OKBC: A Programmatic Foundation for Knowledge Base Interoperability [1] [2]

Vinay K. Chaudhri
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025
vinay@ai.sri.com

Adam Farquhar
Knowledge Systems Laboratory
Stanford University
Stanford, CA, 94309
axf@ksl.stanford.edu

Richard Fikes
Knowledge Systems Laboratory
Stanford University
Stanford, CA 94309
fikes@ksl.stanford.edu

Peter D. Karp[3]
Pangea Systems
4040 Campbell Ave
Menlo Park CA 94025
pkarp@PangeaSystems.com

James P. Rice
Knowledge Systems Laboratory
Stanford University
Stanford, CA 94309
rice@ksl.stanford.edu

Abstract

The technology for building large knowledge bases (KBs) is yet to witness a breakthrough so that a KB can be constructed by the assembly of prefabricated knowledge components. Knowledge components include both pieces of domain knowledge (for example, theories of economics or fault diagnosis) and KB tools (for example, editors and theorem provers). Most of the current KB development tools can only manipulate knowledge residing in the knowledge representation system (KRS) for which the tools were originally developed. Open Knowledge Base Connectivity (OKBC) is an application programming interface for accessing KRSs, and was developed to enable the construction of reusable KB tools. OKBC improves upon its predecessor, the Generic Frame Protocol (GFP), in several significant ways. OKBC can be used with a much larger range of systems because its knowledge model supports an *assertional view* of a KRS. OKBC provides an explicit treatment of entities that are not frames, and it has a much better way of controlling inference and specifying default values. OKBC can be used on practically any platform because it supports network transparency and has implementations for multiple programming languages. In this paper, we discuss technical design issues faced in the development of OKBC, highlight how OKBC improves upon GFP, and report on practical experiences in using it.

Introduction

In the construction of a new knowledge base (KB), significant productivity gains can be obtained by reusing existing knowledge components. These components include pieces of domain knowledge (for example, theories of economics or fault diagnosis) and KB development tools (for example, editors and theorem provers). To support reuse of domain knowledge, the knowledge sharing community has undertaken various efforts, including the development of shared portable ontologies (Farquhar, Fikes, & Rice 1997) and the development of well-defined languages for knowledge interchange (Genesereth & Fikes 1992). There has been, however, less emphasis on the reuse of KB development tools. A significant amount of effort is invested in building customized tools for specific knowledge representation systems (KRSs). These tools work only with a single KRS, and the development effort is wasted if the KRS is no longer used. A KRS developer usually does not have the choice of using off-the-shelf tools and is forced to develop tools on her own.

Open Knowledge Base Connectivity (OKBC) is an application programming interface (API) for KRSs that has been developed to address the problem of KB tools reusability. The name OKBC was chosen to be analogous to ODBC (Open Database Connectivity), as used in the database community (Geiger 1995).

An API specifies the *operations* that can be used to access a system by an application program. When specifying an API for a KRS, some assumptions must be made about the representation used by that KRS. Such assumptions are made explicit in the *OKBC knowledge model*. As it can be too restrictive to enforce the same semantics for all operations in an API across all KRSs, OKBC supports *behaviors* to allow for differences among KRSs. Behaviors are a tool to achieve flexibility in specifying OKBC operations. Thus, the OKBC specification consists of three components: a knowledge model, a collection of operations to access a KRS, and a collection of behaviors.

A KRS can be *bound* to OKBC by defining a mapping from OKBC to the native API of that KRS. To achieve interoperability, a KB tool accesses a KRS using only OKBC operations. Such a tool is isolated from the peculiarities of the KRS and can be used with any KRS that has been bound to OKBC. The interoperability achieved by using OKBC is at the level of the OKBC knowledge model. For example, the OKBC knowledge model defines the concept of a *class* that has the same interpretation across all OKBC bindings. OKBC does

[1]

[2] OKBC implementations in Lisp, C and Java may be obtained from `http://ontolingua.stanford.edu/okbc/`.

[3] The work presented in this paper was done while the author was at SRI International.

not guarantee, however, that a particular class (e.g., `Person`) defined in KBs residing in two different KRSs represents identical concepts.

OKBC is a successor to the Generic Frame Protocol (GFP) (Karp, Myers, & Gruber 1995) and improves upon GFP in two significant ways. First, OKBC supports a larger class of KRSs because its knowledge model includes an *assertional view* of a KRS, provides an explicit treatment of entities that are not frames, and has a much better way of controlling inference and specifying default values. Second, OKBC can be used on practically any platform and with a substantially larger range of applications because it supports network transparency, multiple programming languages, and a remote procedure language.

