In: Proc Int Conf on Formal Ontology and Information Systems - FOIS'98 (1998)
http://www.cs.utexas.edu/users/pclark/papers

Ontology Reuse and Application

Mike Uschold, Mike Healy, Keith Williamson, Peter Clark, Steven Woods
Boeing Applied Research & Technology, PO Box 3707, Seattle, USA

Abstract.
In this paper, we describe an investigation into the reuse and application of an existing ontology for the purpose of specifying and formally developing software for aircraft design. Our goals were to clearly identify the processes involved in the task, and assess the cost-effectiveness of reuse. Our conclusions are that (re)using an ontology is far from an automated process, and instead requires significant effort from the knowledge engineer. We describe and illustrate some intrinsic properties of the ontology translation problem and argue that fully automatic translators are unlikely to be forthcoming in the foreseeable future. Despite the effort involved, our subjective conclusions are that in this case knowledge reuse was cost-effective, and that it would have taken significantly longer to design the knowledge content of this ontology from scratch in our application. Our preliminary results are promising for achieving larger-scale knowledge reuse in the future.

Keywords: ontology reuse, ontology application, ontology translation.

1 Introduction

If there is to be a future for the construction of large-scale, knowledge-based systems, then it is essential that we are able to share and reuse representational components built by others. However, despite the potential advantages of such sharing, and the availability of such components in component libraries (e.g. [5]), it remains a challenging task to import and use such components.

Despite the existence of many ontologies, evident from the literature, there are few published examples of such reuse (e.g. [2, 4, 9]). Furthermore, in cases where an ontology is *reused*, (e.g. as the basis for building a new ontology rather than starting from scratch) descriptions of how the ontologies are *applied* are terse or absent[1].

In this paper, we describe the start-to-finish process of reusing and applying an existing ontology. We conducted an experiment consisting of the following steps: a) take the engineering math ontology [8] written in Ontolingua [7] from the library of ontologies at the Stanford Knowledge Systems Laboratory (KSL) Ontology Server, b) translate it into a target specification language c) integrate it into the *specification* of an existing small engineering software component d) transform and refine the enhanced specification into executable code using Specware, a system for the specification and formal development of software and e) demonstrate the ability to add units-conversion capabilities and dimensional consistency-checking to the engineering software.

[1]This common usage of the term 'reuse' is ambiguous, since it could be that both the original and the new ontology have yet to serve any specific purpose.

Issues and Objectives

One goal of this exercise is to identify the issues that affect successful reuse and application of existing ontologies. This should result in a clearer understanding of how to identify a reuse situation that is likely to succeed, as well as what kind of technical issues and problems will need to be faced. In this paper, we address the following questions. What is the overall process of reusing and applying ontologies? What specific difficulties arise in the process of ontology translation? Was the reuse cost-effective? Is Specware a good tool for ontology capture, reuse and application?

Our conclusions are that (re)using an ontology is far from an automated process, and instead requires significant effort from the knowledge engineer. The process of applying an ontology requires converting the knowledge-level [11] *specification* which the ontology provides into an *implementation*. This is time-consuming, and requires careful consideration of a) the context, intended usage, and idioms of both the source ontology representation language, and the target implementation language, and b) the specific task of the current application.

Despite this, our subjective conclusions from this experiment are that overall, knowledge reuse *is* cost-effective, and that it would have taken significantly longer to design the knowledge content of this ontology from scratch in our application, or from just reading the technical papers describing it. We are therefore cautiously optimistic about the longer term goal of achieving large-scale knowledge reuse.

Outline We begin with a brief of summary of the background context and motivation for our experiment. We examine each step in the process of reusing and applying the engineering math ontology. We highlight important issues and indicate our progress to date. We conclude by summarizing what we have learned from this exercise.

2 Context

Here we introduce the application problem that we are addressing, the target platform for ontology reuse and application (Specware), and the language that the original ontology is written in (Ontolingua).

Panel Layout Application The task performed by the engineering software component whose functionality we are trying to enhance with the engineering math ontology, is a simplified version of the layout design of a stiffened panel. The specific task is to determine the placement of "lightening holes" (for saving weight) on the panel, subject to cost and weight constraints as follows: GIVEN: a length of panel, constraints on hole placement (e.g. various minimum separation distances), and a cost function, DETERMINE: how many holes of what size should be drilled, optimizing the cost function for the panel.

We started with a fragment of production code which had been reverse engineered and re-implemented in Slang, the language of Specware (see below). This first entailed teasing out and declaratively representing much engineering knowledge and hidden assumptions that were implicit in the original code.

Finally, the specification was refined by selecting and specifying data structures and algorithms; from this, Specware synthesized executable code in Lisp.

Our model of panel layout is a research prototype intended to, among other things, demonstrate the feasibility of deriving knowledge based engineering software from reusable

components of formalized engineering knowledge.

