The Generic Frame Protocol

Peter D. Karp Karen L. Myers
Artificial Intelligence Center
SRI International
333 Ravenswood Ave.
Menlo Park, CA 94025
Phone: 415-859-6375 Fax: 415-859-3735
pkarp@ai.sri.com myers@ai.sri.com

Tom Gruber
Knowledge Systems Laboratory
Stanford University
Stanford, CA 94305
gruber@ksl.stanford.edu

Abstract

The Generic Frame Protocol (GFP) accesses knowledge bases stored in frame knowledge representation systems (FRSs). GFP consists of a set of Common Lisp functions that provide a generic interface to these underlying FRSs. This interface isolates an application from many of the idiosyncrasies of specific FRS software and enables the development of generic tools (e.g., graphical browsers, frame editors) that operate on many FRSs. To date, GFP has been used as an interface to LOOM, Ontolingua, THEO, and SIPE–2.

Keywords: Knowledge Representation, Knowledge Sharing, Architectures

This paper has not already been accepted by and is not currently under review for a journal or another conference. Nor will it be submitted for such during IJCAI's review period.

The Generic Frame Protocol

Abstract

The Generic Frame Protocol (GFP) accesses knowledge bases stored in frame knowledge representation systems (FRSs). GFP consists of a set of Common Lisp functions that provide a generic interface to these underlying FRSs. This interface isolates an application from many of the idiosyncrasies of specific FRS software and enables the development of generic tools (e.g., graphical browsers, frame editors) that operate on many FRSs. To date, GFP has been used as an interface to LOOM, Ontolingua, THEO, and SIPE–2.

1 Introduction

Frame knowledge representation systems (FRSs) have a long history within the artificial intelligence (AI) knowledge representation (KR) community. FRSs have seen extensive use as information-management components for planners, expert systems, and natural-language understanding systems. A recent review has identified more than 50 FRSs, including the KL-ONE family (with members such as LOOM and BACK), and the UNITS family (with members such as CYCL and THEO) [5].

FRSs are in danger of extinction though, because of several serious limitations. They do not scale to large knowledge bases (KB). They do not support multiuser access. Most cannot provide networked, client-server operation. It is difficult to reuse the knowledge in a given FRS across applications, and it is difficult to reuse high-level knowledge base management utilities (such as KB browsers and editors) across a range of FRSs. In contrast, object-oriented databases do not suffer many of these limitations, and they duplicate many capabilities of FRSs. Their growing popularity may end all hope that FRSs can have significant impact in the general computing community.

Our groups at SRI and Stanford, and the AI knowledge-sharing community in general, are addressing many of these limitations. At SRI, Karp and Paley are integrating database storage technology and multiuser access capabilities into FRSs [6]. The knowledge-sharing community has undertaken a number of efforts aimed at supporting knowledge reuse, including development of shared portable ontologies in Ontolingua [4], and development of well-defined languages for knowledge interchange such as KIF [3]. Efforts such as Ontolingua and KIF provide knowledge reuse through a paradigm of *specification-time translation*. For example, given an ontology of bibliographic data encoded in Ontolingua, translators convert the Ontolingua encoding into the language of a specific FRS prior to the use of the FRS.

Just as knowledge reuse is extremely important, so is the ability to reuse, and to mix and match, the software components of a large AI system. For

example, consider the knowledge-management utilities (KMUs) that are often developed in conjunction with FRSs, such as graphical KB browsers and editors. Typically, these KMUs are built from scratch for a specific FRS, despite great overlap in the capabilities of KMUs for different FRSs (for instance, the literature shows a pattern of many KB editors with similar capabilities [1, 8]). The result is a tremendous duplication of effort. When the associated FRS falls into disuse, the KMU becomes worthless. The substantial investment involved in building a sophisticated KMU would generate a larger payoff if that KMU could operate with several FRSs.