Two conclusions can be drawn from the design experience of OKBC. First, an expressiveness vs. generality tradeoff emerged. The expressiveness of the knowledge model must be controlled so that it can work with a range of KRSs. If the knowledge model is too expressive, it becomes difficult to define OKBC bindings for systems that have limited representational power. If the knowledge model is insufficiently expressive, the OKBC bindings for systems with more representational power will not expose their capabilities. Second, the protocol was augmented by two features to support variability among KRSs: additional returned values and behaviors. Wherever it is not feasible to legislate certain requirements, KRSs can expose the difference either globally by setting the value of a *behavior* or locally by returning an additional value from an operation.

This paper is devoted to the discussion of the technical issues faced in the design of OKBC. It is not intended to be a comprehensive description of OKBC, which may be found elsewhere (Chaudhri *et al.* 1997; Rice & Farquhar 1998). The paper is organized along the two major classes of enhancement in OKBC: expanding the range of supported KRSs, and expanding the range of supported applications. We also discuss practical experiences in using OKBC.

The OKBC Knowledge Model

The OKBC knowledge model is designed to include representational features supported by several KRSs (Karp 1992). It includes constants, frames, slots, facets, classes, individuals, and knowledge bases. Classes and individuals form two disjoint partitions of a KB (see Figure 1). A *class* is defined as a set of entities. Each of the entities in a class is said to be an *instance* of that class. An *individual* is an entity that is not a set.

Any entity has associated with it a collection of *own* slots. Own slots describe the direct properties of an entity. For example, if the age of Fred is 42, then `age` is an own slot of `Fred`. Own slots and their values are not inherited. A class has associated with it a collection of *template* slots. Template slots describe properties of the instances of a class (own slots of a class describe the properties of the class itself). Template slots are inherited by subclasses of a class; a template slot on a class becomes an own slot on each instance of the class.

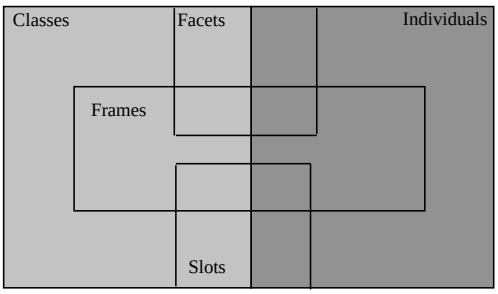

Figure 1: The OKBC knowledge model defines that classes and individuals form disjoint partitions of a KB. It does not commit to whether classes, individuals, slots, and facets are represented as frames. It also does not commit to whether slots and facets should be represented as classes or individuals.

Own facets describe the properties of slots associated with an entity, for example, cardinality or range. A template slot of a class has associated with it a collection of *template* facets that describe own facets for the corresponding own slot of each instance of the class.

Orthogonal to the knowledge-level distinction between classes and individuals is the notion of a *frame*. A frame is a data structure that is typically used to represent a single entity and the slots and facets associated with it. The decision as to which entities are represented as frames is driven primarily by implementation considerations; historically, KRSs have made different decisions about it. The OKBC knowledge model does not legislate which entities are frames (see Figure 1). For example, in a given KRS, classes may or may not be represented as frames. Even if classes are generally represented as frames, OKBC allows for a subset of classes not to be represented as frames. Indeed, it is common for KRSs to excluded unnamed sets, such as {1, 2, 4}, and primitive data structures, such as numbers and strings, from the set of frames.

The OKBC knowledge model does not legislate whether slots and facets should be represented as classes or individuals. In some KRSs, a slot (or more generally, a relation) denotes a set of tuples. In such systems, a slot is therefore also a class. In other KRSs, the space of classes and slots are disjoint. In such systems, a slot is also an individual. As shown in Figure 1, the knowledge model allows for a slot or a facet to be either a class or an individual.

A KB is a collection of classes, individuals, frames, slots, slot values, facets, facet values, frame-slot associations, and frame-slot-facet associations and sentences. Multiple KBs may be represented in a KRS.

OKBC supports operations that apply to specific frames in a KB (for example, querying the values of a slot of a frame), operations that apply to a KRS but not to any specific KB (for example, getting a list of all the KBs defined using a specific KRS), and operations

that apply neither to a KRS nor to a KB (for example, establishing a connection to a knowledge server).

Expanding the range of supported KRSs

Although GFP was successfully used in several projects at Stanford University's Knowledge Systems Laboratory (KSL) (Farquhar, Fikes, & Rice 1997; Farquhar *et al.* 1996) and at SRI International (Paley, Lowrance, & Karp 1997; Karp *et al.* 1996), it lacked the power and flexibility needed in a generic API. Most of the enhancements considered here address the deficiencies encountered while GFP was used at KSL and SRI.