Specware Our platform for demonstrating ontology reuse and application is Specware, [12] a system for the specification and formal development of software. It has a rigorous mathematical foundation, based on logic and category theory. It provides an order-sorted higher order logic representation language called Slang, whose semantics are founded on categorical type theory [3, 10]. It includes a rich set of primitives for composing specifications by reusing and parameterizing one or more copies of other specifications. The user specifies what and how various component specifications are to be included in the whole, and the colimit operation from category theory is used to compute the whole (see figure 2).

Specware was created for the purpose of developing software (neither excluding nor emphasizing knowledge-based systems). It supports the process of refining specifications into provably correct executable code.

Although Slang was not designed to represent knowledge-level ontologies it is entirely adequate for that purpose. The higher level *specifications* in Slang correspond to *ontologies* in Ontolingua (see below), in that both are intended to implementation-neutral. Lower level specifications in Slang must contain implementation information in order for Specware to generate code from them.

Ontolingua Ontolingua, was developed specifically for the purpose of knowledge sharing and reuse in the context of knowledge-based systems, *not* for the development of software in general. Ontolingua, based on KIF [6], was to facilitate reuse and interoperability by acting as an interchange format so that knowledge bases could be translated into and out of it.

However, there is no inference engine for Ontolingua, thus, unlike Slang, the only way an ontology written in Ontolingua can be used (by machine), is by translating it into some implemented language. Translating arbitrary sets of axioms in logic to any given output language, is not feasible. For this reason, Ontolingua is biased towards an object-oriented representation style [7].

3 The Process of Ontology Reuse and Application

Motivation for Reuse

The re-engineering of the panel layout application in Slang was performed prior to this experiment in ontology reuse as part of a wider objective of capturing engineering knowledge in an explicit, reusable form [1]. A physical quantity in engineering software is typically represented as a real-valued variable. So, in Slang, the weight of a panel was represented by a function mapping things of sort `panel` to things of sort `real` – i.e. `weight: panel -> real`. However, in this formulation, it is not possible to convert between different units, nor to test for dimensional consistency of equations.

Our goal in this experiment was to obtain these benefits. Instead of rewriting the original application from scratch, we tried to reuse and apply the engineering math ontology by incorporating it into this application. This ontology defines in a principled implementation-neutral manner, a set of concepts and relations for representing and manipulating physical quantities and units of measure.