The Generic Frame Protocol (GFP) is a set of Common Lisp functions that constitute a generic interface to FRSs. GFP provides a programmatic interface that a KMU or domain application can employ to retrieve, update, and create information in an FRS. GFP supports

- Access to knowledge from multiple FRSs using a single, uniform interface
- Portability with respect to FRSs, thus enabling switches from one FRS to another with minimal effort
- Reuse of KMU software, such as editors and browsers
- Reuse of knowledge through a paradigm of *runtime translation*

By runtime translation, we mean that GFP translates knowledge from a given FRS representation to a GFP-compatible form during the execution of an application. This approach contrasts with the specification-time approach to translation adopted by other knowledge-sharing efforts. In effect, GFP treats a KB and its associated FRS as a complete module that can be spliced into many applications, with translation occurring at runtime as required.

GFP has been designed as a procedural rather than a declarative interface to FRSs because procedural interfaces are more convenient in many cases, and can yield better performance (see Section 5.2).

2 Design Goals

Several design objectives were defined for GFP: **Simplicity:** The protocol should be simple and reasonably quick to implement for a particular FRS, even if this means sacrificing theoretical considerations or support for idiosyncrasies of that FRS. **Generality:** The protocol should apply to many FRSs, and support the most common FRS features. **No legislation:** The protocol should not require substantial changes to an FRS for which the protocol is implemented. That is, the protocol should not legislate the operation of an underlying FRS. **Performance:** Inserting the protocol between an application and an FRS should not introduce a significant performance cost. **Consistency:**

The protocol should exhibit consistent behavior across implementations for different FRSs, that is, a given sequence of operations within the protocol should yield the same result over a range of FRSs. **Precision:** The specification of the protocol should be as precise and unambiguous as possible. **Language independence:** Ideally, GFP should be programming-language-independent, but currently it has many COMMON LISP dependencies. Connecting GFP to an FRS implemented in some other language should be straightforward using LISP foreign-function calls, but implementing a user-callable GFP implementation in a different language would require significant effort.

Satisfying these objectives simultaneously is impossible because many of them conflict. Put another way, the very essence of GFP is involves compromise. Different FRSs behave differently, and unless we legislate a minutely detailed behavioral model for KR systems (which no developers will subscribe to anyway), we cannot force these systems to behave the same. So GFP is a study in the art of compromise because it requires a reference model that encompasses many FRSs and is detailed enough to be useful in practice, but is not so detailed as to exclude every FRS from its model. An example of conflicts among our objectives is that to precisely specify the semantics of the GFP function that retrieves the values of a slot, we must specify the inheritance semantics to be used. However, different FRSs use different inheritance mechanisms [5]. Conformance with a specific semantics for inheritance would require either altering the inheritance mechanism of a given FRS (violating the no-legislation goal), or emulating the desired inheritance mechanism within the implementation of the protocol (violating performance and generality, since the inheritance method used by that FRS is inaccessible through the protocol).

To address variations among key FRS operations, GFP identifies a set of dimensions — called *behaviors* — along which FRSs differ. The GFP implementation for a given FRS declares which behaviors are required for its correct execution, at which time the compatibility of an application and an FRS can be determined. In addition, an application or a KMU can be conditionalized on the behaviors of different FRSs. For example, the display of a frame created by a KB editor can conditionally retrieve and display facets or not, depending on whether the FRS in use supports facets or not.

3 Reference Model

A comprehensive survey of FRSs reveals a large variety of system designs [5]. Some of the differences among these systems are significant, while others are superficial. In defining the generic frame model that underlies GFP, we have attempted to identify those commonalities that are useful for a broad range of applications. Because of the large number of FRSs in existence, researchers often use different terminology to mean the same thing. Our reference model

for GFP employs the terms *knowledge base*, *frame*, *class*, *instance*, *slot*, and *facet*, each of which is described below.[1] The reference model is based on an axiomatic formalization of classes, relations, and functions for the Frame Ontology in Ontolingua [4] but extends the ontology to include aspects relevant to operational applications (e.g., kinds of inheritance, facets).