Support for assertions

GFP was found to be inadequate for use with KRSs that prefer to view a KB as a collection of logical sentences, as well as systems that have a knowledge model more expressive than the knowledge model of GFP. To address these problems, we introduced a *tell/ask* interface that supports an *assertional view* of a KB.

The design approach for supporting OKBC was analogous to the one adopted in KRYPTON (Brachman, Fikes, & Levesque 1983). An OKBC KB supports two alternative and isomorphic views of a KB: a frame-oriented view and an assertional view. (The frame-oriented view was called the *terminological component* in KRYPTON.) While defining the assertional view of a KB, we took a lowest common denominator approach: an assertion language with an expressive power roughly equivalent to an object-oriented frame language is defined. For other assertions, support is provided, but no portability claims are made.

Assertion Language OKBC defines an assertion language (AL) for declarative specification of knowledge. The AL is a first-order language with conjunction and predicate symbols, but without disjunction, explicit quantifiers, function symbols, negation, or equality. The predicate symbols of the OKBC AL are `class`, `individual`, `primitive`, `instance-of`, `type-of`, `subclass-of`, `slot-of`, `facet-of`, `template-slot-of`, `template-facet-of`, `own-slot-value`, `own-facet-value`, `template-slot-value`, and `template-facet-value`. For example, (`instance-of John Person`) means that `John` is an instance of the class `Person`. For convenience, (`instance-of John Person`) may be written as (`Person John`).

A well-formed formula (WFF) of the AL is an atomic formula constructed by enclosing one of the predicate symbols followed by a number of terms in parentheses. The terms of the AL are constants and variables. The conjunction of two WFFs of the AL is a WFF.

OKBC provides the **tell**, **ask** and **untell** operations to query and update a KB using the AL. OKBC guarantees that only ground WFFs can be **tell**ed or **untell**ed. Any WFF may be **ask**ed.

OKBC specifies the effect of **tell**ing any WFF of the AL to a KB by identifying an equivalent set of OKBC operations that does not include **tell**. For example, the operation (**tell** (`instance-of` frame class)), which asserts *frame* to be an instance of *class*, is equivalent to the operation (**add-instance-type** frame class). **Ask**ing any WFF of the AL is similarly equivalent to a set of OKBC operations not including `ask`. For example, the operation (**ask** (`instance-of` ?x class)) is equivalent to the operation (**get-class-instances** class).

Assertions not guaranteed to be supported by OKBC To handle assertions outside of the AL, OKBC defines the operations **tellable** and **askable**. The OKBC operation **tellable** determines which sentences may be acceptable to **tell** for a specific KB. Before using **tell** with an arbitrary formula, an application can check whether a formula is **tellable**. If the formula is not **tellable**, the application cannot safely assert that formula using **tell**.

For example, consider the WFFs (`age John` 30) and (`friend John Sally`). It is straightforward to assert them using either **tell** or **add-slot-value**. An application may, however, wish to assert the disjunction, (`or` (`age` John 30) (`friend` John Sally)), which is not a WFF of the AL. An OKBC binding for a KRS is free to accept this formula, and an application can check for this by using the **tellable** operation. If a formula is **tellable** for a KRS, the **tell** operation can be used to communicate that formula to the KRS. Using this mechanism, a KB may accept formulae that contain quantifiers, functions, or higher arity predicates.

There is no equivalence between using **tell** with arbitrary formulae and a set of OKBC operations that do no use **tell**. Use of such formulae may, therefore, not be portable across different OKBC bindings.

Handling entities that are not frames

As discussed above, KRSs make different assumptions about which entities are represented as frames. These differences influence the semantics of operations that systematically process frames in a KB (for example, **get-kb-frames**, **get-kb-individuals**, **get-kb-classes** that respectively return all the frames, classes, and individuals in a KB). These operations could be specified by saying that they respectively return all the frames, classes, and individuals in a KB. Because of differences in which entities are represented as frames, this simplicity can be deceptive.

Not all entities are frames As shown in Figure 1, not all classes in a KB are necessarily represented as frames. Given such differences, it is not obvious how to define the operation **get-kb-classes**. Should it return only those classes that are frames? Should it return all sets in a KB?

Returning only those classes that are frames is a problem for KRSs that do not represent all classes as frames. Some of the non-frame classes can be important to a client application. Defining **get-kb-classes** to return all the sets is also problematic because the results

of one OKBC operation cannot necessarily be passed to another operation, making an application program more complex. The complexity occurs because it is generally not possible to perform operations such as creating slots and adding slot values to entities that are not frames. Thus, if **get-kb-classes** were to return classes that are not frames, an application program would need to identify those classes that are not frames and treat them differently. A possible solution to this problem is to require a KRS to appear as if it represents every class as a frame. This is not reasonable, however, because it is unnatural and can make the implementation extremely inefficient.

