorp has not provided a clear semantics. For the purpose of this discussion, assume a classical semantics for CycL.

Some conclusions could be made from the original CycL, but others can be made with the resulting XSB. CycL rules like

$$p \rightarrow q \vee r$$

could not directly translate into the target lp; however, since one of the (classically equivalent) forms such as

$$p \wedge \neg q \rightarrow r$$

would be added to the lp, the lp (using a WFS negation) could allow for concluding `r`. The original CycL rule is still used to influence the database since it is used as an integrity constraint.

Recall that steps were taken to ensure a finite domain for the resulting lp. The initial version of the logic program generator took drastic steps to ensure this, such as removing any rule which uses a functional term. This is certainly a source of a loss of information. Later versions will attempt to reduce this loss by translating the function symbols (of arity `n`) into predicates (of arity `n+1`) which are functional (in the last argument); however, this will still entail a finite domain.

Once the restriction to a finite domain has been made, the use of the WFS (vs classical semantics) actually increases the expressiveness. First-order logic on finite domains has a logarithmic time data complexity, while the WFS (first-order plus a least-fixed point operator) is polynomial ([Abi95] and [Imm89]). This allows the lp to define relations that the original CycL could not. An example is defining the transitive closure (TC) of a relation. The original CycL rules saying that one relation is the TC of another can be used under the WFS to say that the one actually defines the TC of the other. This capability allows for more (non-monotonic) conclusions.

So the only clear source of loss is the restriction to a finite domain. The other sources possible sources of loss do prevent some kinds of conclusions, but allow for others.

5 The API generator

The Knowledge Bus runtime system exposes the underlying logic program as an object-oriented API in a com-

[8] 1758 sentences were removed for various reasons dependent on our initial implementation. One such reason is that we are not currently doing any translation of function terms from Cyc.

monly-used language. This allows programmers without training in logic programming to use the database. The goal is to make the deductive layer invisible, appearing to be an ordinary object-oriented API.

We selected Java as the OO language of the runtime system. This allows Knowledge Bus-derived APIs to be used by anybody with skills in this popular language. This also allows development with existing development environments and to integrate with existing applications.

By using a strongly-typed language, some kinds of database constraints can be checked by the compiler. For example, the database may have a requirement for the predicate `drives`: `Person`×`Vehicle`. By defining the method `Vehicle drives()` in the class `Person`, then any attempt to use the `drives` method incorrectly will cause the program to be rejected by the compiler.

Other constraints cannot be represented directly using the class notation. For example, the constraint `parent(X, Y) :- olderThan(X, Y)` cannot be easily expressed using Java classes. When these constraints are violated, the user is notified by an exception at run time.

5.1 Knowledge Bus Object Model

In order to make the structure of the Java class definitions match the structure of the database, we define a set of rules for creating classes. Intuitively, we create a Java class corresponding to each collection such that the class hierarchy corresponds to the `genls` taxonomy in the database. Constants in the database are modeled as Java objects, using the constant itself as the object identifier. Predicates are modeled as methods on the classes; the methods are implemented using the predicates.

These rules define the object model:

1. For all collections `X`, there is a class C_X corresponding to `X`.
2. For all classes C_X, if `genls(X, Y)` then C_Y is a superclass of C_X.
3. For all predicates `p`: $T_1 \times \ldots \times T_n$ and all i from 1 to n, there are n-1 methods in class C_{T_i} named m_j for all j from 1 to n, not including i. The signatures of these methods are
$$\left(C_{T_1},\ldots,C_{T_{j-1}},C_{T_{j+1}},\ldots,C_{T_{i-1}},C_{T_{i+1}},\ldots,C_{T_n}\right)\rightarrow\left[C_{T_j}\right]$$
where `[`τ`]` represents an array of type τ.
In the most common case, where n=2, for the predicate `p:X`×`Y`, this corresponds to the class definitions:

```
class C_X { C_Y[] p(); }
class C_Y { C_X[] p(); }
```