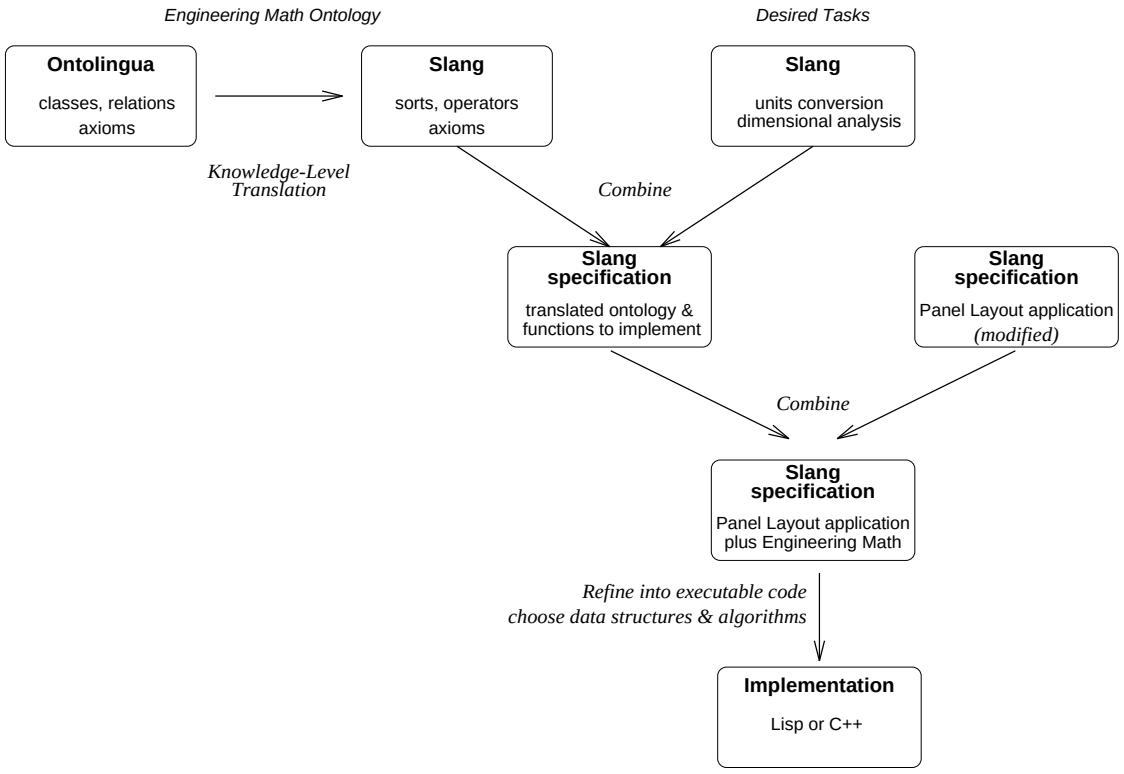


Figure 1: Process of Ontology Reuse and Application

Steps to Reuse

Due to the unique nature of Specware, the process of converting the engineering math ontology into an implemented format is not a single translation step into a target knowledge representation language. Instead, we first translate the ontology into knowledge-level Slang specifications; these are refined into executable code in a separate step. Overall, our experiment involved the following steps (see figure 1):

1. **Understanding the Ontology and Finding Kernel to Reuse**– Read the Ontolingua axioms and associated documentation to understand the engineering math ontology. Next, identify an initial kernel of the ontology which will be sufficient to achieve the primary intended tasks of units conversion and dimensional analysis.

2. **Ontology Translation**– Convert the definitions and axioms comprising the knowledge-level Ontolingua representation into an equivalent knowledge-level Slang formulation.

3. **Task Specification and Refinement into Executable Code**– Define and specify functions that enable the required tasks to be performed. Define refinements of the specifications produced in the above steps which move closer and closer to the implementation, and ultimately into executable code.

4. **Verification**– Verify each refinement step, which will guarantee that the executable code is true to the original specification.

5. **Integration with Application**– Merge the specification of engineering math with the already existing specification of the panel layout example, and together refine this into executable code.

We now describe each of these steps in detail.

3.1 Understanding the Ontology & Finding a Kernel to Reuse

The engineering math ontology is well documented both as a technical paper [8] and in its Ontolingua form. The latter is web-browsable and contains additional documentation not found in the paper. The process of understanding the ontology continued throughout the translation phase, as more details were required. Overall, we found the quality and clarity of the ontology to be high, but we also found some problems.

The engineering math ontology covers a wide range of mathematical and physical concepts in the engineering domain. Many of these things (e.g. tensors) are not of immediate relevance to our simplified panel layout application. Therefore, the first step in this experiment was to identify a kernel, a subset of the full ontology, which captures the essential ideas needed for our two intended tasks: units conversion and dimensional analysis. Initially, only this kernel would be translated into Slang.

The minimum we require from the engineering math ontology is the ability to convert between quantities specified in different units and to perform dimensional analysis. A important secondary requirement is that sufficient axioms are present to be able to prove properties that must hold in order for the algorithms to be valid. This requires the axioms describing the algebraic properties of these operations to be included (e.g. commutativity). The kernel of the engineering math ontology identified to meet our tasks is described below.

- *Physical Quantity*: a hypothetically measurable *amount of something* (e.g. 'the earth's diameter', 'Tom's age', '10 acres').

- *Physical Dimension*: the nature or kind of the something of which the quantity is an amount (e.g. length, time, area). Every quantity has an associated physical dimension.

- *Magnitude*: a measure of the amount of the something with respect to a particular unit of measure. (e.g. the magnitude of '3 feet' in yards is 1).

- *Unit of Measure*: a physical quantity which is used as a standard amount of something (e.g. foot, calorie). Units of measure are distinguished from ordinary quantities in that their magnitudes are always positive, no matter what the unit is.

- *Algebraic Operations*: One can add quantities of the same physical dimension to get a meaningful result (e.g. 15cm and 990mm) but it makes no sense to add 10 feet to 3 pounds. Also, one can divide one dimension by another and get another dimension (e.g. meter/second gives a velocity dimension), but it makes no sense to add dimensions.

  These operations over quantities and dimensions have specific properties, and are examples of common algebraic objects. For example, physical dimensions form a vector space over the reals, where vector addition is multiplication of physical dimensions, and scalar multiplication corresponds to exponentiation defined on physical dimensions and real numbers. Understanding these algebraic properties provides the basis for performing dimensional analysis, e.g. to prevent adding '3 feet' to '45 calories'.

3.2 Ontology Translation

Here we describe the process and results of translating the knowledge-level representation of the engineering math ontology in Ontolingua to an equivalent knowledge-level formulation in Slang.

Note, this is a somewhat unique way to use an Ontolingua ontology. Ontolingua was designed with the expectation that ontologies would be translated directly into an implemented language such as Loom or Clips, not to an equivalent knowledge-level formulation which had to be further transformed into an implementation.

Initially, importing Ontolingua's representation appeared to be reasonably straightforward. The following are some informal mapping rules for translating Ontolingua to Slang.

- ontologies in Ontolingua map to specifications in Slang;
- classes in Ontolingua naturally map to sorts in Slang;
- sub-classes map to sub-sorts, the inheritance works, although somewhat differently;
- slots, functions and relations in Ontolingua map to operators in Slang;
- instances map to constants;
- ontology inclusion maps to importing specifications;
- name translations for included ontologies for local use seems to have a close analogue in Slang.

