Knowledge Systems Laboratory
Technical Report KSL 92-71

September 1992
Revised April 1993

A Translation Approach to Portable Ontology Specifications

by

Thomas R. Gruber

To appear in *Knowledge Acquisition*, 1993.

KNOWLEDGE SYSTEMS LABORATORY
Computer Science Department
Stanford University
Stanford, California 94305

A Translation Approach to Portable Ontology Specifications

Thomas R. Gruber

Knowledge System Laboratory
Stanford University
701 Welch Road, Building C
Palo Alto, CA 94304
gruber@ksl.stanford.edu

Abstract

To support the sharing and reuse of formally represented knowledge among AI systems, it is useful to define the common vocabulary in which shared knowledge is represented. A specification of a representational vocabulary for a shared domain of discourse — definitions of classes, relations, functions, and other objects — is called an ontology. This paper describes a mechanism for defining ontologies that are portable over representation systems. Definitions written in a standard format for predicate calculus are translated by a system called Ontolingua into specialized representations, including frame-based systems as well as relational languages. This allows researchers to share and reuse ontologies, while retaining the computational benefits of specialized implementations.

We discuss how the translation approach to portability addresses several technical problems. One problem is how to accommodate the stylistic and organizational differences among representations while preserving declarative content. Another is how to translate from a very expressive language into restricted languages, remaining system-independent while preserving the computational efficiency of implemented systems. We describe how these problems are addressed by basing Ontolingua itself on an ontology of domain-independent, representational idioms.

1. Introduction

A body of formally represented knowledge is based on a *conceptualization*: the objects, concepts, and other entities that are presumed to exist in some area of interest and the relationships that hold them (Genesereth & Nilsson, 1987). A conceptualization is an abstract, simplified view of the world that we wish to represent for some purpose. Every knowledge base, knowledge-based system, or knowledge-level agent is committed to some conceptualization, explicitly or implicitly.

An *ontology* is an explicit specification of a conceptualization. The term is borrowed from philosophy, where an ontology is a systematic account of Existence. For knowledge-based systems, what "exists" is exactly that which can be represented. When the knowledge of a domain is represented in a declarative formalism, the set of objects that can be represented is called the universe of discourse. This set of objects, and the describable relationships among them, are reflected in the representational *vocabulary* with which a knowledge-based program represents knowledge. Thus, we can describe the ontology of a program by defining a set of

representational terms. In such an ontology, definitions associate the names of entities in the universe of discourse (e.g., classes, relations, functions, or other objects) with human-readable text describing what the names are meant to denote, and formal axioms that constrain the interpretation and well-formed use of these terms.

This paper addresses the problem of portability for ontologies. Portability is a problem because the parties to a common ontology may use different representation languages and systems. Ideally, shared terms should be defined at the knowledge level, independent of specific representation languages. Of course, definitions need to be couched in *some* common formalism if they are to be shareable by knowledge-based applications. However, it is not realistic or desirable to require that those applications be *implemented* in a common representation language or system. This is because different applications require different kinds of reasoning services, and special-purpose languages to support them. Thus, the portability problem for ontologies is to support common ontologies over multiple representation systems.

We describe a translation approach to the portability problem for ontologies. In a translation approach, ontologies are specified in a standard, system-independent form and translated into specific representation languages. In Section 3 we will describe Ontolingua, an implemented system for translating ontologies from a declarative, predicate-calculus language into a variety of representation systems. In Section 4 we discuss the strengths and limitations of the approach. But first, we define the notion of a shared ontology and describe the role of ontologies for sharing knowledge among AI systems.

2. Ontologies and knowledge sharing

Knowledge-based systems and services are expensive to build, test, and maintain. A software engineering methodology based on formal specifications of shared resources, reusable components, and standard services is needed. We believe that specifications of shared vocabulary can play an important role in such a methodology.

Several technical problems stand in the way of shared, reusable knowledge-based software. Like conventional applications, knowledge-based systems are based on heterogeneous hardware platforms, programming languages, and network protocols. However, knowledge-based systems pose special requirements for interoperability. Such systems operate on and communicate using statements in a formal knowledge representation. They ask queries and give answers. They take "background knowledge" as an input. And as agents in a distributed AI environment, they negotiate and exchange knowledge. For such knowledge-level communication, we need conventions at three levels: representation language format, agent communication protocol, and specification of the content of shared knowledge. Proposals for standard knowledge representation formats (Fulton, 1992; Genesereth & Fikes, 1992; Morik, Causse, & Boswell, 1991; Spiby, 1991) and agent communication languages (Finin et al., 1992) are independent of the content of knowledge being exchanged or communicated. Ontologies can be used for conventions of the third kind: *content-specific* specifications (Gruber, 1991).

Current research is exploring the use of formal ontologies for specifying content-specific agreements for a variety of knowledge-sharing activities (Allen & Lehrer, 1992; Bradshaw, Holm, & Boose, 1992; Cutkosky et al., 1993; Fikes, Cutkosky, Gruber, & van Baalen, 1991; Genesereth, 1992; Gruber, Tenenbaum, & Weber, 1992; Neches et al., 1991; Patil et al., 1992;

Walther, Eriksson, & Musen, 1992). A long-term objective of such work is to enable libraries of reusable knowledge components and knowledge-based services that can be invoked over networks.

Consider the problem of reusing a knowledge-based planning program. Such a program takes descriptions of objects, events, resources, and constraints, and produces plans that assign resources and times to objects and events. Although it may use general planning algorithms, like all knowledge-based systems the planner depends on a custom knowledge base (sometimes called a "domain theory" or "background knowledge") to get the job done. The knowledge base may contain some knowledge generic to the planning task, and some that describes the domain situations in which the planner is to run.

If one wished to use the planning system, one would need to adapt an existing knowledge base to a new application domain, or build one from scratch. This requires, at a minimum, a formalism that enables a human user to represent the knowledge so that the program can apply it. Furthermore, the developer needs to know the kinds of information given in inputs and returned as outputs, and the kinds of domain knowledge that is needed by the planner to perform its task. If the planning program were offered as a service that could be invoked over the network, or if a large planning problem were subcontracted out to several cooperating planning agents, then one would need an agreement about the topics of conversation that agents are expected to understand.

Underlying these content-specific agreements are *ontological commitments:* agreements about the objects and relations being talked about among agents, at software module interfaces, or in knowledge bases. For instance, developers and users of the planning system might agree on the definitions of objects that "provide," "require," and "produce" resources; time that comes in "points" and "durations"; resources that are "allocated," "deallocated," and "produced" in "events"; and plans consisting of "state descriptions", "preferences" over states, and "effects" prescribed by "causal rules", events, or "operators."[1]

Ideally, we would like to be able to specify ontological commitments at the knowledge level (Newell, 1982). We say that an agent commits to a knowledge-level specification if its observable actions are logically consistent with the specification. This means that the specification format is independent of the internal (symbol-level) representation of the agent. With a bit of anthropomorphic license, the knowledge-level perspective can be applied to programs and knowledge bases. The "actions" of programs are "observed" through inputs and outputs, and specifications about their behavior may be given in terms of the formal arguments of a functional interface. Similarly, knowledge bases can be observed through a *tell and ask* functional interface (Levesque, 1984), where a client interacts with a knowledge base by making logical assertions (tell) posing queries (ask). The agents doing the telling and asking are viewed as black boxes.