4. For all constants `c` in the database such that `isa(c, X)` holds, then there is a corresponding Java object o_c such that o_c `instanceof` C_X. We define `c` to be the *object identifier* of the object o_c.
5. Define the function ϕ: object→constant such that $\phi(o_c) \rightarrow c$.
6. For each method m_j in class C_X, objects o_m `instanceof` C_{T_m} and t `instanceof` C_X, the semantics of the m_j are defined such that the result of invoking
$t.m_j(o_1, \ldots, o_{j-1}, o_{j+1}, \ldots, o_{i-1}, o_{i+1}, \ldots, o_n)$
is the set of objects Z such that
$$p(\phi(o_1), \phi(o_2), \ldots, \phi(o_{j-1}), \phi(t), \phi(o_{j+1}), \ldots, \phi(o_{i-1}), \phi(Z), \phi(o_{i+1}), \ldots, \phi(o_n))$$
holds, where p is the predicate corresponding to m_j. This can be easily evaluated in the logic program by substituting a variable for $\phi(Z)$, and using all bindings of Z as the object identifiers of the resulting objects.

Rule 6 provides method definitions for the most common case, where exactly one argument is unbound and all others are bound. Section 6.3 provides a more general solution for other binding patterns.

The intuition behind rule 6 is that result of executing a method is the set of objects corresponding to the bindings of variable `Z`. All other arguments to the predicate are bound to the object identifiers of the arguments of the method. Thus, for an object with identifier `joshua` of class `Person`, executing the method `hobbies:()→Activities` yields the array of objects with object identifiers [`marathon_running`, `computer_programming`].

In practice, the class name C_X is usually identical to the constant name `X`. Some mangling of names must be done to support Java's naming conventions (replace hyphens by underscores, etc.). This paper will use the name `X` to refer to C_X when this will not result in confusion.

Rule 6 does not include the metadata parameters to the underlying database relations. These are taken from a single `Context` object which provides limits for these parameters. This works well in the most common case, where all queries are asked at the same time point. The construction of these are straightforward, following the example of section 4.3.

It must be proven that the results of calling a method meet the Java requirement for type safety. It is straightforward to prove given the database constraints that all bindings of `Z` are constants such that `isa(Z,` T_j`)`. This can be used with the definition of the method to prove that for all `Z`, ϕ(`Z`) `instanceof` C_X.

6 The runtime system

The runtime system supports access to the generated databases systems by providing a dynamic class creation capability, and a relational interface which allows for other binding patterns in queries (other than the bound-free pattern provided for by an object-oriented interface).

6.1 Meta-object protocol

The ordinary Java notion of a class is insufficient to fulfill rules 2 and 3 because Java lacks multiple inheritance. In

addition, even if Java had multiple inheritance, it would be impractical to create classes for all possible subsets of the set of collections.

In order to meet these goals, Java is augmented with a Metaobject Protocol for Java ([Eng97]). This is a system for defining classes based on the Metaobject Protocol for CLOS ([Kic91]).

The Metaobject Protocol for Java (hereafter MOP) requires no changes to the Java language, and uses existing Java compilers. It is based on Java's ability to load new classes into a running system. Under MOP, the classes described above are actually Java interfaces. The MOP has the ability to dynamically create a class which implements any arbitrary collection of interfaces. The methods of these new classes are implemented according to rule 6. The MOP uses a protocol similar to CLOS's MOP to resolve conflicts arising from multiple inheritance.

In the Knowledge Bus, there are no conflicts because the underlying logic program enforces the restriction that each predicate has exactly one signature. Problems of multiple inheritance (and overriding in general) are avoided.

Whenever the constant *c* is present in the result of a query, the constant is *swizzled* into an object. This involves creating an object whose class implements the interfaces corresponding to the types of the constant (that is, all `X` such that `isa(`*c*`, X)`). If this is a novel collection of interfaces, a new class is created. The methods of this class are implemented as described in rule 6.