3.1 Representational Primitives

3.1.1 Frames

A *frame* is an object with which facts are associated. Each frame has a unique name. Frames are of two types: classes and instances. A *class* frame represents a semantically related collection of entities in the world. Each individual entity is represented by an *instance* frame. A frame can be an instance of many classes, which are called its *types*, and a class can be a type of many instances. A class can also be an instance, that is, an instance of a class of classes (a *metaclass*). The relation that holds between an instance and a class is primitive, akin to set membership.[2] With this primitive we can define the *sub* relation that holds between classes: a class C_{sub} is a sub of class C_{super} iff whenever i is an instance of C_{sub} then i is also an instance of C_{super}.

3.1.2 Knowledge bases

A *knowledge base*, or KB, is a collection of frames and their associated slots and values. Multiple KBs may be in use simultaneously within an application, possibly serviced by different FRSs. Frames in a given KB can reference frames in another KB, provided that both KBs are serviced by the same FRS.

3.1.3 Slots

Information is associated with a frame via *slots*. A slot is a mapping from a frame and a slot name to a set of values. A slot value can be any Lisp object (e.g., symbol, list, number, string). Slots can be viewed as binary relations; GFP does not support the explicit representation of relations of higher arity.

In some FRSs, slots are modeled as relations that cover the entire class of frames and it is possible to define *slot units*, which are frames that specify KB-wide properties of slots (such as constraints). GFP supports slot units; alternatively, slot properties may be declared locally in GFP for each frame (see Section 3.1.4).

[1] We use the following pairs of terms synonymously: *class* and *concept*, *instance* and *individual*, and *slot* and *role*.

[2] But not as powerful, since not all sets can be described as classes in FRSs.

3.1.4 Facets

Facets provide information about slots. In GFP, facets are identified by a facet name, a slot name, and a frame. A facet has as its values a set of data objects. Some facets pertain to the values of a slot; for example, a facet can be used to specify a constraint on slot values (see Section 3.2.2) or a method for computing the value of a slot. Other facets may describe properties of the slot itself, such as documentation. GFP supports only a single level of facets.

3.2 Inference

The GFP reference model includes three forms of inference, namely, *subsumption reasoning*, *constraint checking*, and *inheritance*.

The reference model assumes *domain closure* on instances; that is, an instance is defined for every object of interest in a given application domain. The model also assumes *predicate closure* on class and slot relations, that is, that the extensions of all such relations are fully specified in a given KB.

3.2.1 Subsumption

Subsumption reasoning is a key inferential capability in virtually all FRSs. For reasoning about subsumption relationships, GFP distinguishes *direct* from *all* relationships. We say that i is a *direct-instance-of* a class C if i is an instance of C and there is no other class C' that is a subclass of C such that i is an instance of C. *All-instance-of* is the transitive closure of direct-instance-of. Similarly, a class C_{sub} is a direct subclass of class C_{super} if C_{sub} is a subclass of C_{super} and there is no other class C' of which C_{sub} is a subclass and which is, in turn, a subclass of C_{super}. The relations *direct-super-of* and *all-super-of* are the inverses of *direct-sub-of* and *all-sub-of*. *All-types-of* is the inverse of all-instance-of, and *direct-type-of* is the inverse of direct-instance-of.

Some FRSs require *direct* relationships to be specified when frames are created. In contrast, FRSs that perform automatic classification may infer the direct relationships by comparing class definitions. In GFP, direct relationships must be specified at frame creation time. GFP operations allow the user to interrogate any of these specified class–subclass and class–instance relationships, no matter how the relationships were derived.

3.2.2 Constraint Checking

A KB often must contain not only ground facts about instances, but also general rules that constrain the valid relationships among classes and instances. These constraints may be checked to assure that the KB remains logically consistent when changes are made. Should a constraint be violated, a representation system may signal an error to the user or take steps to return the

KB to a consistent state. Most FRSs support a limited form of constraints, the most common of which are slot constraints. Slot constraints constrain the possible values that a slot can be assigned. GFP supports two specific slot constraints: type and number restrictions on slot values. They are specified using common facet names that are included in the protocol (see Section 3.3 for details).