To address this problem, we introduced an extra argument, **selector**, to **get-kb-classes** and similar operations. When the value of **selector** is `:frames`, only those classes that are frames are returned, and when its value is `:all`, all classes are returned. For a system in which all classes are represented by frames, **get-kb-classes** returns identical results for these two values of **selector**. We expect many applications to use `:frames` as a value for the **selector** argument, because it has the desirable property that the union of **get-kb-classes** and **get-kb-individuals** equals the result of **get-kb-frames**. A third legal value for this argument is `:system-default`, which gives a KRS the freedom to use the most efficient or natural method of computing **get-kb-classes**.

Not all entity categories are frames As shown in Figure 1, not all KRSs represent all categories of entities as frames. Consider two KRSs: KRS1, which represents slots as frames, and KRS2, which does not. Furthermore, consider KB1, stored in KRS1, and KB2, stored in KRS2, both of which were created using an identical set of OKBC creation operations. Calling an operation such as **get-kb-frames** will return different results on KB1 and KB2. This may make it more difficult for an application to work portably with both KBs, but it is acceptable if OKBC provides a mechanism to detect the difference. The `:are-frames` behavior allows a KRS to indicate which categories of entities are represented as frames.

The values of the `:are-frames` behavior constitute a set of the following keywords: `:class`, `:slot`, `:facet`, and `:individual`. If the values of `:are-frames` contain an entity category, it implies that frames may be used to represent them. In most KRSs, classes and instances of those classes are represented as frames, and therefore we expect the most common set of values for `:are-frames` to be at least {`:class`, `:individual`}. If two KRSs have different values for the behavior `:are-frames`, we can expect to get a different list of frames by executing **get-kb-frames** on these KBs.

Controlling KRS inference

One area in which KRSs differ widely is in the inference mechanisms that they support and in the methods available to control the inference mechanisms. It is critical for applications to have some means of controlling the type and cost of inferences that a KRS performs in response to a retrieval operation. Unfortunately, there is not yet widespread agreement on either the inference mechanisms or the parameters used to control them. This makes it impossible for OKBC to provide a rich KRS-independent method for controlling inference. Instead, OKBC provides a restricted method for specifying which inferences should be performed in retrieval operations, as well as methods for a KRS to indicate the degree to which the specifications have been satisfied. OKBC does not provide means to specify limits on computing time in performing those inferences.

OKBC retrieval operations support an **inference-level** argument that takes one of the following three values: `:direct`, `:taxonomic`, or `:all-inferable`. When **inference-level** is `:direct`, *at least* the directly asserted non redundant values are returned. When **inference-level** is `:taxonomic`, *at least* the directly asserted and inherited values are returned. The inherited values are computed using at least the taxonomic inheritance axioms defined by the knowledge model. For example, a taxonomic inheritance axiom for slot values states that if a template slot S of a class C has value V, then for all instances of C, the own slot S has value V, and for all subclasses of C, the template slot S has value V. Similar inheritance axioms are defined for facet values, and for the class/subclass and class/instance relationships. When **inference-level** is `:all-inferable`, values inferable by any means supported by the KRS are returned, including any values inferable at the `:taxonomic` inference level.

With an **inference-level** value of `:direct`, returning exactly the directly asserted values may impose a high burden on some systems such as forward chaining systems that do not maintain a distinction between directly asserted and inferred values. To permit flexibility in such cases, we use the following two techniques.

First, the **inference-level** argument defines the lower bound on the values that may be returned. For example, when the **inference-level** is `:direct`, *at least* the directly asserted values are returned, but a KRS is not prevented from returning additional values. Second, any OKBC operation accepting the **inference-level** argument returns two additional values, called **exact-p** and **more-status**. The value of **exact-p** is *true* if it is known that exactly the `:direct` (or `:taxonomic`) values are returned. An OKBC implementation that always returns *false* as the value of **exact-p** is compliant. The value of **more-status** is either *false*, which indicates that there are known to be no more results, or `:more`, which indicates that there may still be more results but the KRS was unable to find out how many more, or an integer, which indicates how many more values exist.

By specifying the inference level in terms of the lower bound on the result and returning two additional values, **exact-p** and **more-status**, we were able to permit flexibility in the specification and also be accurate.

Handling defaults

In the absence of any widely accepted model of defaults (Brewka, Dix, & Konolige 1997), OKBC incorporates only simple provisions for default values of slots and facets. Template slots and template facets have a set of *default values* associated with them. Intuitively, these default values inherit to instances unless the inherited values are logically inconsistent with other assertions in the KB, the values have been removed, for example, at the instance, or the default values have been explicitly overridden by other default values. OKBC does not require a KRS to determine the logical consistency of a KB, nor does it guarantee a means of explicitly overriding default values. Instead, OKBC leaves the inheritance of default values unspecified. That is, no requirements are imposed on the relationship between default values of template slots and facets and the values of the corresponding own slots and facets. The default values on a template slot or template facet are simply available to the KRS to use in whatever way it chooses when determining the values of own slots and facets. The slot or facet values that are not default values are referred to as "known true" values. Operations on slot and facet values take a **value-selector** argument that allows a user to choose between only default values and monotonic ("known true") values.