In the context of multiple agents (including programs and knowledge bases), a *common ontology* can serve as a knowledge-level specification of the ontological commitments of a set of participating agents. A common ontology defines the vocabulary with which queries and assertions are exchanged among agents. For example, the words in the planning ontology are

[1]The terms is this example are taken from a common ontology defined for the DARPA/Rome Planning and Scheduling Initiative (Allen & Lehrer, 1992).

technical terms that govern the form of inputs and the interpretation of outputs. The definitions tell the user of a planning system what information must be given about an "event" or "resource" in order for the planner to be able to use the information. Each program must commit to the semantics of the terms in the common ontology, including axioms about the properties of objects and how they are related.[2] However, there need be no commitment to the form or content of knowledge *internal* to the agent.

The axiomatization in an ontology need not be a complete functional specification of the behavior of an agent. Common ontologies typically specify only some of the formal constraints that hold over objects in the input and output in the domain of discourse of a set of agents. They do not say which queries an agent is guaranteed to answer. Thus, a commitment to a common ontology is a guarantee of consistency, but not completeness, with respect to queries and assertions using the vocabulary defined in the ontology. Specifying the inferential capabilities of agents is an open research problem.

Definitions in common ontologies are like the global type declarations in a conventional software library, and the ontological commitments are specified with type restrictions over the inputs and outputs of program modules. Formal argument restrictions can be checked mechanically by compilers, to help ensure that calling procedures pass legal data to called procedures. Similarly, sentences in a tell-and-ask exchange can be checked for logical consistency with the definitions in ontologies. Just as the formal argument list hides the internal workings of a procedure from its environment, a common ontology allows one to interact with a knowledge-based program without committing to its internal encoding of knowledge.

Ontologies are also like conceptual schemata in database systems. A conceptual schema provides a logical description of shared data, allowing application programs and databases to interoperate without having to share data structures. While a conceptual schema defines relations on *Data*, an ontology defines terms which with to represent *Knowledge*. For present purposes, one can think of Data as that expressible in ground atomic facts and Knowledge as that expressible in logical sentences with existentially and universally quantified variables. An ontology defines the vocabulary used to compose complex expressions such as those used to describe resource constraints in a planning problem. From a finite, well-defined vocabulary one can compose a large number of coherent sentences. That is one reason why vocabulary, rather than form, is the focus of specifications of ontological commitments.

There are many aspects of the knowledge sharing problem that are not addressed by common ontologies. Questions not addressed include how groups of people can reach consensus on common conceptualizations, and how terms can be defined outside their context of use. The utility of common ontologies as a sharing mechanism is a hypothesis, the subject of collaborative studies (Neches et al., 1991; Patil et al., 1992). Ontolingua is a tool to support such research.

[2]Of course, it is the programmers of those systems who commit to using terms consistently, and "share the meanings" of the terms.

3. Ontolingua: A system for portable ontologies

Ontolingua is a system for describing ontologies in a form that is compatible with multiple representation languages. It provides forms for defining classes, relations, functions, objects, and theories. It translates definitions written in a standard, declarative language into the forms that are input to a variety of implemented representation systems. Ontologies written in Ontolingua can thereby be shared by multiple users and research groups using their own favorite representation systems, and can be ported from system to system. Figure 1 shows some of the possible applications of a single ontology translated into several systems.

The syntax and semantics of Ontolingua definitions are based on a notation and semantics for an extended version of first-order predicate calculus called Knowledge Interchange Format (KIF) (Genesereth & Fikes, 1992). KIF is intended as a language for the publication and communication of knowledge. It is intended to make the *epistemological-level* (McCarthy & Hayes, 1969) content of a knowledge base clear to the reader, but *not* to support automated reasoning in that form. It is very expressive, designed to accommodate the state of the art in knowledge representation. But it is not an implemented representation system.

Ontolingua translates definitions written in KIF into forms appropriate to implemented representation systems, which typically impose a restricted syntax and support limited reasoning over a restricted subset of full first order logic. Currently, Ontolingua can translate into Loom (MacGregor, 1991) (a KL-ONE style system), Epikit (Genesereth, 1990) (a predicate calculus language descended from MRS), Algernon[3] (Crawford & Kuipers, 1989) (a frame system based on Access Limited Logic), a generic class and slot syntax (for easy translations into simple frame systems), and pure KIF (i.e., in a canonical form, to facilitate further translation into other languages). It would be relatively simple to write translators into CycL (Lenat & Guha, 1990), KEE (Fikes & Kehler, 1985), or EXPRESS (Spiby, 1991). Ontolingua can parse arbitrary KIF sentences, and recognizes many common representation idioms. However, not all KIF sentences can be translated into all target representation systems. When definitions contain sentences that cannot be translated into a given implementation, Ontolingua informs the user and continues.

The set of idioms that Ontolingua can recognize and translate is defined in an ontology, called the Frame Ontology. The Frame Ontology specifies, in a declarative form, the representation primitives that are often supported with special-purpose syntax and code in object-centered representation systems (e.g., classes, instances, slot constraints, etc.). We discuss the Frame Ontology in Section 3.6.

[3]The original Ontolingua-to-Algernon translator was implemented by Juan Carlos Martinez at MCC. An extended, updated translator is under development by James Crawford and the author.

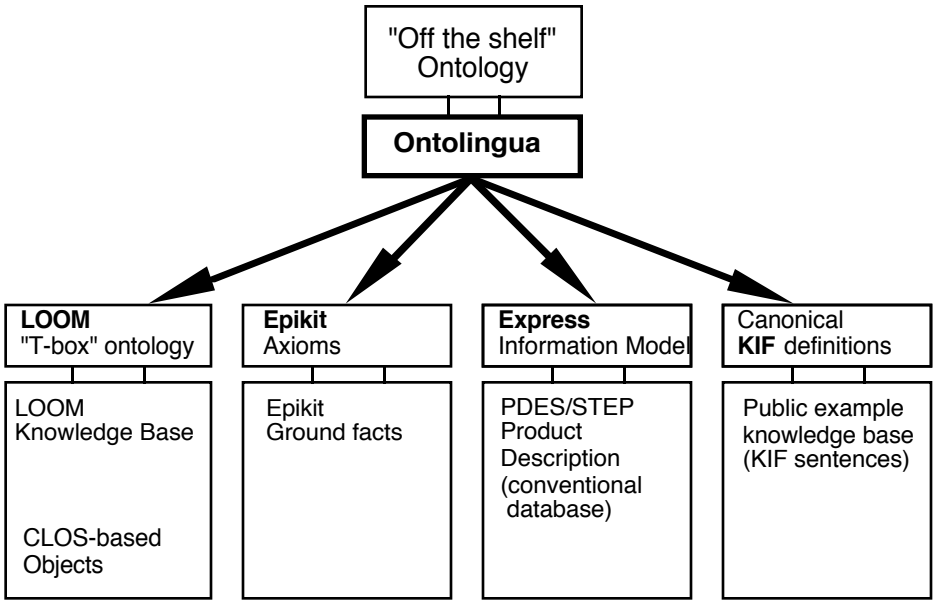

Figure 1: Translating from a common ontology into several applications. The same ontology can be used for different purposes in different systems. Loom can be used during the conceptual design of the ontology, helping with classification. It can also be used to manage a knowledge base of facts about objects. Epikit is useful for exploratory reasoning; it provides a suite of powerful deductive engines. Express is the standard language for describing PDES information models, which are logical database designs for sharable product description data. Other public-domain knowledge bases, such as mode libraries and experimental data sets, can be shared via a common ontology in a canonical form of KIF definitions.

