To appear in: KR'2000
http://www.cs.utexas.edu/users/pclark/papers

Knowledge Patterns

Peter Clark, John Thompson
Knowledge Systems
Boeing Mathematics and Computing Technology
MS 7L66, PO Box 3707, Seattle, WA 98124
{peter.e.clark,john.a.thompson}@boeing.com

Bruce Porter
Computer Science Dept.
University of Texas
Austin, TX 78712
porter@cs.utexas.edu

Abstract

When building a knowledge base, one frequently repeats similar versions of general theories in multiple, more specific theories. For example, when building the Botany Knowledge Base[Porter et al., 1988], we embedded a theory of *production* in representations of *photosynthesis, mitosis, growth,* and many other botanical processes. Typically, a general theory is incorporated into more specific ones by an inheritance mechanism. However, this works poorly in two situations: when the general theory applies to a specific theory in more than one way, and when only a selected portion of the general theory is applicable.

We address this problem with a knowledge engineering technique based on the explicit representation of *knowledge patterns*, i.e., general templates denoting recurring theory schemata, and their transformation (through symbol renaming) for importing into specific theories. This technique provides considerable flexibility. A knowledge pattern may be transformed in multiple ways, and each resulting theory can be imported in whole or in part. We describe an application built using this technique, then critique its strengths and weaknesses. We conclude that this technique enables us to better modularize knowledge-bases and to reuse their general theories.

1 The Limitations of Inheritance

Consider the following fragment of a hypothetical knowledge-base about computers, expressed in Prolog[1]:

```
% Basic facts about myComputer, an instance of the class of computers:
isa(myComputer,computer). [2]
speed(myComputer,400).        /* MHz */
ram_size(myComputer, 128).    /* MB */
disk_space(myComputer,2000). /* MB */
expansion_slots(myComputer,4).

% "Available RAM space is the total RAM minus the occupied RAM."
available_ram(Computer,A) :-
    isa(Computer,computer),
    ram_size(Computer,S),
    occupied_ram(Computer,R),
    A is S - R.

% "The number of free expansion-slots is the total number of slots minus the number filled."
free_slots(Computer,N) :-
    isa(Computer,computer),
    expansion_slots(Computer,X),
    occupied_slots(Computer,O),
    N is X - O.
```

The two clauses above are syntactically different, yet they both instantiate the same general axiom, which we could explicate as:

```
FREE_SPACE(X,S) :-
    isa(X,CLASS),
    CAPACITY(X,C),
    OCCUPIED_SPACE(X,O),
    S is C - O.
```

[1] Variables start with upper-case letters and are universally quantified; ':-' denotes reverse implication (←); ',' denotes conjunction; and `is` denotes arithmetic computation.

[2] `isa(I,C)` denotes that `I` is an instance of the class `C`.

As part of a general *container* theory, this axiom relates a container's free space, capacity, and occupied space.

The clauses for `available_ram` and `free_slots` are instantiations of this axiom just when a computer is modeled as a container of data and expansion cards, respectively. However, unless this general theory of *containers* is represented explicitly, its application to the domain of computers is only implicit. Clearly, we would prefer to explicitly represent the theory, then to reuse its axioms as needed.

This is typically done with inheritance. The knowledge engineer encodes an explicit theory of *containers* at a high-level node in a taxonomy, then its axioms are automatically added to more specific theories at nodes lower in the taxonomy. One axiom in our *container* theory might be:

```
free_space(Container,F) :-
    isa(Container,container),
    capacity(Container,C),
    occupied_space(Container,O),
    F is C - O.
```

To use inheritance to import this axiom into our *computer* theory, we assert that computers are containers and that `ram_size` is a special case (a 'subslot,' in the terminology of frame systems) of the `capacity` relation:

```
% "Computers are containers."[3]
subclass_of(computer,container).

% "RAM size is a measure of capacity."
capacity(X,Y) :-
        isa(X,computer),
        ram_size(X,Y).
```

However, this becomes problematic here as there is a second notion of "computers as containers" in our original axioms, namely computers as containers of expansion cards. If we map this notion onto our *computer* theory in the same way, by adding the axiom:

```
% "Number of explansion slots is a measure
of capacity"
capacity(X,Y) :-
        isa(X,computer),
        expansion_slots(X,Y).
```

then the resulting representation captures that a computer has two capacities (memory capacity and slot capacity), but loses the constraints among their relations. Consequently, memory capacity may be used to compute the number of free expansion slots, and slot capacity may be used to compute available RAM. This illustrates how the general container theory can be "overlaid" on a computer in multiple ways, but inheritance fails to keep these overlays distinct.