Expanding the range of applications

The OKBC implementation was heavily influenced by pragmatic considerations, for example, the need to support different programming languages and efficient operation over a network. Network operation is necessary because many applications are developed using a client-server model, and because knowledge sharing should not be restricted to sharing of KBs on the same machine or within the same institution.

Network transparency is achieved using an abstraction called a *connection* that encodes the actual location of an OKBC KB and mediates communication between an OKBC application and the KB. To communicate with a KB, an application program first establishes a connection to the KRS in which the KB resides and subsequently, with each OKBC operation, the program must indicate the connection that should be used in executing it. Some OKBC operations take an explicit connection argument, whereas others derive the connection from a KB argument. Thus, once a connection is established, a user need not be aware of the actual location of the KB or whether the KB is being accessed from the same address space as the application or over the network.

The network substrate in the OKBC implementation plays an important role in supporting multiple programming languages, as it allows OKBC applications to manipulate KBs through a network connection that appears and operates just like a local KB. Client-side implementations for OKBC exist for Lisp, C, and Java.

To improve efficiency in a networked environment, OKBC defines an implementation-language-independent *procedure language*. The procedure language allows an application writer to combine several OKBC operations into a single procedure or set of procedures. These procedures can be recursive, and are transmitted over a network to achieve a substantial performance boost. For example, computing the information necessary to display a complete class graph for a KB may require calling at least two OKBC operations for each class (one to get its subclasses, and the other one to get a printable representation of the class name). This could result in many thousands of invocations of OKBC operations. With the procedure language, all the invocations can be done within a single procedure, and only a single network call is needed.

OKBC operations on large KBs may return many values. If only a portion of the result is necessary, significant speedup can be obtained by retrieving only the desired part of the result. OKBC supports enumerator operations that allow an application to retrieve the result in batches. For C++ and Java, enumerators are a common programming idiom. Operations are defined on enumerators to get the **next** element, to determine if an enumerator **has-more** elements, to **fetch** a list of elements, to **prefetch** a batch of elements, and to **free** an enumerator.

Experiences in using OKBC

Defining a metric to measure the success of a generic API is difficult. We will argue that OKBC has been successful in its goal of enabling the construction of interoperable tools by presenting empirical evidence based on the definition of OKBC bindings for several KRSs. We also consider a small case study of building an interoperable tool using OKBC.

OKBC bindings

Defining OKBC bindings for a KRS means implementing a subset of OKBC operations by using calls to the native API of that KRS (Rice & Farquhar 1998). OKBC bindings for several systems have been defined by our research groups at KSL and SRI. At SRI, OKBC bindings were defined for LOOM (MacGregor & Burstein 1991), Theo (Mitchell *et al.* 1989), SIPE-2 (Wilkins 1988), and Ocelot (Paley, Lowrance, & Karp 1997). At KSL, OKBC bindings were defined for Ontolingua (Farquhar, Fikes, & Rice 1997), Abstract Theorem Prover (ATP) (a theorem prover developed at KSL), CML (Farquhar *et al.* 1996), Tuple-KB (Rice & Farquhar 1998), file system KB, and CLOS. The University of Southern California's Information Sciences Institute has now produced its own version of an OKBC binding for LOOM. An OKBC binding for Cyc (Lenat & Guha 1990) has been defined by Cycorp.

OKBC was recently licensed by Pangea Systems Inc. (see `http://www.panbio.com`) in support of its projects in the area of bioinformatics. It is used extensively in several ongoing projects at Stanford and SRI,

and has been adopted by DARPA's HPKB program (see `http://www.teknowledge.com/HPKB/`). OKBC server implementations in Lisp and Java and client implementations in Lisp, C, and Java may be obtained from `http://ontolingua.stanford.edu/okbc/`.

The KRSs for which OKBC bindings were defined fall into three categories: systems with a knowledge model that closely match the OKBC knowledge model, systems with knowledge models more expressive than the OKBC knowledge model, and systems with a knowledge model less expressive than the OKBC knowledge model. Defining OKBC bindings for systems that have a knowledge model closely matching the OKBC knowledge model is straightforward. We discuss how we handled the systems in the other two categories.