3.2.3 Slot Value Inheritance

A slot maps a particular frame to a set of values. For perspicuity, FRSs generally allow the user to describe a set of such mappings for all instances of a class. The instances are then said to inherit those slot values from the class. Many FRSs further augment inheritance by allowing slots to specify default values; such values are to be inherited only when they do not conflict with local information available for a frame.

Inheritance in GFP is based on the use of *template* and *own* slots. A template slot is associated with a class frame, but applies to all instances of that class. In many FRSs, template slots are presented as if they were actually slots. However, they are really a way of specifying, in one place, slots for all the instances of that class. For example, if we wanted to say that all instances of the class `female-person` have a slot called `gender` with the value `female`, we could define a template slot called `gender` for the `female-person` frame and give it value `female`. Then if we created an instance of `female-person` called `mary`, and we asked for the value of the slot `gender` on `mary`, we would be told that her gender is `female`. What would we get if we asked for the value of the `gender` slot of `female-person`? The question is ambiguous, because we could mean the template slot on the frame `female-person` viewed as a class, or the slot on the frame viewed as an instance (e.g., of a class of classes). In the latter case, this slot is referred to as an own slot. GFP allows the user to declare a slot as either `own` or `template`.

Inheritance in GFP can be characterized as follows. An own slot may have *local values*, which are asserted directly for that slot. The values for slot S of a frame f are determined by "combining" the local values and template values of $C.S$ for any class C that is a superclass of f, provided those values do not "conflict". GFP allows the use of different semantics for "combining" and "conflict", to support a range of inheritance methods. Selection of a specific inheritance mechanism is controlled through declarations for the behavior `inheritance`, as described in Section 4.

Currently, the protocol makes no commitment (either directly or through behaviors) regarding the inheritance of values for frames that have multiple direct superclasses. Thus, applications should not depend on a specific semantics for inheritance in such cases.

3.3 Common Names

To insulate applications from meaningless variability among the names of frequently used objects in different FRSs, GFP specifies common names for certain frames, slots, and facets. Implementations of the protocol must translate these names into the appropriate FRS-specific objects. The common names for GFP are taken from the frame ontology (in which they are defined by a formal axiomatization [4]) and the basic data types in KIF [3].

For example, the common frame names include `number` (which has the frame `integer` as a subclass) and `sequence` (which has the frames `list`, `string`, and `array` as subclasses). The frame `class` is defined as the all-super-of all frames that denote a class.

4 Behaviors

Although GFP necessarily imposes some common requirements on the organization of knowledge (KBs, frames, slots, facets) and semantics of some assertions (instance and subclass relationships, inherited slot values, slot constraints), it allows for some variety in the operation of underlying FRSs. This diversity is supported through behaviors, which provide explicit models of the FRS properties that may vary. At the time of declaration, the protocol determines whether the behaviors required by the application can be supplied. Those behaviors might already be present in the underlying FRS, or the protocol itself might emulate them on behalf of the FRS. In cases where one or more of the required behaviors cannot be supplied, the protocol issues warnings to this effect. The GFP behaviors have a second role, namely to configure the operation of an FRS at runtime. For an FRS that can provide more than one of several alternative functionalities, the behaviors allow the user to specify how the FRS should operate at a given time.

Here we describe the behaviors defined currently in GFP. We expect that additional behaviors will be supported in future versions of the protocol. It is important to note that an implementation of the protocol for a given FRS might not provide every possible behavior because of both the variability among FRSs and the complexity of GFP.

The behavior `:facets` determines whether the FRS supports facets (see Section 3.1.4). The behavior `:class-slot-types` is used to indicate the slot types for classes supported by a given FRS, either template only or both template and own (see Section 3.2.3). Testing the equality of slot and facet values is a common operation within FRSs, although the nature of the test used varies. For this reason, GFP supports a behavior `:default-test-fn` for specifying an FRS-dependent function to be used for these value comparisons.

The behavior `:inheritance` can be used to specify the model of inheritance used by the FRS. Two possibilities are currently supported:

override The presence of any local value in a given slot of a frame blocks inheritance of any values for that slot from superclasses of the frame.[3]

incoherence A slot inherits from its superclasses all values that do not violate any constraint associated with the slot.