This problem might be avoided in various ways. We could insist that a general theory (e.g., *container*) is applied at most once to a more specific theory (although there is no obvious, principled justification for this restriction). We would then revise our representation so that it is not a computer, but a computer's *memory*, which contains data, and similarly that a computer's *expansion slots* contain cards. While this solves the current problem, the general problem remains. For example, we may also want to model the computer's memory as a container in other senses (e.g., of transistors, files, information, or processes), which this restriction prohibits.

Another pseudo-solution is to parameterize the container theory, by adding an argument to the *container* axioms to denote the *type* of thing contained, to distinguish different applications of the *container* theory. With the changes italicized, our axioms become:

```
% "Free space for content-type T = capacity
for T - occupied T."
free_space(Container,ContentType,F) :-
  isa(Container,container),
  capacity(Container,ContentType,C),
  occupied_space(Container,ContentType,O),
  F is C - O.

% "ram_size denotes a computer's RAM ca-
pacity."
capacity(X,ram,Y) :-
  isa(X,computer),
  ram_size(X,Y).
```

Again, this solves the current problem (at the expense of parsimony), but is not a good general solution. Multiple parameters may be needed to distinguish different applications of a general theory to a more specific one. For example, we would need to add a second parameter about the container's `Dimension` (say) to distinguish physical containment (as in: "a computer contains megabytes of data") from metaphysical containment (as in: "a computer contains valuable information"). This complicates our *container* axioms further, and still other parameters may be needed.

A second limitation of inheritance is that it copies axioms (from a general theory to a more specific one)

[3] We assume a general inheritance axiom `isa(I,SuperC) :- isa(I,C), subclass_of(C,SuperC).`

in an "all or nothing" fashion. Often only a selected part of a theory should be transferred. To continue with our example, the general *container* theory may include relations for a container wall and its porosity, plus axioms involving these relations. Because the relations have no counterpart in the *computer* theory, these relations and axioms should not be transferred.

These two problems arise because inheritance is being misused, not because it is somehow "buggy." When we say "A computer is a container," we mean "A computer (or some aspect of it, such as its memory) *can be modeled as* a container." Inheritance is designed to transfer axioms through the *isa* relation, not the *can-be-modeled-as* relation. Nevertheless, knowledge engineers often conflate these relations, probably because inheritance has been the only approach available to them. This leads to endless (and needless) debates on the placement of abstract concepts in taxonomies. For example, where should *container* be placed in a taxonomy with respect to *object, substance, process* and so on? Almost anything can be thought of as a container in some way, and if we pursue this route, we are drawn into debating these modeling decisions as if they were issues of some objective reality. This was a recurrent problem in our earlier work on the Botany Knowledge-Base [Porter et al., 1988], where general theories used as models (such as *connector* and *interface*) sit uncomfortably high in the taxonomy. The same issue arises in other ontologies. For example, *product* is placed just below *individual* in Cyc [Cycorp, Inc., 1996] and *place* is just below *physical-object* in Mikrokosmos [Mahesh and Nirenburg, 1995].

2 Knowledge Patterns

Our approach for handling these situations is conceptually simple but architecturally significant because it enables us to better modularize a knowledge-base. We define a *pattern* as a first-order theory whose axioms are not part of the target knowledge-base, but can be incorporated via a renaming of their non-logical symbols.