6.2 Naming conflicts

The rules defined in section 5.1 have a serious flaw with respect to the Java language. Consider two collections `X` and `Y` such that `genls(X,Y)`, and a predicate `p: X×Y`. This gives rise to these interface definitions:

```
interface X { Y[] p(); }
interface Y implements X { X[] p(); }
```

`Y` inherits the definition of `p` from `X`. This means that `Y` has two method definitions named `p` with identical arguments but different return types. This is an error.

To resolve this error, we have chosen the naming convention which adds `Of` to any method whose implementation involves leaving the first argument unbound. Thus, we define `Y` as:

```
interface Y implements X { X[] pOf(); }
```

This naming convention feels natural to the programmers. For example, for `owner: Agent×Thing`, this definition matches the programmer's intuition:

```
interface Thing { Agent[] ownerOf(); }
interface Agent { Thing[] owner(); }
```

A different, less natural naming convention exists for higher-arity predicates, in which conflicts are resolved by affixing the binding pattern (a string of `B`s and `F`s) to the name of the predicate.

6.3 Other binding patterns

Since most predicates are binary, the most common query involves one bound argument and one unbound argument. Even where there are more arguments, there is generally exactly one unbound argument. This situation is well modeled by the methods described here; invoking a method is equivalent to finding all answers to a query with one unbound argument. Java programmers usually find this paradigm very natural.

Sometimes, a programmer may require alternative binding patterns, especially with higher-arity predicates. For this reason, an additional view of the database called the *relational interface* is provided. The relational interface consists of a collection of static methods defined thus:

For all predicates $p: T_1 \times \ldots \times T_n$, there is a static method `m:` $(C_{T_1}, \ldots, C_{T_n}, S, E, A) \rightarrow$ `Query`. When invoked on a set of objects $o_1 \ldots o_n$, this sets up an object of class `Query` which models a general query into the logic program[9].

Any subset of the arguments o_i may be *unbound objects*. An unbound object is an object without a fixed object identifier. Unbound objects are created by a specially provided method, `makeVar(Class[])`, which returns a new object with the appropriate types but no object identifier.

The `Query` object is used by invoking the method `nextBinding`. The logic program is asked the query `p(`x_1, x_2, x_3, …, x_n, S, E, A`)`, where x_i corresponds to the object identifier of the object (if it is a bound object) or a variable (if it is an unbound object). If the same unbound object appears in more than one position, the same variable is used.

For each set of bindings of the variables, the object identifiers of the unbound objects are altered to match the bindings.

`Query` objects may be combined with conjunctions like `and`, `or`, `not`, and others. For example, to find all of `joshua`'s aunts, (where `joshua` is an object of class `Person`) use this Java code:

```
Person X =
    (Person) makeVar(Person.class);
Person Y =
    (Person) makeVar(Person.class);
Query q =
    parent(joshua, X), sister(X, Y);
while(q.nextBinding()) {
   // Y is now bound to one of
   // Joshua's aunts
}
```

[9] The extra arguments (S, E, A) are added during the translation process. See sections 4.3 and 4.8.

7 Future Work

7.1 Introspection, mobile agents, and discovery

An indirect but extremely important benefit of using Java for the generated programming interfaces is that dynamic linking of client systems is greatly simplified. Using the reflection capabilities of Java 1.1, a client which subscribes to the common ontology could interrogate a generated database to discover if it manifests a desired interface. If it does, the client can dynamically link to the database to operate on it. This is an important enabling feature for the widespread application of mobile agents. We already use a rudimentary form of this capability in building client applications - methods which require heavy database access can be shipped to a running database and evaluated, thus reducing network overhead.

7.2 Improvements to the temporal model

The two limitations of the temporal model that we are most concerned with is the restriction to stratified programs and the inability to merge overlapping temporal intervals (for otherwise unifiable literals). We would like to develop efficient solutions to both of these problems.