Binding a less expressive KRS A compliant OKBC binding must implement all the OKBC operations. Many OKBC users are interested in only a subset of the functionality specified by OKBC, because their KRSs have knowledge models less expressive than the OKBC knowledge model. Instead of excluding such systems, OKBC defines compliance classes that allow a KRS to specify which subset of OKBC functionality it supports. By reviewing numerous KRS bindings, we developed the following compliance classes: `:facets-supported`, `:user-defined-facets`, `:read-only`, and `:monotonic`. A KRS in the `:facets-supported` class supports facets, in the `:user-defined-facets` class it supports user-defined facets, in the `:read-only` class it supports at least all the read operations, and in the `:monotonic` class it supports at least all the operations that monotonically update a KB.

For example, consider the OKBC bindings for the Unix file system: directories are mapped to classes, subdirectory relationships are mapped to subclass relationships, and the files in a directory that themselves are not directories are mapped to individuals. For such a binding, there is no natural way to create new facets. Therefore, the OKBC bindings for a Unix directory system will not satisfy the `:user-defined-facets` compliance class. If a user does not have write permissions on a file system, it can still be compliant in the `:read-only` compliance class. An implementation of OKBC bindings for a file system is included in the OKBC source distribution (Rice & Farquhar 1998).

Binding a more expressive KRS The ATP system, developed at KSL, is a model elimination theorem prover that supports full first-order logic (FOL), provides limited support for axiom schema definitions, and is designed to handle a large number of ground facts efficiently. It provides a good example of a system with a knowledge model that is much more expressive than that of OKBC. Defining an OKBC binding for ATP presented several interesting design choices.

Because ATP is not a frame-oriented system, there is considerable freedom in deciding which objects should correspond to frames. We considered two possibilities. The first possibility is to introduce a specific class "frame", all of whose instances are frames. The second possibility is to make every object, function, and relation constant a frame (ATP allows for predications over relation and function constants). The first choice makes it harder to use the OKBC binding with an arbitrary ATP KB that does not include axioms for the class "frame". The second choice makes the ATP notion of a frame slightly more inclusive than many KRSs. For example, ATP would have a frame representing the number 42. Copying the frame representing the number 42 to a KRS that provides data structures for all frames can easily result in a frame data structure being allocated for 42, rather than using the built-in machine representation that the target KRS would prefer. Nonetheless, we chose to model all constants as frames.

ATP supports many ways of representing the basic relationships used by OKBC. Consider the subclass relationship between the class `dog` and the class `mammal`. This can be represented by the implication `(=> (dog ?x) (mammal ?x))`, or by the WFF (`subclass-of dog mammal`) of the AL. Because we expected querying and asserting subclass relationships to be a common operation, we wanted it to be efficient. ATP provides an efficient mechanism for storing ground facts. To exploit this efficient representation, all WFFs of the AL are implemented using ground facts. In addition to being efficient for OKBC's basic queries, this simplifies the deletion of frames.

For **inference-level**, OKBC defines the values: `:direct`, `:taxonomic`, and `:all-inferable`. We implemented this by using the multiple theories feature of ATP to place all of the taxonomic inference axioms in a separate theory. For inference level `:direct`, ATP looks only at the ground facts; for inference level `:taxonomic`, ATP uses the facts together with the taxonomic axioms; and for inference level `:all-inferable`, all available axioms are used.

For ATP, any FOL sentence is both **tellable** and **askable**. Some surprises may arise when a sentence is told that it is equivalent to some WFF in the AL, but has a different form. For example, the relationship between `dog` and `mammal` can be asserted in the implication, `(=> (dog ?x) (mammal ?x))`, instead of using the corresponding WFF of the AL, (`subclass-of dog mammal`). As long as the inference level is `:all-inferable`, the expected inferences are drawn (for example, a `dog` will be a `mammal`). At the `:taxonomic` inference level, however, the implication would not be used, and the inference would not be drawn.

Interoperable tools built using OKBC

At least two browsing and editing tools have been built using OKBC: the Generic Knowledge Base Editor (GKB-Editor) (Paley, Lowrance, & Karp 1997) and the Java Ontology Tool (JOT). GKB-Editor is a tool for graphically browsing and editing KBs and is written using Common Lisp. JOT is a Java-based tool for viewing and editing the contents of KBs and was written using

a commercial off-the-shelf widget package. For the purpose of the current discussion, we say that a browser has been "successfully tested" with a KRS if it is able to display the class-subclass relationships and all the contents of frames.