A theory acquires its status as a pattern by the way it is used, rather than by having some intrinsic property. First, the knowledge engineer implements the pattern as an explicit, self-contained theory. For example, the *container* theory would include the axiom:

```
free_space(Container,F) :-
    isa(Container,container),
    capacity(Container,C),
    occupied_space(Container,O),
    F is C - O.
```

Second, using terminology from category theory [Pierce, 1991], the knowledge engineer defines a *morphism* for each intended application of this pattern in the target knowledge-base. A morphism is a consistent[4] mapping of the pattern's non-logical symbols, or *signature*, to terms in the knowledge-base, specifying how the pattern should be transformed. Finally, when the knowledge base is loaded, morphed copies of this pattern are imported, one for each morphism. In our example, there are two morphisms for this pattern:

```
container -> computer
capacity -> ram_size
free_space -> available_ram
occupied_space -> occupied_ram
isa -> isa
```

and

```
container -> computer
capacity -> expansion_slots
free_space -> free_slots
occupied_space -> occupied_slots
isa -> isa
```

(The reason for mapping a symbol to itself, e.g., the last line in these morphisms, is explained in the next paragraph). When these morphisms are applied, two copies of the *container* pattern are created, corresponding to the two ways, described above, in which computers are modeled as containers.

There may be symbols in the pattern that have no counterpart in the target knowledge base, such as the thickness of a *container wall* in our computer example. In this event, the knowledge engineer omits the symbols from the morphism, and the morphing procedure maps each one to a new, unique symbol (generated by Lisp's gensym function, for example). This restricts the scope of these symbols to the morphed copy of the pattern in the target knowledge base. Although the symbols are included in the imported theory, they are invisible (or more precisely, hidden) from other axioms in the knowledge base. Note that we cannot simply delete axioms that mention these symbols because other axioms in the imported theory may depend on them.[5] This selection

[4] Two examples of inconsistent mappings are: (i) mapping a symbol twice, e.g., {A->X,A->Y}, (ii) mapping a function f to g, where g's signature as specified by the mapping conflicts with g's signature as already defined in the target KB, e.g., {f->g,A->X,B->Y}, where f : A → B in the source pattern but g is already in the target and does not have signature g : X → Y.

[5] Although specific axioms may be removed if they do not contribute to assertions about symbols that are imported. A dependency analysis algorithm could, in principle, identify and remove such "dead code".

of just those predicates and functions we require corresponds to Burstall and Goguen's 'derive' operation [Burstall and Goguen, 1977] which is used to build algebraic theories from others.

Our overall approach with knowledge patterns is similar to the use of theories and morphisms in the formal specification of software (e.g., [Goguen, 1986, Srinivas and Jullig, 1995, Williamson et al., 2000]), and part of our goal is to motivate, simplify, and apply it in the context of knowledge engineering. As these authors have shown, category theory, applied to algebraic theories, provides a formal basis for this approach. To apply this to logic, Burstall and Goguen [Burstall and Goguen, 1977] show how a first-order logic theory can be understood as a many-sorted algebraic theory (comprising a set of sorts, a set of operators over those sorts, and a set of laws that those operators must satisfy) by:

- introducing truth values as a sort, and two no-argument operators (constants) `true` and `false`.
- expressing predicates as operators which produce a truth value as a result.
- Defining boolean connectives as operations (e.g., $\wedge$: boolean, boolean $\rightarrow$ boolean).
- Replacing universally quantified variables with sorts (which are implicitly universally quantified).

Drawing from [Williamson et al., 2000], we can thus define signatures, specifications (our theories or patterns), and morphisms in terms of sorted logic (rather than algebra) as follows. A *signature* consists of:

1. A set S of sort symbols
2. A triple O = <C,F,P> of operators, where C is a set of constant symbols, F is a set of function symbols, and P is a set of predicate symbols

A *specification* (corresponding to our notion of theory or pattern) consists of:

1. A signature $Sig = <S, O>$, and
2. A set Δ of axioms over Sig

A *signature morphism* (in the context of this category) is a consistent mapping from one signature to another (from sort symbols to sort symbols, and from operator symbols to operator symbols). Finally, given two specifications $<Sig_1, \Delta_1>$ and $<Sig_2, \Delta_2>$, a signature morphism M between Sig_1 and Sig_2 is a *specification morphism* between the specifications iff:

$$\forall a \in \Delta_1, (\Delta_2 \vdash M(a)) \quad (1)$$

That is, every axiom a in Δ_1, after being translated by M, follows from Δ_2.