Ontolingua is an unsorted logic, so conversion to Slang resulted in a more concise notation, avoiding frequent use of unary predicates required in Ontolingua. As a tradeoff, Slang's type theory requires using a relax operator to map sorts to supersorts, which is inconvenient and makes expressions harder to read. The differences seemed relatively minor.

This being the case, there was hope that some automated assistance might be possible, if only to kick-start the translation process. However, on closer examination, a number of important differences and other problems were identified. First, there are fundamental semantic differences. Second, even if we were able to produce a direct semantically correct translation using the above mapping rules, it may not use the expected representational idioms and conventions in the target language. This makes the translated version hard to understand and/or awkward to use. This is analogous to 'misuse' of Prolog by someone who is unfamiliar with the logic programming paradigm. Finally, whether or not the representational idioms are being correctly used, there may be representational choices that are influenced by the intended task. We discuss some of these issues below.

3.2.1 Semantics

Because Ontolingua is an extension of KIF, its semantics are founded on set theory; the semantics of Slang are founded on categorical type theory. To be confident of mapping classes and subclasses to sorts and subsorts, semantic differences would have to be carefully checked. If inferences were possible in one but not the other, work would

have to be done to identify and contain the semantic differences responsible for such inconsistencies, in order to produce correct translations. We have not resolved this issue.

3.2.2 Representational Style

Generating Unnecessary Sorts In Ontolingua, every class *must* be a subclass of *Thing*, which in the set-theoretic semantics of KIF means that it is not an unbounded set. In Slang, there is no need to distinguish bounded vs unbounded sets. Naive application of the above mapping rules would result in every Ontolingua class being introduced as a subsort; this is unnecessary and undesirable. Furthermore, every specification would have to have a superfluous sort corresponding to Thing, which brings us to the next point.

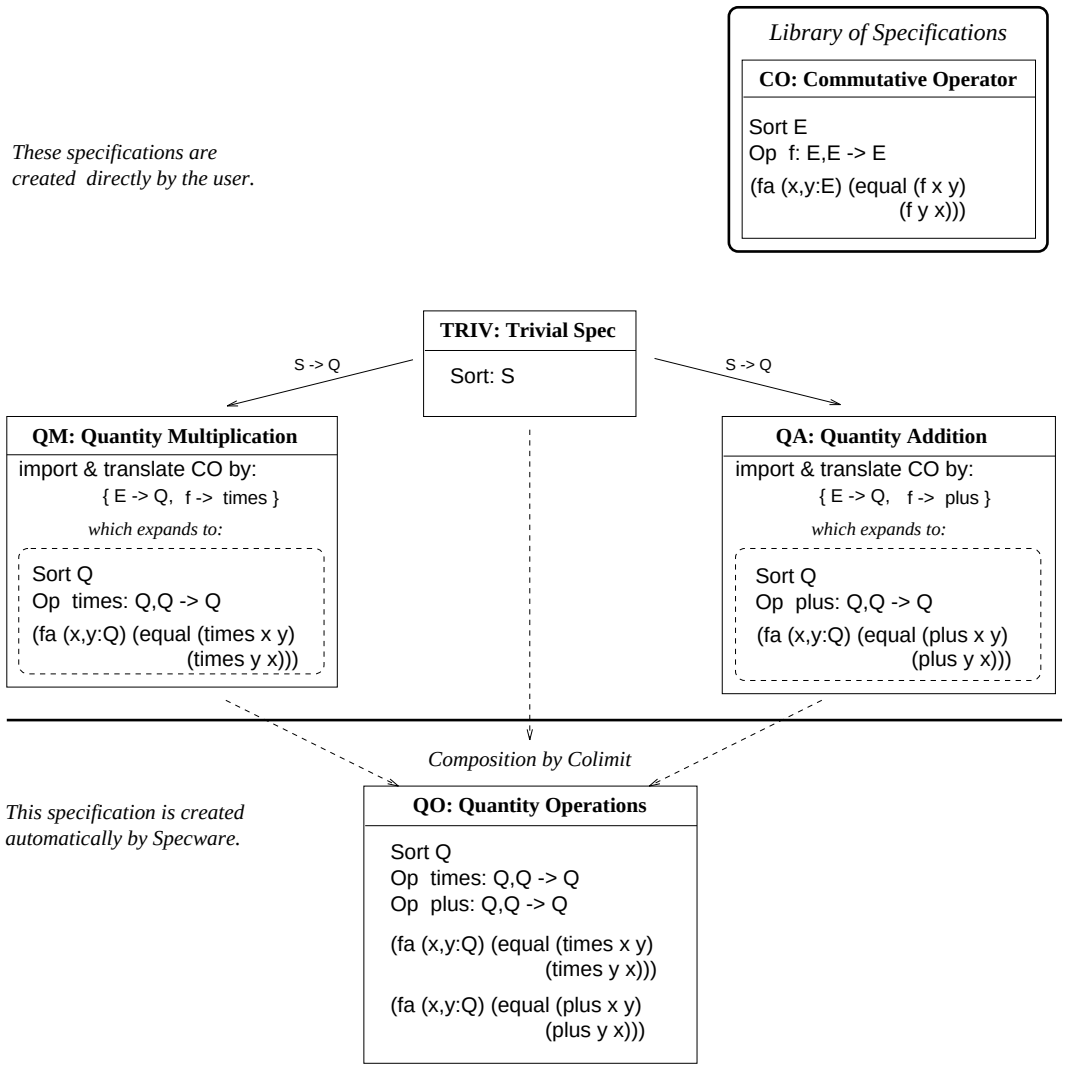
Translating Basic Infrastructure Virtually every ontology in Ontolingua imports the frame ontology, which is itself an extension of KIF. This is a substantial amount of infrastructure, including the class *Thing*, definitions of relations, numbers etc. Slang has its own notion of relations which is not explicitly defined in Slang, but rather is built in as part of the language. Slang specifications exist which define numbers. Instead of attempting to ensure that the whole Ontolingua/KIF infrastructure was represented in Slang, we introduced such things on an as-needed basis.