3.1 A brief introduction to the syntax and semantics

Ontolingua definitions are written in natural language and KIF sentences. The natural language text is not parsed or translated, but is passed on to those systems that have a place for documentation (surprisingly, many representation systems do not!). The technical task is to translate declarative sentences, which are axioms that constrain the meaning of the defined terms.

KIF provides a Lisp-like notation for writing the axioms in Ontolingua definitions; it is a case-insensitive, prefix syntax for predicate calculus with functional terms and equality. Objects are denoted by object constants (Lisp atoms), or term expressions, which are constructed from lists whose first element is a function constant. Sentences are formed from lists whose first element is a relation constant and remaining elements are terms, or by logical operations over such sentences. Individual variables that may be universally or existentially quantified are marked with the '?' prefix. There are operators for conjunction, disjunction, implication, and negation. For example, the following is a KIF sentence that says, “All writers are misunderstood by some reader.”

```
(forall ?W
   (=> (writer ?W)
       (exists (?R ?D)
          (and (reader ?R)
               (document ?D)
               (writes ?W ?D)
               (reads ?R ?D)
               (not (understands ?R ?D))))
```

The symbol `writer` is a relation constant, and the sentence `(writer ?W)` says that `?W` is a writer. `=>` is material implication. `?W` is a universally quantified variable, and `?R` and `?D` are existentially quantified. `Reads` is a binary relation, and `(reads ?R ?D)` says that `?R` reads the document `?D`. Quantification is not typed or sorted, although in practice the types of quantified variables such as `?W` and `?D` are constrained with unary predicates such as `writer` and `document`.

The KIF specification (Genesereth & Fikes, 1992) provides model-theoretic semantics for the language and an axiomatization of the primitive object types such as sets, lists, relations, and functions. Sets are defined using a standard set theory. Lists are finite sequences of objects. There is no distinction among linked lists, arrays, or any other encoding of sequences, since KIF is a declarative specification language and not an implementation. Relations are defined as sets of tuples, where each tuple is a list of objects. Functions are a special case of relations, in which the last object in each tuple is unique given the preceding objects. A function of N arguments is a relation of N+1 arguments in which the value of the last argument is a function of the first N arguments.

Ontolingua definitions are Lisp-style forms that associate a symbol with an argument list, a documentation string, and a set of KIF sentences labeled by keywords. An Ontolingua ontology is made up of definitions of classes, relations, functions, distinguished objects, and axioms that relate these terms.

A relation is defined with a form like the following:

```
(define-relation CONNECTS (?comp1 ?comp2)
   "The most general binary connection relation between components.
   Connected components cannot be subparts of each other."
   :def (and (component ?comp1)
             (component ?comp2)
             (not (subpart-of ?comp1 ?comp2))
             (not (subpart-of ?comp2 ?comp1)))
```

The arguments `?comp1` and `?comp2` are universally quantified variables ranging over the items in the tuples of the relation. This example is a binary relation, so each tuple in the relation has two items. Relations of greater arity can also be defined. The sentence after the `:def` keyword is a KIF sentence stating logical constraints over the arguments. Constraints on the value of the first argument of a binary relation are effectively domain restrictions, and those on the second argument of a binary relation are range restrictions. There may also be complex expressions stating relationships among the arguments to a relation. The `:def` constraints are *necessary* conditions, which must hold if the relation holds over some arguments. It is also possible to state sufficient conditions, or any combination. In textbook notation for predicate calculus, the form above means:

$$\forall x,y \;\; \text{connects}(x,y) \Rightarrow \quad \text{component}(x) \wedge \text{component}(y) \wedge \neg \text{subpart-of}(x,y) \wedge \neg \text{subpart-of}(y,x).$$

If the `:iff-def` keyword is used instead of `:def`, the definition becomes necessary *and* sufficient. That is:

$$\forall x,y \;\; \text{connects}(x,y) \Leftrightarrow \quad \text{component}(x) \wedge \text{component}(y) \wedge \neg \text{subpart-of}(x,y) \wedge \neg \text{subpart-of}(y,x).$$

A class is defined with a similar form, where there is exactly one argument, called the instance variable. In Ontolingua, classes are treated as unary relations to help unify object- and relation-centered representation styles.

```
(define-class CONNECTION (?connection)
    "A connection is a special case of module where there are no internal variables. They can be thought of as
    conduits for transfer of energy/matter/information among modules. They also can be viewed as an
    encapsulation of intermodule constraints. Connections are typed by the number and type of their ports."
     :def (and (module ?connection)
               (not (exists ?variable
                      (internal-state-variables ?connection ?variable)))))
```

The `:def` sentence describes those things that must be true of instances of the class, that is, necessary constraints. Unary relations applied to the instance variable specify superclasses of the class being defined. For example, `connection` is a subclass of `module` by definition, because every connection `?connection` must also be a `module`. Other properties of instances can be stated using KIF sentences mentioning the instance variable. The logical meaning of the class definition above is

$$\forall x \;\; [\text{connection}(x) \Rightarrow \text{module}(x) \wedge \neg \exists y \;\; \text{internal-state-variables}(x,y) \;]$$

A function is defined like a relation, with a slight variation in syntax, moving the final argument outside of the argument list:

```
(define-function SQUARED (?n) :-> ?value
    "The squared of a number is the product of it times itself."
    :def (and (number ?n)
              (nonnegative-number ?value))
    :lambda-body (* ?n ?n))
```

As in definitions of relations, the arguments to a function are constrained with necessary conditions following the `:def` keyword. The function is only defined on arguments that satisfy these domain constraints; in this case, the condition `(number ?n)` means the function is only defined for numbers.[4] A restriction on the range of the function is specified with a sentence constraining the value variable after the `:->` keyword (i.e., `?value` is a nonnegative number). The body of the `define-function` form is the same as the `define-relation` form, except it also allows a `:lambda-body` to be given. A lambda-body is a KIF term expression that denotes the value of the function in terms of its arguments. In the example, `*` is a function and the lambda-body is a term that denotes the product of `?n` and itself. Since definitions are purely declarative, the functions in a lambda body are "side-effect free." The logical meaning of the example function definition is

[4]The constraint `(number ?n)` does not say that the function is *defined* for all numbers, only that when it is defined for some argument `?n`, that `?n` is a number.

$\forall x,y\ [y = \text{squared}(x)] \Rightarrow \text{number}(x) \wedge \text{nonnegative-number}(y)$
$\forall x\ \text{squared}(x) = x^2$

Functions need not be defined on all objects. When a function is applied to argument on which it is undefined, the term expression denotes a special KIF object called `bottom` (⊥). For example, if the function constant `*` is defined on all numbers, then the `squared` function could be described as:

$$\text{squared}(x) = \begin{cases} x^2 & \textit{if}\ \text{number}(x) \\ \bot & \textit{otherwise} \end{cases}$$

Finally, it is possible to define distinguished individuals in an ontology. Although Ontolingua is intended for defining vocabulary, rather than storing the ground facts of a knowledge base, sometimes the name of a particular constant is part of the vocabulary. For example:

```
(define-instance ZERO (number)
    "Zero is the identity element for addition."
    :axiom-def (forall ?x
                   (=> (number ?x)
                       (and (= (+ ?x ZERO) ?x) (= (+ ZERO ?x) ?x)))
    := 0)
```

This form defines the object constant `zero`. The list `(number)` says that `zero` is an instance of the class `number`. An object can be an instance of more than one class. The sentence labeled with `:axiom-def` is a stand alone axiom constraining the object constant. After the `:=` keyword is a KIF term expression that denotes the object being defined. It is like the `:lambda-body` expression for functions, but without free variables.