In our case of unsorted first-order logic, a pattern corresponds to a specification where all variables are of a single sort, and a morphism corresponds to a specification morphism. Statement (1) trivially holds because our approach deals with the special case where the resulting theory Δ_2 is by definition the translation of Δ_1 by M.

3 Using Patterns for Building a Knowledge-Base

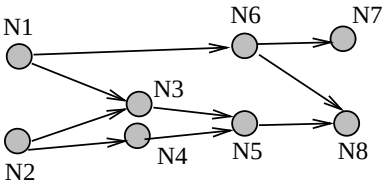
We encountered the limitations of inheritance and developed the approach of knowledge patterns while building KB-PHaSE, a prototype knowledge-based system for training astronauts to perform a space payload experiment called PHaSE (Physics of Hard Spheres Experiment). PHaSE involves projecting a laser beam through various colloidal suspensions of tiny spheres in liquids, to study the transitions among solid, liquid, and glass (not gas) states in microgravity. KB-PHaSE trains the astronaut in three ways. First, it provides a simple, interactive simulator in which the astronaut can step through the normal procedure of the experiment. Second, it introduces simulated faults to train the astronaut to recover from problems. Finally, it supports exploratory learning in which the astronaut can browse concepts in the knowledge-base and ask questions using a form-based interface. All three tasks use the underlying knowledge-base to infer: properties of the current experimental state, valid next actions, and answers to user's questions. The prototype was built as a small demonstrator, rather than for in-service use, to provide input to Boeing and NASA's Space Station Training Program. Details of KB-PHaSE are presented in [Clark et al., 1998] and the question-answering technology is described in [Clark et al., 1999].

Our interest here is how the underlying knowledge-base was assembled from component theories, rather than written from scratch. KB-PHaSE includes representations of many domain-specific objects (such as electrical circuits) and processes (such as information flow) that are derived from more general theories. For example, we can think of an electrical circuit in terms of a simple model of distribution, in which producers (a battery) distribute a product (electricity) to consumers (a light). To capture this in a reusable way, we formulated the general model of distribution as an independent, self-contained pattern, shown in the upper half of Figure 2. Then we defined a morphism that creates from it a model of electrical circuits, as shown the lower half of this Figure. Our general theory of distribution was built, in turn, by extending a general

Theory: DAG (Directed Acyclic Graph)

Synopsis

Name: `dag`
Summary: Core theory of directed acyclic graphs.
Uses: (none)
Used by: `blockable-dag`



Description: This component provides a basic axiomatization of DAGs, a fundamental structure for modeling many real-world phenomena. In a DAG, a NODE is directly linked TO and FROM zero or more other nodes [1]. A node REACHES all its downstream nodes [2], and is REACHABLE-FROM all its upstream nodes [3].
Signature: *Node*, *DAG*, *node-in*, *to*, $from$, *reaches*, *reachable-from*, *isa*.

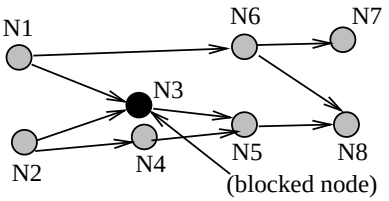
Axioms:

$\forall x, y\ to(x, y) \rightarrow isa(x, Node) \wedge isa(y, Node)$ [1]
$\forall x, y\ to(x, y) \leftrightarrow from(y, x)$
$\forall x, y\ to(x, y) \rightarrow reaches(x, y)$ [2]
$\forall x, y, z\ to(x, y) \wedge reaches(y, z) \rightarrow reaches(x, z)$
$\forall x, y\ from(x, y) \rightarrow reachable\text{-}from(x, y)$ [3]
$\forall x, y, z\ from(x, y) \wedge reachable\text{-}from(y, z) \rightarrow reachable\text{-}from(x, z)$
$\forall x, y\ isa(x, DAG) \wedge node\text{-}in(y, x) \rightarrow isa(y, Node)$