Composition The most obvious and significant difference between styles of use between Ontolingua and Slang is how composition of larger modules from smaller ones is accomplished. The difference is fundamental and is reflected in the composition primitives available in each language. Slang's composition mechanism is defined in using diagrams in category theory; it is more general but also has some drawbacks.

Ontology inclusion in Ontolingua results in the union of the sets of axioms in all included ontologies. This semantics allows for cyclic inclusion of ontologies (e.g. A includes B which includes C which includes A). Slang does not support this. On the other hand, this way of doing composition makes it impossible to compose an ontology in Ontolingua by including multiple copies of an existing ontology which are instantiated differently by using parts of the ontology as parameters.

In Slang, this capability is fundamental. Specifications are often highly abstract, and effectively instantiated many times over. For example, one can include two copies of the specification for commutative operators one for addition of physical quantities, and one for multiplication of physical quantities. This is made convenient by being able to introduce different local names for the different instantiations of the pieces of a specification. In one copy, you might call the operator which was named `f` in the original spec as `plus` and in the other, `times`. Importantly, in Slang, these are two genuinely different things (see figure 2).

Ontolingua allows you to give the same thing different local names when included into different ontologies. However, each local name is just a synonym; internally there is only one term to which all the synonyms refer. Continuing with the above example, suppose we have ontologies `CR` (commutative relation), `QM`, and `QA` (Quantity Multiplication and Addition); `f` is a relation defined in `CR`, `QM` and `QA` both include `CR`, `f` is renamed to `times` in `QM` and to `plus` in `QA`. This results in the *single* relation whose 'real name' is `f@CR`; the two local names `times@QM` and `plus@QA` are synonyms. If QA had also used the local name `times` for `f` then the second synonym would be `times@QA`



This figure illustrates how specifications can be parameterized during composition in Specware. The sort `E` *and the operator* `f` *are used as parameters of the specification* `CO` *which contains an axiom stating the commutative property.* `CO` *is imported twice and the local names changed. This results in two distinct copies, and thus two distinct sorts (*`Q.QM`*. and* `Q.QA`*), two distinct operators (*`times` *and* `plus`*) and two commutativity axioms. We desire a specification which combines these two copies which has both operators, but only one sort,* `Q`*. To achieve this, we create a diagram containing the specification* `TRIV` *whose single sort,* `S`*, is mapped to both* `Q.QM` *and* `Q.QA`*. The colimit operation performed by Specware joins the component specifications as required.*

Figure 2: Composition of Modules in Specware

instead – this difference would be of no real import. In Ontolingua, composition cannot be used this way to create two commutative relations.

Conversely, in Slang, when different specifications import the same component specification, *copies* of the original specification are inserted. In each copy, a unique local name may be given which is different from the original name, but unlike Ontolingua, they refer to *different* things, even if they have the same local name (see figure 2). Analogous to the example above, `CO` (Commutative Operator), QM, and QA are specifications, and f is an operator; `QM` and `QA` both import specification `CO`; and the local renaming is as above. There is *no relationship* between the local names for `f` in `QM` and `QA` unless you specify it. The two copies of f are called `times.QM` and `plus.QA`. If `QA` also renamed `f` to `times`, then the second copy of `f` would instead be called `times.QA`; it would not be a synonym for anything.