Details on the language and software are documented in a longer report (Gruber, 1992) and the documentation distributed with the software.

3.2 An example definition

Consider an ontology for bibliographic information.[5] The purpose of a bibliography ontology is to support knowledge sharing tasks such as exchanging bibliographic data among databases, integrating bibliographic databases with other databases (e.g., address books, company directories), linking citations across hyperdocument boundaries, and providing network-based services for bibliographic data processing. The domain of discourse of these tasks includes documents, authors, publishers, dates, places, and the references that occur in documents and card catalogs. In this ontology, authors are defined as follows.

[5]The examples here are only fragments of larger ontologies, which are available as examples with the Ontolingua software. Complete ontologies are coherent theories that constitute publications in themselves. See Appendix 1 for excerpts from an ontology of relevance to knowledge acquisition community.

```
(define-class AUTHOR (?author)
    "An author is a person who writes things.  An author must have created at least one document.
    In this ontology, an author is known by his or her real name."
    :def (and (person ?author)
              (= (value-cardinality ?author AUTHOR.NAME) 1)
              (value-type ?author AUTHOR.NAME biblio-name)
              (>= (value-cardinality ?author AUTHOR.DOCUMENTS) 1)
              (<=> (author.name ?author ?name)
                   (person.name ?author ?name))))
```

This form defines the term `author`, which denotes a class.[6] Individual authors such as Marvin Minsky are instances of the class. The `:def` sentence gives necessary conditions on class membership. The first conjunct of the sentence, `(person ?author)`, says that all authors must also be persons. The second conjunct says that authors must have exactly one associated name, given by the relation `author.name`. This means that the relation `author.name` maps every instance of the class `author` to some name, with only one name per author. The third conjunct specifies the type restriction that the value of the `author.name` relation applied to instances of `author` must be an instance of the class `biblio-name`. The fourth conjunct says that there must be some document(s) associated with every author by the relation `author.documents`. The fourth conjunct specifies that the `author.name` and the `person.name` of an author are the same. This constraint explicitly rules out calling authors by pen names.

3.3 The ontological commitments implied by definitions

What is the nature of the ontological commitments specified by such definitions? Agents that commit to the ontology agree to use the terminology in a manner that is consistent with the axioms and documentation strings. One can take advantage of such a commitment when using knowledge services. For instance, imagine that a program asks for the author of some document, perhaps with this query:

```
(ask ?x (author.documents ?x society-of-mind))
```

The database service might return the identifier `author-423455` as a binding for `?x`. Then the asking program could request the name of the author with this query:

```
(ask ?name (author.name author-423455 ?name))
```

From the definition of `author`, the asking program can now count on the name returned by the `author.name` query to be the same as the name of the *person* Marvin Minsky. It could then use that name when querying other knowledge bases about famous people, only some of whom are authors.

An agent that commits to the bibliography ontology does not necessarily have to *store* a database of authors, or to be able to answer any query about authors. The ontological commitments are still useful under partial knowledge. Consider the case where there are several bibliography knowledge servers on the network that are committed to the bibliography ontology, each with incomplete knowledge. Imagine we ask one knowledge server for the books

---

[6]The relations `author.name` and `author.documents` can be viewed as slots on instances of the class `author`. The relations `value-cardinality` and `value-type`, defined in the Frame Ontology, are used to state slot constraints (see Section 3.6 and Appendix 2).

by Marvin Minsky, and it replies that it isn't aware of any. That does not mean that there are no such books, of course. We can assume that there must be some works written by Minsky, since he was called an author. We can then query other knowledge servers, until we find one that knows the answer. Similarly, if we asked for the name of an author in a distributed fashion and got an answer, we know from the definition that there is at most one name per author, so we can stop asking other knowledge servers for his name.

3.4 Translation into representation systems

The definition of the author class specifies constraints that are typical of descriptions in many systems. Let us look at how it translates into different representation languages.

In Epikit, a predicate calculus language, the author definition translates to

```
(=> (author $author)
    (person $author))
(=> (author $author)
    (exists $y (author.name $author $y) (biblio-name $y)))
(=> (author $author)
    (=> (author.name $author $y) (author.name $author $y2)
        (= $y $y2)))
(=> (author $author)
    (exists $y (author.documents $author $y)))
(=> (author $author)
    (<=> (author.name $author $name)
         (person.name $author $name)))
```

Epikit is based on an early version of KIF; it is a prefix-form predicate calculus with equality and functional term expressions. Epikit does not distinguish definitions from any other axioms. Thus, the necessary conditions of the Ontolingua definition get turned into implications. Epikit also does not support reasoning about relations as objects. In the Ontolingua definition some of the relations take classes and other relations as arguments. For example, the relation `value-type` took the binary-relation `author.name` and the class `biblio-name` as arguments. Relations that take other relations as arguments are called *second-order relations* (we will say more about them in Section 3.6). To translate such statements into pure first-order systems such as Epikit, Ontolingua recognizes the second-order relations and instantiates axiom schemata for them. For example,

```
(>= (value-cardinality ?author author.documents) 1)
```

becomes

```
(exists $y (author.documents $author $y)).
```

In Loom, a object-oriented KL-ONE-style language, the author definition translates as

```
(define-concept author
  :is (:and :primitive
            person
            (:the author.name biblio-name)
            (:at-least 1 author.documents)
            (:same-as author.name person.name)))
```

The Loom translation looks similar to the Ontolingua form, except that there are no free variables. The second-order relations such as `value-type` and `value-cardinality` were

fashioned after the analogous operators in Loom derivatives such as `:all` and `:at-least`. They capture common constraints that hold between classes and binary relations applied to their instances. However, in Ontolingua, these second-order relations are not part of the syntax or primitive operators of the language. They are just vocabulary—relations defined in KIF—for specifying portable ontologies and translating among Loom-like systems.

In KEE, an object-oriented, frame-based system, the example would appear like this in a frame browser:

```
Unit: AUTHOR
Comment: "An author is a person who writes things.
   An author must have created at least one document.
   In this ontology, an author is known by his or her real name."
Superclasses: PERSON
Member Of: CLASSES
Member slot: AUTHOR.NAME
  ValueClass: (MEMBER.OF BIBLIO-NAME)
  Min.Cardinality: 1
  Max.Cardinality: 1
Member slot: AUTHOR.DOCUMENTS
  Min.Cardinality: 1
Member slot: PERSON.NAME  from Person
```

KEE is a distant cousin of Loom, and it also has special syntax for slot constraints (e.g., Valueclass, Min.Cardinality). However, the KEE frame system is less expressive than Loom and the logic-based languages. Consequently, a constraint that would be said declaratively in Loom might be implemented with an attached procedure in KEE. For instance, the KEE frame browser shows a template for the slot `person.name` "inherited" from the superclass `person`, but it does not have a constraint language capable of expressing the fact that the value of the `author.name` slot is the same as the value of the `person.name` slot. To implement this constraint, one would write an attached procedure that computes the value of one of the slots from the other if either is given a value. The existence of embedded procedural code in object-oriented systems such as KEE is one reason for maintaining ontologies in a more expressive, declarative language, and translating *into* the restricted languages. The assumption is that it is easier to generate procedural code to implement a declarative specification than to infer the declarative semantics from a procedure.