Imagine that slot `color` records all colors visible on the surface of an animal, and that the default at class `Elephant` for `color` is `gray`. Suppose that the elephant `Clyde` has `blue` as a local value if `color`, to reflect the color of Clyde's eyes. For the `override` inheritance semantics, the user-visible value of `Clyde.color` would be `blue`, whereas for `incoherence`, `Clyde.color` would be {`blue,gray`}. In the first case, the local value blocks inheritance of the default value, whereas in the second case, inheritance is not blocked because no constraint specifies that `gray` and `blue` are inconsistent values. If we further specified that `color` is a single-valued slot using a cardinality slot constraint, then the user-visible value of the slot under `incoherence` would be `blue`.

We note that *incoherence* semantics makes no commitment in cases where inheritance from all superclasses of a given frame leads to inconsistency, but not from some subsets of those supers. This vagueness is intentional to enable more inheritance mechanisms to satisfy the *incoherence* semantics, with the caveat that the FRSs should not depend on inherited values for such cases.

5 The Generic Frame Protocol

This section summarizes the operations that comprise the Generic Frame Protocol, and describes the process of implementing the protocol for a new FRS.

5.1 Programmatic Interface

The Generic Frame Protocol defines a programmatic interface of common operations that span the different object types in the reference model, namely, *knowledge bases*, *frames*, *classes*, *instances*, *slots*, and *facets*. There are three main categories of operations supported for each object type: *retrieval operations*, *manipulator operations*, and *iterators*. Retrieval operations extract information about objects and object values. Retrieval operations generally come in two forms: functional operations, which retrieve a value, and relational operations, which test whether a relation holds between an object and some value(s). Manipulator operations create, destroy, and modify objects.

GFP supports three kinds of iterators: `do-<object>-<reln>`, `map-<object>-<reln>`, and `mapc-<object>-<reln>`, where `<object>` ranges over the object types of the reference model, and `<reln>` specifies a class of objects related to `<object>`

[3] This form of inheritance is sometimes referred to as *specificity* inheritance.

in some manner (e.g., `do-class-direct-subs` and `mapcar-kb-classes`). The iterators are included in GFP to support access to efficient iteration mechanisms that underlying FRSs may provide, rather than using the straightforward approach of first consing up a list of the objects to be iterated over, then iterating through the list.

5.1.1 Operations on KBs

New KBs are created in GFP through the `create-kb` operation. A parameter of this operation is a CLOS class corresponding to the underlying FRS to be used for the new KB, such as the class `loom-kb` or `theo-kb`. `Create-kb` returns a CLOS instance of that class — the KB descriptor. This descriptor is the handle for all subsequent access to the KB.

In GFP, there is a notion of a current KB. All GFP operations apply to the current KB by default, unless a different KB description is specified for an operation. Additional KBs can be accessed; the notion of the current KB is one of convenience, as it defines a default context for GFP operations.

GFP provides functions for storing and retrieving KBs to and from secondary storage. Its model of access allows KBs to reside on a variety of storage types, including traditional flat files and database systems on remote servers.

5.1.2 Operations on Frames, Classes, and Instances

GFP includes manipulator operations to create, copy, delete, rename, and print frames. Retrieval operations retrieve the name and slots of a given frame and test the type of a frame (whether it is a class or an instance), the containment of a frame within a designated KB, and the coercion of a frame name to a frame object (possibly relative to a KB). Iterators are provided for the slots of a frame.

Manipulators create classes and instances, and retrieval operations test all possible subsumption relationships between classes and instances, for both the direct and all relationships described in Section 3.2.1. Iterators are defined for all subsumption relationships (i.e., `direct-subs`, `all-subs`, `direct-supers`, `all-supers`, `direct-types`, `all-types`, `direct-instances`, `all-instances`). Additional operations test the equivalence, consistency, and disjointedness of classes, determine whether a given class is primitive (in the sense of classification), and determine the most specific/general classes from a list of classes.