Suppose we wish to specify that `QM` and `QA` are both to be part of yet a larger specification (say `QO` for Quantity Operations). We really do want two distinct operators, so what we have so far is fine. However, the single sort, `E`, in `CO` (see figure 2) will also result in two distinct copies, `Q.QM` and `Q.QA` even if we give them the same local name. This is not desired; we want both operators to operate on the *same* sort Q. In order to get the desired effect, we create a formal diagram (as defined in category theory). The user can specify in the diagram that both copies, sort `Q.QM` and `Q.QA`, refer to a single term in the composite specification: `Q.QO`. All the specifications in the diagram are brought together using the colimit operation resulting in `QO`. In this case these three names are placed in an equivalence class of names which means they *are* synonyms, as in the Ontolingua example.

This flexibility enables reuse of specifications to operate in a parsimonious way. The price paid for this flexibility is that simple things can be more work to specify and may appear unduly complicated to those unfamiliar with this style of composition.

Ontolingua has alternate ways to achieve the parsimony and multiple reuse of basic concepts such as a binary operator which in Slang is accomplished by importing multiple parameterized versions of specifications (see the abstract algebra ontology in the KSL ontology library).

Each language supports a very different way of thinking about and specifying composition. There are advantages and disadvantages of each. Because these differences are so large, each language has its own idioms and conventions for what constitutes a 'good' way to modularize a theory. It turns out that the smallest ontologies in Ontolingua which are used to build up larger ontologies tend to be much larger than the smallest component specifications in Slang.

3.3 Task Specification and Refinement into Executable Code

We have not completed the discussion of ontology translation. Next we consider task definition and implementing the specifications.

3.3.1 Define Tasks

The translation of the initial kernel the engineering ontology into Slang resulted in an equivalent knowledge level formulation. We aimed for increased generality to support a variety of ways to implement operations on physical quantities and dimensions.

We did have certain tasks in mind, however, that the Slang specifications would be used for – namely, units conversion and dimensional analysis. Although the engineering

math ontology was designed to make it possible to perform these tasks, they were not specified. These had to be identified and specified in Slang in order to apply the ontology (see figure 1).

This adds a task-specific component to the Slang specification, and included the necessary functions for performing units conversion. What is still missing is sufficient information about how to *implement* the knowledge-level theory of physical quantities as executable functions. This we describe next.

3.3.2 Refinement

In keeping with the design decisions of the original ontology, the knowledge-level Slang encoding does not commit to any implementation details. Slang is a wide-spectrum language used to represent specifications ranging from purely abstract mathematical theories (e.g. for sets) which may be completely free of implementation biases, all the way to detailed specifications which include data structures and corresponding algorithms. For example, to implement sets using a list data structure, an algorithm is required for inserting new elements into lists which prevents duplicates.

Slang contains primitives for specifying refinements, which are formal mappings between specifications which move closer and closer to implementation. Formally, these consist of morphisms in the category of specifications and specification morphisms [12, 13]. The higher level specifications are refined step by step until sufficient implementation information is given that Specware may automatically generate executable code (currently in Lisp or C++).

In order to generate executable code, there are two main tasks. First, we must choose data structures for representing quantities and dimensions. Second, we must specify algorithms for performing operations on those data structures that correspond to operations and tasks to be performed on quantities and dimensions in the knowledge-level specification (e.g. addition, multiplication, units conversion).

If strings were used to represent physical dimensions, then string operations (e.g. concat) would be used to specify algorithms. Similarly, if lists were chosen, then list operations such as head and tail would be used. In this experiment, we chose to represent the dimension of a physical quantity as a tuple of real exponents, one for each of the basic dimensions (e.g. length, mass, time, money). For example, `(1 0 0 0)` and `(1 0 -1 0)` represent length and velocity respectively. The algorithm for multiplication of dimensions entails adding exponents.

In Slang, data structures are specified using the `sort-axiom` statement, which states an equivalence between two sorts. For example, the sort `Quantity` is equivalent to a two-tuple whose first argument is a real number (for magnitude), and whose second argument is a dimension. The sort, `Dimension`, in turn is a 4-tuple of reals. See figure 3 for Slang specifications of these data structures, as well as algorithms for computing multiplication of quantities and dimensions.

A formal relationship is stated between the knowledge-level and implementation-level specifications. These are refinements. There may be a sequence of refinements, or just one depending on the particular problem. After we specified the required refinements, Specware generated Lisp code which successfully performed the required operations. When different units are specified, the necessary conversions take place.

```
sort-axiom Quantity  = Real, Dimension
sort-axiom Dimension = Real, Real, Real, Real

definition of times : Dimension, Dimension -> Dimension is
axiom (equal (times d1 d2)
             (make-dimension (plus (length-expt d1) (length-expt d2))
                             (plus (time-expt   d1) (time-expt   d2))
                             (plus (mass-expt   d1) (mass-expt   d2))
                             (plus (money-expt  d1) (money-expt  d2))))
end-definition

definition of times : Quantity, Quantity -> Quantity is
  axiom (equal (times q1 q2)
       (make-quantity (times (magnitude-of q1)
                             (magnitude-of q2))
                      (times (dimension-of q1)
                             (dimension-of q2))))
end-definition
```

Figure 3: Data Structures and Algorithms

3.4 Verification

Specware is a system for the specification and *formal* development of software. It assists the user in producing executable code that is provably correct with respect to the specification. In the previous step, we described how the user defines a series of refinements which ultimately lead to executable code. It is the user's responsibility to prove the validity of each refinement step, using the theorem prover provided.