3.5 Translation architecture

The design of Ontolingua was constrained by three competing requirements:

- to offer an expressive, declarative, system-independent language in which to write definitions
- to support translation into specialized representations with restricted expressive and inferential power
- to allow for easy extension in expressiveness and target implementations

It is impossible achieve all three of the requirements completely. If the source language is more expressive than the target languages, then the translation will be incomplete. If one built separate, special-purpose translators for each language, the subset of KIF that is handled

would depend on the target language. And each additional translator would be as expensive to implement as the first.

Ontolingua's design strikes a balance in which translation is complete only for a restricted subset of representational idioms, but these are defined independently of any target system. For each of these idioms, Ontolingua code *recognizes* its expression in KIF and transforms it into a *canonical form*. The canonical form is an intermediate representation that facilitates translation into multiple target languages. Sentences in the canonical form have a predictable format that simplifies pattern matching by reducing the number of ways that an equivalent thing can be stated.

For example, in the `author` example, the necessary condition

```
:def (... (person ?author) ...)
```

is an idiom that is transformed by Ontolingua into the stand alone axiom

```
(subclass-of author person)
```

Similarly, the slot equivalence constraint

```
(<=> (author.name ?author ?name)
     (person.name ?author ?name))
```

is turned into

```
(same-slot-values author author.name person.name).
```

In both of these cases, Ontolingua recognized a representational idiom, and transformed it into a logically equivalent sentence using a *second-order* relation. A second-order relation is a relation that can take other relations (including classes and functions) as arguments. For example, the statement `(range` *relation class*`)` specifies that all values of the given binary *relation* must be an instance of *class*. `Range` is a second-order relation that is recognized by Ontolingua; it is defined in the Frame Ontology. This constraint could be expressed in several equivalent ways in KIF:

```
(=> (relation ?x ?y) (class ?y))
(forall (?x ?y) (=> (relation ?x ?y) (class ?y)))
(<= (class ?y) (relation ?x ?y))
(=> (member ?tuple relation) (class (second ?tuple)))
...
```

Recognizing idioms and transforming them into canonical form are two of the front-end processes that Ontolingua performs when translating. The canonical form for Ontolingua is to put constraints into ground (variable-free) axioms using the Frame Ontology vocabulary. Back-end translators are written to anticipate these forms and transform them into output appropriate to specific target systems. For example, translators look for patterns of the form (`range` *symbol*) instead of all of the variants listed above. Since the front-end Ontolingua processor can recognize and transform the common variants, the back-end translators have a finite set of cases to consider.

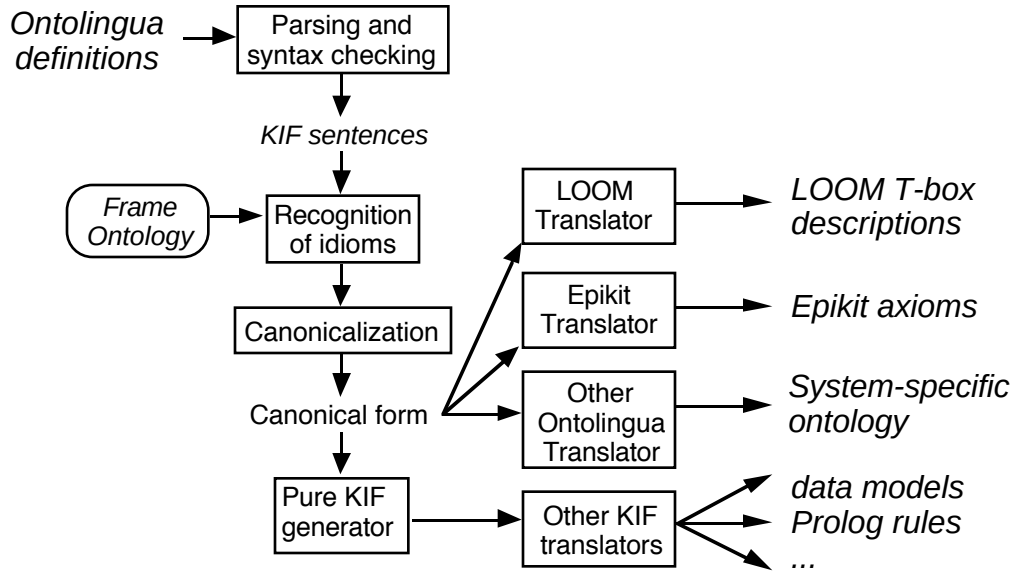

Figure 2: Translation architecture for Ontolingua. Instances of common representation idioms are recognized and transformed into a simpler, equivalent form using the second-order vocabulary from the Frame Ontology. These and other transformations result in a canonical form, which is given to back-end translators that generate system-specific output. A pure KIF output is also available at this stage, to be given to other translators developed for KIF, such as KIF-to-Prolog.

Figure 2 shows the data flow of the translation architecture. This architecture is analogous to a conventional programming language compiler, which parses source into an intermediate form that is then given to specialized code generation modules.

3.6 The Frame Ontology

The choice of supported second-order relations was made to capture common knowledge-organization conventions used in object-centered or frame-based representations. The second-order relations recognized by Ontolingua are defined in an ontology, called the *Frame Ontology*. It contains a complete axiomatization of classes and instances, slots and slot constraints, class and relation specialization, relation inverses, relation composition, and class partitions. Each second-order term is defined in natural language and KIF axioms. A current list of the Frame Ontology vocabulary is given in Appendix 2. By no coincidence, these are the constructs that are supported with special-purpose syntax and code in object-centered representation systems. Together, they constitute a purely declarative representational framework for describing hierarchies of classes with slots. Users of Ontolingua and developers of translators can consequently share a well-defined vocabulary and make assumptions about its consistent use.

The Frame Ontology is carefully designed to allow both relational and object-centered styles of representation to co-exist (and it is consistent with the built-in ontologies in KIF for relations and functions). The design choices are not arbitrary. There are several possible ways to axiomatize classes, for example. Some systems treat them as simple sets; others as unary relations; others as intentional entities or descriptions. The Frame Ontology defines classes to

be coextensional with unary relations, so that the relational-style `(person Fred)` is compatible with the object-style `(instance-of Fred person)`. Similarly, slots can be viewed as binary relations, unary functions onto sets, projections of relations, or some combination. The Frame Ontology makes a minimal commitment that allows slots to be viewed as binary relations or unary functions, and does not commit to distinctions among attributes, properties, and slots as local to classes (see (Guarino, 1992; Sowa, 1992)).