5.1.3 Operations on Slots and Facets

Manipulation operations for slots add, remove, or replace a value, or replace the entire set of slot values. A complete slot value can be retrieved, or checks can be made to see whether the slot contains one particular value. GFP can also retrieve the type of a slot (either `template` or `own`). Iterators are provided

for both the facets and values of a slot. Two additional operations determine whether a frame has a slot with a given name (`slot-p`) and follow a chain of slots to retrieve a value (`follow-slot-chain`).

For facets, the manipulation operations add, remove, or replace values. Retrieval operations can obtain all values for a facet, or test for membership of a particular value for a facet. Iteration over facet values is supported. GFP can also determine whether a frame has a facet with a given name.

5.1.4 Operations on Behaviors

Retrieval operations obtain information about the behaviors supported by GFP in general, the behaviors that a given FRS supports, and the behaviors that are enabled for a particular KB.

5.2 Programmatic vs. Declarative

The procedural/declarative controversy, a long-standing issue in AI, also arises in relation to interfaces for FRSs. Many authors argue that declarative Tell/Ask style interfaces [9] are preferable because of their simplicity and well-defined logical semantics; others believe that procedural interfaces provide more natural and efficient interactions and can be ascribed comparable semantics.

We believe that both kinds of interface are of value, depending on the situation at hand. Declarative interfaces are useful in interactive settings and for formulating complex queries, whereas procedural interfaces are preferable when embedded within other software. Our group employs both the procedural interface embodied by GFP, and a declarative, first-order query facility that is implemented on top of GFP (see Section 7). We have not built a declarative assertional facility (i.e., no Tell interface).

We view a programmatic interface as essential to an operational KR system for the following reasons:[4] (1) Although the procedural and declarative approaches provide equivalent expressive power in principle, it is more convenient to represent certain constructs procedurally. For example, combining quoted and evaluated terms is trivial in a procedural system but awkward in a declarative system.[5] (2) KMU tools often need access to meta-information about knowledge; for instance, a KB browser may need to determine whether a slot on a frame has a value in order to determine how to display the frame. The GFP programmatic interface provides an explicit function `slot-has-value-p` for accessing this information. Other useful metalevel relationships that are testable directly in the GFP programmatic interface include whether a given

[4]It is interesting to note that LOOM users clamored for a programmatic interface to be added to the original declarative interface provided for that system.

[5]Indeed, Tell/Ask interfaces rarely provide the same range of capabilities as their procedural counterparts because of this awkwardness.

facet has a value; whether a name denotes a frame, class, instance, or facet; what slots exist for a given frame; and which values of a slot were inherited rather than asserted locally. (3) Metalevel queries must execute fast for interactive applications, but are likely to be answered extremely slowly when proof techniques are used. Indeed, a query to distinguish inherited from local slot values would not be expressible in most Tell/Ask interfaces, but can be useful in understanding how various KB inferences were derived.

5.3 GFP Implementation

Users access all GFP operations as either functions or macros. At the implementation level, most of the functions and macros call a generic function to do their work — an FRS-specific method implements the operation. The extra level is introduced to allow default values for arguments to be supplied, and to allow keyword arguments (neither of which is provided by CLOS generic functions). Every generic function dispatches on an argument called `kb`, which is a KB descriptor (a CLOS instance) that defaults to the current KB but can be overridden (as a keyword argument to the function wrapper around the generic function).

A set of FRS-specific methods implement the GFP operations for each FRS. Therefore, to provide a GFP implementation for a new FRS, we provide a new module of methods for that FRS. This module need not provide a method for every GFP operation, only for operations in a small *kernel* of GFP. GFP provides default methods for all operations outside the kernel, which are defined in terms of operations inside the kernel. For example, the default method for `slot-has-value-p` calls the kernel operation `get-slot-values`. The kernel consists of roughly 30 operations. The default methods can of course be overridden to improve efficiency or for better integration with development environments. Their purpose is to simplify the GFP implementation for new FRSs. Two other connections must be made between GFP and a given FRS, namely, the linking of common GFP object names with the appropriate FRS objects, and the specification of the appropriate behaviors for the FRS.