Above, we described how we implemented physical quantities and units of measure as tuples. Multiplication of physical dimensions in the abstract theory is commutative. Therefore, to verify the refinement, we must prove that multiplication of dimensions as defined by manipulating tuples is still commutative. Given that multiplication is implemented by adding real exponents, this follows from the fact that integer addition is commutative. Specware automatically generated the conjectures requiring proof; we have manually performed some of the proofs, but this verification step remains unfinished in our experiment.

Interestingly, we discovered that at least one axiom in the original Slang specification could not be proved from the refined implementation-level theory. This failure was traced not to an error in the refinement process, but to 'missing' axioms in the original Ontolingua ontology. Specifically, the proof required axioms about some concepts which were treated as primitives, and hence were undefined, in the original engineering math ontology. We subsequently added these axioms in our Slang formulation.

The absence of this axiom in the engineering math ontology need not be viewed as an error. Eventually definitions must ground in some primitives. Because we needed to prove theorems, we had to add axioms that were deemed unnecessary in the original ontology. This example also shows how the required tasks which the ontology is to support affects representational choices in the original design, and subsequently affects ease of reuse.

3.5 Integration with the Application

The last phase is to integrate the result into the panel layout application. At present, this integration has been partially completed. Integration is not a simple matter of merging the code from the previous step with the code that already had been generated by Specware from the specification of the panel layout application. This will not work because the functions and equations in the former, do not refer to any of the things in the engineering math specification (see figure 1). In particular, recall from § 3 that the original version of the application represented physical quantities as real numbers. This is the usual thing to do in engineering software. However because we wish to add the capabilities of doing units conversion and dimensional analysis, this is no longer appropriate.

We proceed by modifying the *specification* of the panel layout task, and then merging this with the engineering math specification. In the new version, physical quantities are represented not as real numbers, but as instances of the sort `physical-quantity` in the Slang language. So instead of `weight: panel -> real`, we have `weight: panel -> physical-quantity`. This allows us to gain the desired functionality.

Importantly, integration does not occur at the *implementation* level, e.g. by inserting Lisp functions from both applications into a larger application. Rather it occurs at the *specification* level. This is advantageous, as the changes are then being made at a level of abstraction above executable code, and hence are simpler to make and more reusable. For example, we could change the refinement step described in the previous section and represent physical dimensions as strings rather than as tuples of integers. The knowledge-level specification would remain unchanged.

After the specifications have been merged, the combined declarative specifications must *together*, be refined into executable code. We have generated a Lisp function that computes the volume of a panel given dimensions in different units, providing the answer in a specified unit. We have also demonstrated two possible ways to perform dimensional consistency analysis. This is step towards our goal of having a general capability for producing engineering applications whose equations are always dimensionally correct.

A very important feature in Specware is that the refinement of a composite specification may be specified in terms of refinements of its individual components. The idea is that to refine the module containing both the engineering math and the panel layout specifications, we need only refine the revised versions of the panel layout specifications, while leaving the engineering math refinements unchanged. This is then combined with the results in the previous step of refining the engineering math specification. As compositions and refinements of existing knowledge already exist for the original panel layout example, only minor adaptations were be needed.

This is fundamental to achieving reuse, but comes at a price: it is complicated. It requires a solid understanding of the some of the basics of category theory in general, and of the category of specifications and specification morphisms in particular. Furthermore, there are some limitations of the Specware system which can require creative work-arounds. We have much to do before the application is fully developed using all of the functionality of Specware.

Concluding Remarks We have now described the start-to-finish process of reusing and applying an existing ontology in an engineering application. The application is small, and this is a feasibility demonstration only. Also, it remains to complete the integration, and to demonstrate that the approach to ensuring dimensional consistency and

units conversion is sufficiently convenient to work with and scales to larger problems. Nevertheless, we have learned a great deal from this experiment.

4 Discussion and Conclusions

This experiment was designed to explore and test the premise that it is possible to reuse existing bodies of engineering knowledge that may be used in the design of aircraft. We now reflect on our experience, and draw some conclusions about the engineering math ontology itself, Ontolingua, and the issues involved in incorporating an ontology into an application.

Ontolingua and the Engineering Math Ontology Ontolingua appears satisfactory as an ontology representation language from the standpoints of a) providing adequate implementation-neutral constructs and 2) being able to read and understand the ontology. The KSL ontology server provides a convenient way to browse the ontologies.