GKB-Editor was initially developed and tested with Ocelot. After the initial testing with Ocelot, GKB-Editor was tested with Ontolingua. One of the differences between Ontolingua and Ocelot is the value of the `:frame-names-required` behavior. Ocelot sets the value of the `:frame-names-required` behavior to *true*; Ontolingua sets it to *false*. Browsing with the GKB-Editor substantially depends on frame names. Therefore, porting the GKB-Editor to Ontolingua required us to assign fictitious names to frames in Ontolingua. Since the fictitious names have no significance to a user, they are never displayed, and instead the pretty names of the frames are shown. Except for this difficulty with frame names, we were able to test the GKB-Editor with Ontolingua successfully. In addition to Ocelot and Ontolinuga, GKB-Editor has been successfully tested with LOOM and Theo. In fact, it is being used in conjunction with LOOM in a natural language generation project at the Technical University of Berlin (Stede & Umbach 1998).

JOT was initially developed and tested with Ontolingua and Tuple-kb. It was successfully tested with Ocelot without any difficulty. JOT demonstrates the language independence of OKBC, as one can use it freely and transparently to browse and edit tightly coupled KBs implemented in Java, network-based Java KBs, and numerous different KBs written in Common Lisp. Much of this editor's operation is done by means of remote procedures. This experiment shows that tools written using OKBC, such as JOT, really do interoperate with a wide range of KRSs.

Limitations of OKBC

The goal of OKBC is to enable the construction of reusable KB tools, that is, the application programs that access a KRS to perform browsing, editing, or reasoning tasks. Empirical evidence has shown that it has been successful in meeting this goal. Potential users of OKBC are usually concerned with whether they can successfully use OKBC in their projects. Here, we identify some of the commitments and sacrifices that they may need to make to use OKBC successfully.

To construct a new OKBC binding for a KRS, it is necessary to identify the knowledge model used by the KRS and define a mapping between it and the OKBC knowledge model. By providing both frame-oriented and assertional views of a KB, OKBC is capable of supporting a wide range of systems. Some systems do not easily admit to either of these views. While an OKBC binding can be defined for such systems, some users may not find it to be an intuitive or natural mapping. OKBC bindings work best when the knowledge model of the KRS closely matches that of OKBC.

OKBC bindings isolate a KB tool from many of the peculiarities of a KRS, but certainly cannot cover all of them. Therefore, porting a KB tool to a new KRS usually requires some additional effort. For example, Ocelot supports a slot type called `:unique`. The slots of this type are inherited by subclasses and instances, but their values are not inherited. For the GKB-Editor to handle this peculiarity, a small amount of Ocelot-specific code had to be added. Similarly, ATP provides operations to return the proof that a value satisfied some query, but OKBC does not currently provide any operations for specifically extracting proofs.

OKBC is neither the lowest, nor the highest common denominator protocol. It cannot hope to expose all of the functionality of every system, but it exposes what we believe most applications want. In addition, the protocol is specifically designed to be extensible by means of the behavior mechanism so that clients and servers can negotiate the use of a more powerful functionality than is provided by the protocol.

OKBC does not solve the problem of semantic KB interoperation. For example, using the OKBC operation **get-slot-values**, an application may query the salary of a person from two different systems, but there is no guarantee that the returned values will be semantically identical — one system may return annual salary and the other system may return monthly salary. Semantic interoperation is beyond the scope of OKBC.

OKBC is a functional interface to a KB (Brachman, Fikes, & Levesque 1983) and does not specify the data structures that should be used to implement its knowledge model. Using OKBC, an application cannot manipulate internal data structures of a KRS that are used to implement frames.

Summary and Conclusions

We have shown how OKBC provides an effective interface between diverse software tools and KRSs. The OKBC knowledge model is inspired by an extensive study of existing KRSs. OKBC defines a comprehensive set of operations for accessing a KB. The semantics of those operations are precise, yet flexible enough to support both frame-oriented and assertional interaction.

For supporting variability among KRSs, behaviors and additional return values proved to be central techniques. Wherever it is not feasible to legislate certain requirements, KRSs can expose their differences from the OKBC knowledge model either globally by setting the value of a *behavior*, or locally by returning an additional value from an operation. For example, support for frame names can be advertised by using the `:frame-names-required` behavior, and the degree of conformity to the **:inference-level** argument is exposed by an extra return value.

Design experience with OKBC suggests the existence of an "expressiveness vs. generality" tradeoff that is similar to the "expressiveness vs. tractability" tradeoff (Levesque & Brachman 1987). Some of the KRSs with which we have used OKBC are highly expressive

and would require OKBC to support an equally expressive knowledge model to expose their full functionality. A highly expressive knowledge model, however, makes defining OKBC bindings for KRSs with limited functionality difficult and time consuming. Throughout the design of OKBC, this tradeoff was a guiding principle for carefully controlling the expressiveness of the knowledge model. We believe that expressiveness vs. generality is a fundamental tradeoff in knowledge sharing.