6 FRSs Supported by GFP

To date, there exist four FRS-specific implementations of GFP, for Loom [10], Theo [11], SIPE–2 [13], and Ontolingua [4]. Figure 1 summarizes the behaviors supported for each of these implementations. These FRSs cover a broad range of capabilities, from classification-based to nonclassificatory.

LOOM LOOM fits the GFP model fairly closely, but we note two exceptions. First, LOOM instances do not differentiate local from inherited values. Second, attributes of LOOM classes are specified through complex definition

Behavior	Loom	Ontolingua	Theo	Sipe-2
facets	yes	yes	yes	no
inheritance	incoherence	override	override	override
class-slot types	template	template and own	template	template
default-test-fn	equal	equal	equal	equal

Figure 1: GFP Behavior Specifications for Current FRS Implementations

expressions. LOOM has no notion of incremental redefinition for a facet of a template slot in a class, instead requiring the user to issue an entire new definition when only a small change occurs (such as changing a default value). The GFP methods for LOOM translate between facets and definitions, submitting an entire new definition to LOOM when any facet changes.

Ontolingua Ontolingua is primarily a translation and analysis tool for ontologies, but its most recent version (4.0) includes a limited frame system. By using this embedded FRS, it is possible to write portable KIF ontologies and store or access them as knowledge bases using GFP. The Ontolingua implementation supports the full range of GFP functionality.

THEO THEO is a conceptual descendant of RLL that fits the GFP model fairly closely; exceptions are a simplified multiple-KB system and the use of facets within facets to an arbitrary depth (not supported by GFP).

SIPE-2 The SIPE–2 planner includes a simple frame-style knowledge representation system that corresponds to a restricted subset of the GFP model. For instance, it has no KB operations.

We expect that GFP interfaces to other FRSs (such as CLASSIC or BACK) would be no more difficult to implement than those we have described.

7 Knowledge Management Utilities

Our group is building a collection of generic knowledge management utilities (KMUs). Because these tools are implemented on top of GFP, they can be used in conjunction with any GFP-compatible FRS.

Graphical KB Browsers and Editors We are developing a suite of graphical tools for interactive KB browsing and editing. The various tools provide different visualizations of the information within a KB and different editing operations. One tool presents the class/subclass/instance hierarchy as a graph, with incremental expansion of nodes to support exploration of large KBs. An-

other tool graphs arbitrary relationships among frames as a semantic network. The third tool displays the slots and facets of an individual frame. A fourth tool shows selected slot values for a set of frames in tabular form. Prototype implementations of the first three tools exist; their development is ongoing.

Query Processor GFP provides application programmers with an efficient procedural interface for accessing frame-based knowledge bases. As a complementary method, we have implemented a GFP-based query processor that provides a declarative interface for extracting information from a knowledge base. The query processor supports an extended first-order query language that constitutes a restricted version of KIF. To date, the query processor has been used in conjunction with LOOM and THEO knowledge bases.

8 Discussion

8.1 Adequacy of the Model

The GFP reference model encompasses many but not all of the capabilities of current-generation FRSs. It does not provide rules or methods. It does not provide all operations required by classificatory FRSs; in particular, there is no explicit concept-definition language (although facets provide a means of building concept definitions in a structural way). The KRSS specification [12] is a good candidate upon which to base such a concept-definition language. The GFP model of KBs is simplistic because it has no notion of dependencies or imports between KBs. No operations are defined for changing class–subclass or class–instance relationships. We welcome input from the KR community in addressing these issues, and we hope to see other groups adopt GFP for use in conjunction with other FRSs.

8.2 The Price of Generality

Our experimental evaluations indicate that the performance penalty for using GFP is reasonable. Using a LOOM implementation of GFP, we compared the running times of key GFP kernel operations with their corresponding LOOM operations. The results showed the GFP operations to be 1% – 50% slower (depending on the operation). The high overhead costs resulted for operations without direct counterparts in LOOM. For example, GFP provides an operation for retrieving a frame when given an identifier; LOOM has no such operation, instead providing separate operations for instances and classes. The GFP operation must consider whether the name corresponds to a class or an instance in order to invoke the appropriate underlying LOOM operation. We note that on an absolute scale, the overhead is very small in this case (approximately .02 milliseconds).