The engineering math ontology is of high quality and at the right level of generality for our purposes. Reusing ontologies that are less generic and less carefully designed may be much more difficult.

Our experiment raised the general question: where should off-the-shelf ontologies "bottom out" into primitives with assumed definitions? The choice made for the engineering ontology, while perfectly appropriate for their envisioned uses, did not meet our requirements. The formal approach required by Specware meant that additional work was needed to augment the existing axiomatization. This is a general issue for the design of sharable ontologies, and one which requires further investigation.

Translation Perhaps the most important observation from this experiment is that there is significant manual effort involved in translating an ontology. This observation is particularly significant given recent interest in fully automatic translation between Ontolingua and AI languages supporting computation (e.g. Lisp, Prolog, LOOM) [7].

From our experience, we believe that the issue involves not only a rich source of difficult challenges which may in time be surmounted, there are some inherent barriers to fully automating the process of producing an accurate translation between any two highly expressive languages. Technical barriers include the requirement that the translator must embody full knowledge of the syntax *and* semantics of both languages. Related to this is the question of how much of the foundational infrastructure of a language must be translated before beginning with the ontology of interest. Even if these problems were overcome, and a provably correct mapping was possible between an arbitrary sentences in a given language, the output may be either hard to understand, or hard to use. This will likely arise when there are differences in the paradigms and/or intended purposes of the two languages. It will be especially difficult to integrate such automatic translations with hand-crafted knowledge bases which will use the correct idioms. An excellent example of this is the dramatically different styles of composition from component modules for Slang and Ontolingua.

Intrinsic problems that may never be overcome arise when design decisions required to make a good translation depend on information not present in the original ontology. In particular, one must consider the *tasks* to which the ontology is intended to serve. This is not to advocate abandoning work on translation tools, but to suggest that such tools will likely be user-assisted for the foreseeable future.

Specware We believe that Specware provides a very good platform for exploring ontology reuse and application. It has a very rich composition framework which supports reuse of specifications *and* refinements at many levels. Also, by supporting formal refinements from knowledge-level to implementation, it is a way to directly use and apply ontologies.

The Cost-Benefits of Ontology Reuse Although significant work was involved in the full process of incorporating the engineering math ontology into our application, our subjective conclusions are that it would have taken significantly longer to design the knowledge content of this ontology from scratch in our application, or from just reading the technical papers describing it. The definitions in Ontolingua provided a clear, formal description of the target vocabulary, and greatly assisted us in understanding what was meant by the terms it defined.

Finally, if the engineering math ontology becomes to some degree a standard, there is the longer-term potential that this and other applications built with it can interoperate more easily, as they conform to the same physical-quantities vocabulary. Although this conclusion is tentative, it is promising for achieving larger-scale knowledge reuse in the future.

Acknowledgements We are grateful to Mike Barley for helpful discussions about ontology translation.

References

[1] M. Barley, P. Clark, K. Williamson, and S. Woods. The neutral representation project. In *Proc AAAI-97 Spring Symposium on Ontological Engineering*. AAAI Press, 1997.

[2] P. Borst, H. Akkermans, and J. Top. Engineering ontologies. *International Journal of Human-Computer Studies (submitted)*, 1996.

[3] R.L. Crole. *Categories for Types*. Cambridge University Press, 1993.

[4] M. Cutkosky, R. Engelmore, R. Fikes, M. Genesereth, T. Gruber, W. Mark, J. Tenenbaum, and J. Weber. PACT: An experiment in integrating concurrent engineering systems. *IEEE Computer*, pages 28–37, Jan 1993.

[5] R. Fikes, A. Farquhar, and J. Rice. Tools for assembling modular ontologies in ontolingua. In *Proceedings of AAAI-97*, pages 436–441, 1997.

[6] 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.

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

[8] T. Gruber and G. Olsen. An ontology for engineering mathematics. In *Proc. of the Fourth International Conference on Principles of Knowledge Representation and Reasoning*. Morgan Kauffman, 1994.

[9] J. McGuire, D. Kuokka, J. Weber, J. Tenenbaum, T. Gruber, and G. Olsen. SHADE: Knowledge-based technology for the re-engineering problem. *Concurrent Engineering: Applications and Research (CERA)*, 1(2), Sep 1993.

[10] B. Pierce. *Basic Category Theory for Computer Scientists*. MIT Press, 1991.

[11] M. (editor) Uschold. Knowledge level modelling: Concepts and terminology. *Knowledge Engineering Review*, 13(1), 1998.

[12] R. Waldinger, Y.V. Srinivas, A. Goldberg, and R. Jullig. *Specware Language Manual*, 1996.

[13] K. Williamson, M. Healy, and R. Jasper. Formally specifying engineering design rationale. Technical Report ISSTECH-97-011, Applied Research and Technology, The Boeing Company, 1997.