OKBC represents a major advance over its predecessor GFP. OKBC supports a larger class of KRSs: its knowledge model includes an assertional view of a KRS, provides an explicit treatment of KB entities that are not frames, has a stronger method to control inferences and has a better method for specifying default values. Unlike GFP, OKBC can be used on practically any platform and with a substantially larger range of applications because it supports network transparency, multiple programming languages, and a remote procedure language.

In summary, OKBC substantially advances our ability to achieve interoperability between KRSs and client-side KB tools. Its success is shown by its use with a broad range of systems, one of which is being used in a commercial environment. Availability of OKBC implementations in multiple programming languages makes it an attractive choice for the developers of applications that make use of the content and services provided by KRSs. They can have increased confidence that their applications will interoperate with multiple KRSs. With the availability of OKBC-compliant tools, KRS developers will be able to use off-the-shelf knowledge components that are not the primary focus of their work. We believe that OKBC makes a modest contribution toward achieving plug-and-play operation between KB tools and KRSs.

Acknowledgments

At Stanford University, this work was supported by the Department of Navy contracts titled *Technology for Developing Network-based Information Brokers* (Contract Number N66001-96-C-8622-P00004) and *Large-Scale Repositories of Highly Expressive Reusable Knowledge* (Contract Number N66001-97-C-8554). At SRI International, it was supported by a Rome Laboratory contract titled *Reusable Tools for Knowledge Base and Ontology Development* (Contract Number F30602-96-C-0332) and a DARPA contract entitled *Ontology Construction Toolkit*. The protocol has undergone revisions based on input from many people, including Fritz Mueller, Karen Myers, S. Paley, and Bob MacGregor.

References

Brachman, R.; Fikes, R.; and Levesque, H. 1983. KRYPTON: A functional approach to knowledge representation. *IEEE Computer* 16(10):67–73.

Brewka, G.; Dix, J.; and Konolige, K. 1997. *Non Monotonic Reasoning*. Cambridge University Press.

Chaudhri, V. K.; Farquhar, A.; Fikes, R.; Karp, P. D.; and Rice, J. P. 1997. Open Knowledge Base Connectivity 2.0. Technical Report KSL-98-06, Available from Knowledge Systems Laboratory, Stanford University.

Farquhar, A.; Iwasaki, Y.; Fikes, R.; and Bobrow, D. G. 1996. A compositional modeling language. In *Proceedings of the 1996 Qualitative Reasoning Workshop*.

Farquhar, A.; Fikes, R.; and Rice, J. P. 1997. A Collaborative Tool for Ontology Construction. *International Journal of Human Computer Studies* 46:707–727.

Geiger, K. 1995. *Inside ODBC*. Microsoft Press.

Genesereth, M. R., and Fikes, R. E. 1992. Knowledge Interchange Format, Version 3.0 Reference Manual. Technical Report Logic-92-1, Computer Science Department, Stanford University.

Karp, P.; Riley, M.; Paley, S.; and Pellegrini-Toole, A. 1996. EcoCyc: Electronic Encyclopedia of *E. coli* Genes and Metabolism. *Nuc. Acids Res.* 24(1):32–40.

Karp, P.; Myers, K.; and Gruber, T. 1995. The Generic Frame Protocol. In *Proceedings of the 1995 International Joint Conference on Artificial Intelligence*, 768–774.

Karp, P. 1992. The Design Space of Frame Knowledge Representation Systems. Technical Report 520, SRI International Artificial Intelligence Center.

Lenat, D., and Guha, R. 1990. *Building Large Knowledge-Based Systems: Representation and Inference in the CYC Project*. Addison-Wesley.

Levesque, H., and Brachman, R. 1987. Expressiveness and tractability in knowledge representation and reasoning. *Computational Intelligence* 3(2):78–93.

MacGregor, R., and Burstein, M. 1991. Using a description classifier to enhance knowledge representation. *IEEE Expert* 6(3):41–46.

Mitchell, T.; Allen, J.; Chalasani, P.; Cheng, J.; Etzioni, E.; Ringuette, M.; and Schlimmer, J. 1989. Theo: A framework for self-improving systems. In *Architectures for Intelligence*. Erlbaum. 323–355.

Paley, S. M.; Lowrance, J. D.; and Karp, P. D. 1997. A generic knowledge base browser and editor. In *Proceedings of the Ninth Conference on Innovative Applications of Artificial Intelligence*.

Rice, J. P., and Farquhar, A. 1998. OKBC: A Rich API on the Cheap. Technical Report KSL-98-09, Available from Knowledge Systems Laboratory, Stanford University.

Stede, M., and Umbach, C. 1998. Dimlex: A lexicon of discourse markers for text generation and understanding. Technical report, Technical University of Berlin, submitted for publication.

Wilkins, D. 1988. *Practical Planning: Extending the Classical AI Planning Paradigm*. San Mateo, CA: Morgan-Kaufmann Publishing.