Theory: Blockable-DAG

Synopsis

Name: `blockable-dag`
Summary: Extension to DAG theory, in which nodes can be blocked (preventing reachability).
Uses: `dag`
Used by: `distribution-network`



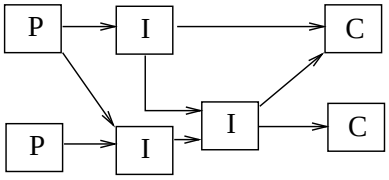
Description: A NODE may be BLOCKED or UNBLOCKED [1]. A node UNBLOCKED-REACHES a downstream node if there is a path of UNBLOCKED nodes connecting the two [2].
Signature: That for `dag`, plus *blocked*, *unblocked*, *unblocked-directly-reaches*, *unblocked-directly-reachable-from*, *unblocked-reaches*, *unblocked-reachable-from*,
Axioms: `dag` theory axioms, plus:

$\forall x\ isa(x, Node) \rightarrow blocked(x) \vee unblocked(x)$ [1]
$\forall x\ blocked(x) \leftrightarrow \neg unblocked(x)$
$\forall x, y\ to(x, y) \wedge \neg blocked(y) \rightarrow unblocked\text{-}directly\text{-}reaches(x, y)$
$\forall x, y\ unblocked\text{-}directly\text{-}reaches(x, y) \rightarrow unblocked\text{-}reaches(x, y)$ [2]
$\forall x, y, z\ unblocked\text{-}directly\text{-}reaches(x, y) \wedge$
$\quad unblocked\text{-}reaches(y, z) \rightarrow unblocked\text{-}reaches(x, z)$
$\forall x, y\ from(x, y) \wedge \neg blocked(y) \rightarrow unblocked\text{-}directly\text{-}reachable\text{-}from(x, y)$
$\forall x, y\ unblocked\text{-}directly\text{-}reachable\text{-}from(x, y) \rightarrow unblocked\text{-}reachable\text{-}from(x, y)$
$\forall x, y, z\ unblocked\text{-}directly\text{-}reachable\text{-}from(x, y) \wedge$
$\quad unblocked\text{-}reachable\text{-}from(y, z) \rightarrow unblocked\text{-}reachable\text{-}from(x, z)$

Figure 1: Two component theories (DAG and Blockable-DAG), used by KB-PHaSE.

Theory: Distribution Network

Synopsis



Name: `distribution-network`
Summary: Simple theory of producers, intermediaries and consumers.
Uses: `blockable-dag`
Used by: `electrical-circuit`

Description: A distribution network consists of three classes of nodes: PRODUCER, CONSUMER, and INTERMEDIARY [1], and the type of item transported is denoted by TRANSPORT-MATERIAL-TYPE (e.g., `Water`) Examples include: electrical circuits, hydraulic circuits, commuter traffic.
In this model, there is a flow of TRANSPORT-MATERIAL-TYPE from PRODUCERs to CONSUMERs via INTERMEDIARYs, providing the intermediary is not BLOCKED. A CONSUMER/INTERMEDIARY is SUPPLIED if there is at least one UNBLOCKED path to it from a SUPPLIER [2]. All elements in the network transport that network's TRANSPORT-MATERIAL-TYPE [3].

Signature: That for `blockable-dag`, plus *Producer*, *Consumer*, *Intermediary*, *Transport-Material-Type*, *supplied*, *product-type*, *consumes-type*.