For directly comparable operations, the upper-bound on overhead was 35%. Much of the increased execution time results from activities common to all GFP operations. Thus, the overhead is high on a percentage basis for fast operations such as slot value retrievals (35% for a .3 millisecond operation), but low for more expensive operations such as retrieving all instances of a class (1% for a 16 millisecond operation).

8.3 Relation to Knowledge-Sharing Efforts

Both GFP and KIF [3] seek to provide a domain-independent medium that supports the portability of knowledge across applications and storage. GFP is narrower in representative scope than KIF: Whereas KIF is intended to be a comprehensive first-order representation formalism, GFP is focused on the representation of class hierarchies. Ontolingua [4] is a set of tools for writing and analyzing KIF knowledge bases along with translators for mapping KIF KBs to specific FRSs. KIF and Ontolingua are declarative representation languages; GFP is a procedural interface for accessing representation structures. KIF and Ontolingua are designed for use in sharing a large corpus of knowledge at specification time, through the use of translators. GFP is designed for runtime access to and modification of existing KBs. GFP is similar to KQML [2] in that it provides a set of operations defining a functional interface for use by application programs. However, the GFP operations are grounded in knowledge representation structures, while KQML operations correspond to performatives for agent execution.

9 Conclusions

In addition to supporting the development of the KMUs described in Section 7, GFP is in use in two applications. The SIPE–2 planning system can access static information about a planning domain via GFP; it has successfully solved military transportation planning problems for which the planning domain is defined in a LOOM KB. In addition, the EcoCyc project at SRI has constructed a large KB and associated graphical user interface of *E. coli* genes and biochemistry. All code for managing the THEO EcoCyc KB, and for accessing the KB from the graphical user-interface, employs GFP.

Future work related to GFP can go in several directions. One option is to extend GFP to provide greater coverage of FRS features. A second direction is to extend GFP to support access to KBs distributed across a network.

References

[1] G. Abrett, M. Burstein, J. Gunsbenan, and L. Polanyi. KREME: A user's

introduction. Technical Report 6508, BBN Laboratories Inc., Cambridge, MA, 1987.

[2] T. Finin *et al.* Specification of the KQML Agent-Communication Language. Technical Report EIT TR 92-04, Enterprise Integration Technologies, Palo Alto, CA, 1992.

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

[4] T.R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, 1993.

[5] P.D. Karp. The design space of frame knowledge representation systems. Tech. Report 520, SRI International AI Center, 1992.

[6] P.D. Karp, S.M. Paley, and I. Greenberg. A storage system for scalable knowledge representation. In *Proceedings of the Third International Conference on Information and Knowledge Management*, 1994.

[7] P.D. Karp and T. Gruber. A generic knowledge-base access protocol. Technical report, AI Center, SRI International, 1994. Available via World Wide Web URL http://www-ksl.stanford.edu/knowledge-sharing/lib/gfp/spec/paper.dvi.

[8] T.P. Kehler and G.D. Clemenson. KEE the knowledge engineering environment for industry. *Systems and Software*, 3(1):212–224, 1984.

[9] H. J. Levesque. Foundations of a functional approach to knowledge. *Artificial Intelligence*, 23:155–212, 1984.

[10] R. MacGregor. The evolving technology of classification-based knowledge representation systems. In J. Sowa, editor, *Principles of semantic networks*, pages 385–400. Morgan Kaufmann Publishers, 1991.

[11] T.M. Mitchell, J. Allen, P. Chalasani, J. Cheng, E. Etzioni, M. Ringuette, and J.C. Schlimmer. Theo: A framework for self-improving systems. In *Architectures for Intelligence*. Erlbaum, 1989.

[12] P. Patel-Schneider and B. Swartout. Description Logic Specification from the ARPA KRSS Effort. In preparation.

[13] D.E. Wilkins. Can AI planners solve practical problems? *Computational Intelligence*, 6(4):232–246, 1990.