We have already described how the second-order vocabulary can facilitate translation by providing a canonical form for representation idioms. The Frame Ontology vocabulary can also be used by ontology writers directly in their definitions, as a succinct representation for simple things that can be awkward in pure first-order predicate calculus. For example, in the `author` definition the relation `value-cardinality` was used as follows

```
(= (value-cardinality ?author author.name) 1)
```

which could have been stated using this form:

```
(and (exists ?name (author.name ?author ?name))
     (forall (?name1 ?name2)
         (=> (and (author.name ?author ?name1)
                  (author.name ?author ?name2))
             (= ?name1 ?name2))))
```

The same second-order relation is also used in this constraint:

```
(>= (value-cardinality ?author author.documents) 1)
```

which might otherwise have been stated using this idiom:

```
(exists ?document (author.documents ?author ?document)).
```

Note that second-order relations are not substitutional macros for the idioms. Obviously the idioms for these two applications of value cardinality are not expanded from the same template. Compare them to the definition of `value-cardinality` as given in the Frame Ontology, which uses a set-theoretic axiomatization instead of existential quantification.

```
(define-function VALUE-CARDINALITY (?instance ?binary-relation) :-> ?n
  "The VALUE-CARDINALITY of a binary-relation with respect to a given domain instance is the number of
range-elements to which the relation maps the domain-element. For a function (single-valued relation), the
VALUE-CARDINALITY is 1 on all domain instances for which the function is defined. It is 0 for those instances
outside the exact domain of the function."
  :lambda-body (cardinality (setofall ?y
                                  (holds ?binary-relation ?instance ?y)))
```

Using the second-order relations of the Frame Ontology, one can also describe properties of classes, functions, and other relations, without mentioning the instance arguments or instance variables. For example, the definition of the class author could have been defined using only the second order vocabulary:

```
(define-class AUTHOR (?author)
    "An author is a person who writes things.  An author must have created at least one document.
    In this ontology, an author is known by his or her real name."
    :axiom-def (and (subclass-of author person)
                    (slot-value-cardinality author author.name 1)
                    (slot-value-type author author.name biblio-name)
                    (minimum-slot-cardinality author author.documents 1)
                    (same-slot-values author author.name person.name)))
```

The `:axiom-def` label indicates that this is a stand alone axiom with no free variables. Sentences in the `:axiom-def` mention the constant `author` rather than the instance variable `?author`. The definition above is the canonical form passed on to Ontolingua translators. The relations with "slot" in their names are slot constraints, which appear as facets or special syntax some frame systems.

Independent of Ontolingua, the Frame Ontology can serve as a framework for defining representational clichés and idioms. The same conventions facilitate translation *into* KIF from object-centered systems, offering a canonical form for stating things that might be written in several equivalent forms. With feedback from Ontolingua and the knowledge representation community, we expect to add new second-order relations to accommodate the expressive needs for defining common ontologies. Fortunately, the cost of adding a new second-order relation is limited to the effort required to support that "special case" in each target representation. Importantly, the syntax, operators, and semantics of the shared language need not change.

4. Discussion

Ontolingua is not a replacement for a representation system like Algernon or Loom. One of the motivations for trying to make ontologies portable over several implementations of representation systems, rather than just standardizing on one system, is that different systems provide different computational services at different costs. The translation strategy allows one to use some computational services at conceptual design time (e.g., terminological classification, general-purpose inference) and to use the same ontology for the development of production systems (e.g., translating it into object-oriented class definitions or database schemata). It is important to note that Ontolingua does not support query processing; users call implemented systems directly to invoke inference procedures. This lessens the need to write ontologies in a special-purpose language, because many of the compromises made in the expressiveness of a representation are made to optimize run-time inference.

Because it translates into restricted languages, Ontolingua is inherently incomplete with respect to the KIF language. The only way to be truly portable over the specialized systems is to take a common-denominator approach, writing definitions using only those constructs known to be supported in all of the systems of interest. The Frame Ontology is a compromise: it identifies a set of *some* common idioms that are supported in *most* of the target systems. Although it can parse any legal KIF expression, Ontolingua is only designed to translate a subset of KIF (which is described in the documentation). For instance, the current implementation is not guaranteed to translate sentences using some of the more sophisticated language features of KIF, such as meta-level operators. It currently does not support user-defined second-order relations; only the terms defined in the Frame Ontology are recognized.

However, the full, first-order predicate calculus is available to the ontology writer. Consequently, incompleteness is a fact of life in the representation translation business. When Ontolingua cannot translate a sentence into a target implementation, it issues an informative message. The practical consequence of not translating a sentence could be that the target system may be unable to enforce a constraint, or it may have to fall back on inefficient theorem proving. The good news is that target systems can be customized or replaced without changing the ontology.

Ontolingua is a domain-independent translation tool; it does not help with the intellectual task of designing ontologies. Deciding which concepts and relations to include, and writing axiomatic definitions, requires knowledge that is not found in existing knowledge bases. Ontologies like the Frame Ontology and the axiomatization of set theory in KIF are called *representation ontologies*. Representation ontologies provide a framework, but do not offer guidance about how to represent the world. *Content ontologies* make claims about how the world (or a conceptualization of it) should be described. Some content ontologies are intended to be comprehensive—to be "conceptual coat rack" on which to "hang" more specific ontologies and domain knowledge. Examples of comprehensive content ontologies include Cyc's global ontology (Lenat & Guha, 1990; Lenat, Guha, Pittman, Pratt, & Shepherd, 1990), the Penman Upper Model (Bateman, Kasper, Moore, & Whitney, 1990), and the Lilog KB (Pirlein, 1993). Whether these ontologies can help in the design of more specialized ontologies is an empirical question. Ontolingua was created to support experimentation by making such ontologies accessible as off-the-shelf artifacts.

As a representation ontology, the Frame Ontology *does* specify a conceptualization implicit in knowledge bases written in the object-centered style. Classes, specialization, and slot constraints are not built into KIF; they are a convention supported in specialized representation-system architectures. The Frame Ontology defines these object-centered concepts in a declarative form that is stylistically compatible with pure relation-centered usage. The vocabulary for these concepts defines what can be translated, and offers a succinct form for users to write portable ontologies. In these ways, the Frame Ontology is a specification of the ontological commitments of Ontolingua.

Acknowledgments

This work was motivated by collaborative activities of the DARPA Knowledge Sharing Effort. Thanks for insights and advice from Richard Fikes, Mike Genesereth, Bill Mark, Bob MacGregor, Bob Neches, Ramish Patil, and Marty Tenenbaum. Karl "Fritz" Mueller redesigned and implemented the current version of the software. Special thanks to Richard Fikes, Hania Gajewska, and Charles Petrie for thoughtful reviews. Funding for this work is provided by DARPA prime contract DAAA15-91-C-0104 through Lockheed subcontract SQ70A3030R, and NASA Grants NCC 2-537 and NAG 2-581 (under ARPA Order 6822).

References

Allen, J., & Lehrer, N. (1992). *DARPA/Rome Laboratory Planning and Scheduling Initiative Knowledge Representation Specification Language (KRSL), Version 2.0.1 Reference Manual*. ISX Corporation.

Bateman, J. A., Kasper, R. T., Moore, J. D., & Whitney, R. A. (1990). *A General Organization of Knowledge for Natural Language Processing: The Penman Upper Model*. Technical report, USC/Information Sciences Institute, Marina del Rey, CA.

Bradshaw, J. M., Holm, P. D., & Boose, J. H. (1992). Sharable ontologies as a basis for communication and collaboration in conceptual modeling. *Proceedings of the 7th Knowledge Acquisition for Knowledge-based Systems Workshop*, Banff, Canada.

Crawford, J. M., & Kuipers, B. J. (1989). Toward a theory of access-limited logic for knowledge representation. *Proceedings of the First International Conference on Principles of Knowledge Representation*, Morgan Kaufmann.