7.3 Improvements to XSB

Unfortunately, there are no commercial deductive DBMS currently available. To duplicate the same functionality using XSB, we need to implement two important capabilities - concurrent query evaluation and transactions. Work is already underway at SUNY Stony Brook to support these features. In fact, we plan to use the proposed concurrency extensions to implement distributed query evaluation.

7.4 Improvements to the generation process

The current generation process isn't quite so automatic as we would like. For example, due to the limitations of the current temporal model, we are forced to eliminate some axioms to ensure stratification, which requires some interaction.

We are trying to improve the specificity of the sub-ontology extraction process. While the current system has yielded promising results, many superfluous collections and axioms are extracted from Cyc. Part of this is due to the structure of the Cyc KB. Cycorp is working on improvements to Cyc's context mechanisms to allow more fine-grained specification of an application context.

8 Related Work

[Swa96] describes an algorithm which uses heuristic techniques to extract application-focused subsets from the large (70K concept) SENSUS ontology. SENSUS is a linguistically-based ontology with subsumption but no relations or axioms.

9 Acknowledgments

We would like to thank the other members of the Knowledge Bus project - Jon Shoemaker and Paul Brinkley for their excellent Java programming work. We would also like to thank Professors David S. Warren of SUNY, Stony Brook and V.S. Subrahmanian of the University of Maryland, College Park for their invaluable advice and assistance. Doug Lenat, Nick Siegel, Keith Goolsbey, Fritz Lehmann and David Gadbois of Cycorp helped us understand the intricacies of Cyc's KB and inference engine. Special thanks to Joe O'Kane of DoD for having enough vision to support our work.

This work was supported by internal R&D funding from the U.S. Government, Department of Defense.

10 References

[Abi95] S. Abiteboul, R. Hull, V. Vianu, *Foundations of Databases*, Addison-Wesley Pub. Co. Inc., 1995.

[Aho79] A.V. Aho, J.D. Ullman: The Universality of Data Retrieval Language. *POPL* 1979: 110-120.

[Imm89] N. Immerman, Descriptive and Computational Complexity. *FCT* 1989: 244-245.

[Eng97] J. Engel, *A Meta-Object Protocol for Java*, unpublished MS, 1997.

[Gel91] A. Van Gelder, K.A. Ross, J.S. Schlipf, The Well-Founded Semantics for General Logic Programs. *JACM* 38(3), 1991: 620-650.

[Guh94] R.V. Guha, D.B. Lenat: Enabling Agents to Work Together. CACM 37(7), 1994: 126-142.

[Kic91] G. Kiczales, J. des Rivières, and D. G. Bobrow, *The Art of the Metaobject Protocol*, MIT Press., 1991.

[Pry88] T.C. Przymusinski, On the Declarative Semantics of Deductive Databases and Logic Programs. *Foundations of DD and LP*, ed. J. Minker, Morgan Kaufmann Pub. Inc., 1988: 193-216.

[Pry91] T.C. Przymusinski, Three-Valued Non-Monotonic Formalisms and Semantics of Logic Programs, *Artificial Intelligence Journal* 13, 1991: 445-464.

[Sag96] K.F. Sagonas, T. Swift, D.S. Warren, An Abstract Machine for Computing the Well-Founded Semantics. *Proceedings of the Joint Conference and Symposium on Logic Programming*, Sep. 1996, MIT Press: 274—288.

[She88] J.C. Shepherdson, *Foundations of DD and LP*, ed. J. Minker, Morgan Kaufmann Pub. Inc., 1988: 19-88.

[Swa96] B. Swartout, R. Patil, K. Knight, T. Russ, Toward Distributed Use of Large-Scale Ontologies, *Proceedings, Tenth Knowledge Acquisition for Knowledge-Based Systems Workshop*, Banff, Canada, 1996.

[Swi94] T. Swift, D.S. Warren, Analysis of Sequential SLG Evaluation, *ILPS* 1994: 219--238.