Axioms: `blockable-dag` theory axioms, plus:

$\forall x\ isa(x, Producer) \rightarrow isa(x, Node)$ [1]
$\forall x\ isa(x, Consumer) \rightarrow isa(x, Node)$
$\forall x\ isa(x, Intermediary) \rightarrow isa(x, Node)$
$\forall x\ isa(x, Consumer) \wedge (\ \exists y\ isa(y, Producer) \wedge unblocked\text{-}reaches(y, x)\ ) \rightarrow supplied(x)$ [2]
$\forall x\ isa(x, Producer) \rightarrow product\text{-}type(x, Transport\text{-}Material\text{-}Type)$ [3]
$\forall x\ isa(x, Consumer) \rightarrow consumes\text{-}type(x, Transport\text{-}Material\text{-}Type)$

Theory: Electrical Circuit

Synopsis

Name: `electrical-circuit`
Summary: Top level concepts for reasoning about electrical circuits.
Uses: `distribution-network`, with morphism:

DAG	→	*Electrical-Circuit*	*Producer*	→	*Electrical-Power-Supply*
Node	→	*Electrical-Device*	*Consumer*	→	*Electrical-Appliance*
to	→	*wired-to*	*Intermediary*	→	*Electrical-Connector*
from	→	*wired-from*	*blocked*	→	*open*
supplied	→	*powered*	*unblocked*	→	*closed*
unblocked-reachable-from	→	*circuit-between*	*consumes-type*	→	*consumes-type*
Transport-Material-Type	→	*Electricity*	*product-type*	→	*product-type*
isa	→	*isa*			

Description: In this model, an ELECTRICAL-POWER-SUPPLY provides ELECTRICITY to ELECTRICAL-APPLIANCES via ELECTRICAL-CONNECTORS. ELECTRICAL-CONNECTORS (e.g., a switch) may be OPEN (off) or CLOSED (on). An appliance is POWERED if there is an open connection from at least one power supply [1].

Signature: Morphed version of `distribution-network`, using above mapping.

Axioms: Morphed version of `distribution-network` axioms, e.g.,

$\forall x\ isa(x, Electrical\text{-}Power\text{-}Supply) \rightarrow product\text{-}type(x, Electricity)$
$\forall x\ isa(x, Electrical\text{-}Appliance) \wedge (\ \exists y\ isa(y, Electrical\text{-}Power\text{-}Supply) \wedge$
$\quad circuit\text{-}between(y, x)\ ) \rightarrow powered(x)$ [1]

Figure 2: Component theories for distribution networks and electrical circuits, used by KB-PHaSE. The latter is defined as a morphism of the former.

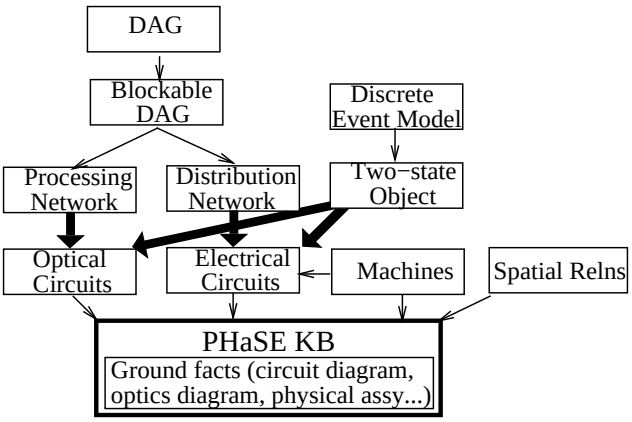


Figure 3: The component theories used in KB-PHaSE. Each box denotes a theory (set of rules) describing a phenomenon, and arcs denote inclusion relations, the thick arcs involving morphing the source.

theory of blockable directed acyclic graphs (blockable-DAGs), which in turn was built by extending a general theory of DAGs (Figure 1). The application, including these and other theories, is implemented in the frame-based language KM [Clark and Porter, 1999].

By separating these theories as modular entities, they are available for reuse. In this application, we also modeled information flow in the optical circuit (laser to camera to amplifier to disk) using a morphed pattern describing a processing network, which, in turn, was defined as an alternative extension of the basic blockable DAG theory, thus reusing this theory. Similarly, the general pattern of a "two-state object" occurs several times within KB-PHaSE (e.g., switches, lights, and open/closed covers), and this pattern was again made explicit and morphed into the knowledge base as required. These patterns and their inter-relationships are shown in Figure 3.

4 Related Work

There are several important areas of pattern-related work, differing in the type of reusable knowledge they encode and the way they encode it.