Cutkosky, M., Engelmore, R. S., Fikes, R. E., Gruber, T. R., Genesereth, M. R., Mark, W. S., Tenenbaum, J. M., & Weber, J. C. (1993). PACT: An experiment in integrating concurrent engineering systems. *IEEE Computer*, 26(1), 28-37.

Fikes, R., Cutkosky, M., Gruber, T., & van Baalen, J. (1991). *Knowledge Sharing Technology Project Overview*. Technical Report KSL 91-71, Stanford University, Knowledge Systems Laboratory.

Fikes, R., & Kehler, T. (1985). The role of frame-based representation in reasoning. *Communications of the ACM*, 28(9), 904-920.

Finin, T., Weber, J., Wiederhold, G., Genesereth, M., Fritzson, R., McKay, D., McGuire, J., Pelavin, P., Shapiro, S., & Beck, C. (1992). *Specification of the KQML Agent-Communication Language*. Technical Report EIT TR 92-04, Enterprise Integration Technologies, Palo Alto, CA.

Fulton, J. A. (1992). *Technical report on the semantic unification meta-model*. Standards working document ISO TC184/SC4/WG3 N103, IGES/PDES Organization, Dictionary/Methodology Committee. Contact James Fulton, Boeing Computer Services, P. O. Box 24346, MS 7L-64, Seattle, WA 98124-0346.

Genesereth, M. R. (1990). *The Epikit Manual*. Epistemics, Inc. Palo Alto, CA.

Genesereth, M. R. (1992). An Agent-Based Framework for Software Interoperability. *Proceedings of the DARPA Software Technology Conference*, Meridian Corporation, Arlington VA, pages 359-366. Also Report Logic-92-2, Computer Science Department, Stanford University, June 1992.

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

Genesereth, M. R., & Nilsson, N. J. (1987). *Logical Foundations of Artificial Intelligence*. San Mateo, CA: Morgan Kaufmann Publishers.

Gruber, T. R. (1991). The Role of Common Ontology in Achieving Sharable, Reusable Knowledge Bases. In J. A. Allen, R. Fikes, & E. Sandewall (Eds.), *Principles of Knowledge Representation and Reasoning: Proceedings of the Second International Conference*, Cambridge, MA, pages 601-602, Morgan Kaufmann.

Gruber, T. R. (1992). *Ontolingua: A mechanism to support portable ontologies*. Technical Report KSL 91-66, Stanford University, Knowledge Systems Laboratory. Revision.

Gruber, T. R., Tenenbaum, J. M., & Weber, J. C. (1992). Toward a knowledge medium for collaborative product development. In J. S. Gero (Eds.), *Artificial Intelligence in Design '92*. Boston: Kluwer Academic Publishers.

Guarino, N. (1992). Concepts, Attributes, and Arbitrary Relations: Some Linguistic and Ontological Criteria for Structuring Knowledge Bases. *Data and Knowledge Engineering*, 8.

Lenat, D. B., & Guha, R. V. (1990). *Building Large Knowledge-based Systems: Representation and Inference in the Cyc Project*. Menlo Park, CA: Addison-Wesley.

Lenat, D. B., Guha, R. V., Pittman, K., Pratt, D., & Shepherd, M. (1990). Cyc: Toward Programs with Common Sense. *Communications of the ACM*, 33(8), 30-49.

Levesque, H. J. (1984). Foundations of a functional approach to knowledge representation. *Artificial Intelligence*, 23, 155-212.

MacGregor, R. (1991). The Evolving Technology of Classification-based Knowledge Representation Systems. In J. Sowa (Eds.), *Principles of Semantic Networks: Explorations in the Representation of Knowledge*. San Mateo, CA: Morgan Kaufmann.

Marcus, S., Stout, J., & McDermott, J. (1988). VT: An expert elevator designer that uses knowledge-based backtracking. *AI Magazine*, 9(1), 95-111.

McCarthy, J., & Hayes, P. J. (1969). Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer & D. Michie (Eds.), *Machine Intelligence 4*. Edinburgh: Edinburgh University Press.

Morik, K., Causse, K., & Boswell, R. (1991). *A Common Knowledge Representation Integrating Learning Tools*. Technical Report, GMD.

Neches, R., Fikes, R. E., Finin, T., Gruber, T. R., Patil, R., Senator, T., & Swartout, W. R. (1991). Enabling technology for knowledge sharing. *AI Magazine*, 12(3), 16-36.

Newell, A. (1982). The knowledge level. *Artificial Intelligence*, 18(1), 87-127.

Patil, R. S., Fikes, R. E., Patel-Schneider, P. F., McKay, D., Finin, T., Gruber, T. R., & Neches, R. (1992). The DARPA Knowledge Sharing Effort: Progress report. In C. Rich, B. Nebel, & W. Swartout (Eds.), *Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference*, Cambridge, MA, Morgan Kaufmann.

Pirlein, T. (1993). Reusing a large domain-independent knowledge base. *Proceedings of the Fifth International Conference on Software Engineering and Knowledge Engineering*, San Francisco.

Sowa, J. F. (Eds.). (1992). *Principles of Semantic Networks*. San Mateo, CA: Morgan Kaufmann Publishers.

Spiby, P. (1991). *EXPRESS Language Reference Manual*. International Standard ISO 10303-11, Project Three ("Express Language") Working Group Five ("STEP Development Methods") of Subcommittee Four ("Manufacturing Languages and Data") of ISO Technical Committee 184 ("Industrial Automation Systems and Integration"); International Standards Organization.

Walther, E., Eriksson, H., & Musen, M. A. (1992). *Plug and play: Construction of task-specific expert-system shells using sharable context ontologies*. Technical Report KSl-92-40, Knowledge Systems Laboratory, Stanford University.

Yost, G. R. (1992). *Configuring elevator systems*. Unpublished manuscript, AI Research Group, Digital Equipment Corporation, 111 Locke Drive, Marlboro, MA 02172.

Appendix 1: An example ontology

Here is an excerpt from an ontology about configuration design. The ontology defines vocabulary that can be used to formally specify a design task, where the task specification includes knowledge about an engineering domain as well as customer requirements and other constraints. The ontology was written to support a collaborative experiment among knowledge acquisition researchers in the domain of elevator design (the domain of the VT system (Marcus, Stout, & McDermott, 1988)). Each participant in the experiment will build a system capable of solving configuration design tasks based on documentation of the domain knowledge, and will compare the resulting products. This ontology will support the formal encoding of the domain knowledge and problem description that is documented in over 20 pages of prose and data (Yost, 1992).

```
(define-theory CONFIGURATION-DESIGN
   (frame-ontology physical-quantities scalar-quantities)
   "This is an ontology for describing configuration design tasks: describing components, parameters, constraints,
   and configurations. Like parametric design tasks, a configuration design task is a search for values of
   parameters that satisfy a set of constraints. In configuration design, however, the set of relevant parameters
   and constraints is a function of the components chosen. The specification of a particular design task, such as
   the VT task for designing an elevator, is the description of a system component like an elevator, a set of
   constraints on that component (e.g., customer requirements), and a library of background constraints (e.g.,
   laws of physics) and available components. A valid design is a complete description of the system component,
   with values for relevant parameters and choices for subcomponents, that satisfies all the constraints. The
   definitions in this ontology embody this tight relationship between components, constraints, and parameters.
   Constraints are defined as unary predicates that hold for components. Parameters are unary functions on
   components. Components are related to other components through binary subpart relations.")

(in-theory 'configuration-design)

(define-class COMPONENT (?x)
   "A component is a primitive module or assembly of primitive modules that participate in a design. Components
   need not correspond to physically whole objects such as standard parts from a parts catalog. Components may
   also represent functional and behavioral abstractions. This ontology only says that components are the locus of
   attributes and constraints. To say that a component C has-attribute A means that there is a function A from C
   to the value of the attribute. In object-oriented terminology, one can think of A as a slot of C, and calling it an
   attribute means that it may be mentioned in constraints on C. Similarly, a component C has-subpart S means
   the function S maps C to another component which plays the S role in C. Subpart slots may identify structural,
   functional, or similar kinds of relationships among components."
   :def (and (value-type ?x HAS-ATTRIBUTE attribute-slot)
             (value-type ?x HAS-SUBPART subpart-slot)
             (value-type ?x HAS-CONSTRAINT constraint)
             (value-type ?x SATISFIES-CONSTRAINT constraint)
             (value-cardinality ?x COMPONENT.COST 1)
             (value-type ?x COMPONENT.COST cost-quantity)))

(define-relation HAS-ATTRIBUTE (?component ?attribute-slot)
   "A component has an attribute if the attribute value is given by a unary function, called an attribute-
   slot, that is defined for that component. Calling a slot an attribute means that it is a design parameter: it will
   need to be assigned a value and may be mentioned in constraints."
   :def (and (component ?component)
             (attribute-slot ?attribute-slot)
             (value-cardinality ?component ?attribute-slot 1)))
```

```
(define-class ATTRIBUTE-SLOT (?unary-function)
```
"An attribute slot is a unary function from `component`*s to* `attribute-value`*s, which are either scalar quantities or strings."*
```
   :def (and (unary-function ?unary-function)
             (domain ?unary-function component)
             (range ?unary-function attribute-value)))
```

```
(define-relation HAS-SUBPART (?component ?subpart-slot)
```
"A component has a subpart if the subpart is given by a unary function, called a `subpart-slot`*, that is defined for that component."*
```
   :def (and (component ?component)
             (subpart-slot ?subpart-slot)
             (value-cardinality ?component ?subpart-slot 1)))
```

```
(define-class SUBPART-SLOT (?unary-function)
```
"A subpart slot is a unary function from components to other components. It is antisymmetric and antireflexive."
```
   :def (and (unary-function ?unary-function)
             (domain ?unary-function component)
             (range ?unary-function component)
             (antisymmetric-relation ?unary-function))
             (antireflexive-relation ?unary-function)))
```

```
(define-class VALID-COMPONENT (?component)
```
"A component is 'configured' or fully specified if all of its constraints are satisfied and all of its subparts are also configured. By definition, there exist values for all the attributes of a component. Whether an agent can tell you what the values of attributes are is not part of the definition of configured component. Knowing all the constraints associated with a component will require making a closed world assumption on the has-constraint slot."

```
   :iff-def (and (component ?component)
                 (=> (has-constraint ?component ?constraint)
                     (satisfies-constraint ?component ?constraint))
                 (=> (has-subpart ?component ?part-slot)
                     (valid-component (value ?part-slot ?component)))))
```

```
(define-relation OPTIMAL-COMPONENT (?comp ?component-class)
```
"An optimal-component is the least costly instance of a component class. To evaluate this relation will require making some kind of closed-world assumption over possible components."

```
   :iff-def (and (component ?comp)
                 (component-class ?component-class)
                 (instance-of ?comp ?component-class)
                 (=> (instance-of ?other-component ?component-class)
                     (=< (component.cost ?comp)
                         (component.cost ?other-component)))))
```

Appendix 2: Vocabulary defined in the Frame Ontology

This appendix lists the vocabulary of the Frame Ontology (Version 4). The complete axiomatization and documentation is included with the Ontolingua software.

class **relation** (?relation)
class **function** (?function)
class **class** (?class)
relation **instance-of** (?individual ?class)
function **all-instances** (?class) :-> ?set-of-instances
function **one-of** (@instances) :-> ?class
relation **subclass-of** (?child-class ?parent-class)
relation **superclass-of** (?parent-class ?child-class)
relation **subrelation-of** (?child-relation ?parent-relation)
relation **direct-instance-of** (?individual ?class)
relation **direct-subclass-of** (?child-class ?parent-class)
function **arity** (?relation) :-> ?n
function **exact-domain** (?relation) :-> ?domain-relation
function **exact-range** (?relation) :-> ?class
relation **total-on** (?relation ?domain-relation)
relation **onto** (?relation ?range-class)
class **n-ary-relation** (?relation)
class **unary-relation** (?relation)
class **binary-relation** (?relation)
class **unary-function** (?function)
relation **single-valued** (?binary-relation)
function **inverse** (?binary-relation) :-> ?relation
function **projection** (?relation ?column) :-> ?class
function **composition** (?relation-1 ?relation-2) :-> ?binary-relation
relation **composition-of** (?binary-relation ?list-of-relations)
function **compose** (@binary-relations) :-> ?binary-relation
relation **alias** (?relation-1 ?relation-2)
relation **domain** (?relation ?class)
relation **domain-of** (?domain-class ?binary-relation)
relation **range** (?relation ?class)
relation **range-of** (?class ?relation)
relation **nth-domain** (?relation ?integer ?domain-class)
relation **has-value** (?domain-instance ?binary-relation ?value)
function **all-values** (?domain-instance ?binary-relation) :-> ?set-of-values
relation **value-type** (?domain-instance ?binary-relation ?class)
function **value-cardinality** (?domain-instance ?binary-relation) :-> ?n
relation **same-values** (?domain-instance ?relation-1 ?relation-2)
relation **inherited-slot-value** (?domain-class ?binary-relation ?value)
function **all-inherited-slot-values** (?domain-class ?binary-relation) :-> ?set-of-values
relation **slot-value-type** (?domain-class ?binary-relation ?range-class)
function **slot-cardinality** (?domain-class ?binary-relation) :-> ?n
relation **minimum-slot-cardinality** (?domain-class ?binary-relation ?n)
relation **maximum-slot-cardinality** (?domain-class ?binary-relation ?n)
relation **single-valued-slot** (?domain-class ?binary-relation)
relation **same-slot-values** (?domain-class ?relation-1 ?relation-2)
class **class-partition** (?set-of-classes)
relation **subclass-partition** (?c ?class-partition)
relation **exhaustive-subclass-partition** (?c ?class-partition)
relation **asymmetric-relation** (?binary-relation)
relation **antisymmetric-relation** (?binary-relation)
relation **antireflexive-relation** (?binary-relation)
relation **irreflexive-relation** (?binary-relation)
relation **reflexive-relation** (?binary-relation)
relation **symmetric-relation** (?binary-relation)
relation **transitive-relation** (?binary-relation)
relation **weak-transitive-relation** (?binary-relation)
relation **one-to-one-relation** (?binary-relation)
relation **many-to-one-relation** (?binary-relation)
relation **one-to-many-relation** (?binary-relation)

relation **many-to-many-relation** (?binary-relation)
relation **equivalence-relation** (?binary-relation)
relation **partial-order-relation** (?binary-relation)
relation **total-order-relation** (?binary-relation)
relation **documentation** (?object ?string)