In software engineering there has been considerable work on formal methods for software specification, based on the construction and composition of theories, and using category theory (applied to algebraic specifications) as a mathematical basis (e.g., [Goguen, 1986, Srinivas and Jullig, 1995]). SpecWare is an example of a software development environment which is based on this approach and is capable of synthesizing software semi-automatically [Jullig et al., 1995]. As described in Section 2, our work can be viewed as motivating, simplifying, and applying similar ideas to the task of knowledge engineering.

Work on reusable problem-solving methods (PSMs), in particular KADS [Wielinga et al., 1992] and generic tasks [Chandrasekaren, 1986], addresses modularity and reuse in the context of procedural knowledge. PSMs are based on the observation that a task-specific method can be decomposed into more primitive – and more reusable – sub-methods, and that working with a library of such primitives may accelerate building a system and make it more understandable and maintainable. Work on PSMs shares the same general goal that we have — to identify and make explicit recurring generalizations — but it differs in two respects. First, while PSMs are (mostly) patterns of procedural inference, we have been targetting the basic domain knowledge (models) which those procedures may operate on. (Although, since logic has both a declarative and procedural interpretation, this distinction becomes blurred). Second, the mechanics of their usage differ: implementations of PSMs can be thought of as parameterized procedures, applied through instantiating their "role" parameters with domain concepts (e.g., the "hypotheses" role in a diagnosis PSM applied to medical diagnosis might be filled with disease types); in contrast, our patterns are closer to schemata than procedures, and applied instead through morphing.

Research on compositional methods for constructing knowledge bases (eg [Falkenhainer and Forbus, 1991, Clark and Porter, 1997, Noy and Hafner, 1998]) has explored factoring domain knowledge into component theories, analagous to factoring procedural knowledge into PSMs. A component theory describes relationships among a set of objects (its participants) and is applied in an analagous way to PSMs, by instantiating these participants with domain concepts. Knowledge patterns develop this idea in two ways. First, they provide further generalization, capturing the abstract structure of such theories. Second, their method of application differs (morphing, rather than axioms linking participants with domain concepts). This permits a pattern to be applied in multiple, different ways to the same object, as discussed in Section 1. Compositional modeling has also explored the automated, run-time selection of appropriate components to use [Falkenhainer and Forbus, 1991, Rickel and Porter, 1997], an important issue which we have not addressed here.

"Design patterns" in object-oriented programming (e.g., [Gamma et al., 1995]) are descriptions of common, useful organizations of objects and classes, to

help create specific object-oriented designs. They again try to capture recurring abstractions, but (in contrast to the approaches described earlier) their primary intent is as architectural guidance to the software designer, not as computational devices directly. As a result, they are (and only need be) semi-formally specified, and they do not require a method for their automatic application. ([Menzies, 1997] gives an excellent discussion of the relationship between object-oriented patterns and problem-solving methods). Another area of related work from programming languages is the use of template programming methods, where a code template is instantiated by syntactic substitution of symbols within it (e.g., Ada generics, C++ templates), corresponding to the syntactic implementation of pattern morphing, but without the associated semantics.

Work on analogical reasoning is also closely related, as it similarly seeks to use a theory (the base) to provide extra knowledge about some domain (the target), by establishing and using a mapping between the two. However, work on analogy has mainly focussed on identifying what the appropriate mappings between the base and target should be [Falkenhainer et al., 1986], a task which we have not addressed and which could be beneficial for us to explore further. In addition, an alternative way of applying our patterns would be to transform a domain-specific *problem* into the vocabulary of a pattern (and solve it there, and transform the solution back), rather than transforming the pattern into the vocabulary of the domain. In the PHaSE KB, for example, a query about the electrical circuit would be transformed to a query about a distribution network which was isomorphic to the electrical circuit, solved there, and the answer transformed back to the electrical circuit. This alternative approach is similar to (one form of) solution by analogy, in which the pattern (e.g., the distribution network) takes the role of the base, and the domain facts (e.g., the electrical circuit) the target [Falkenhainer et al., 1986]. It is also similar to the use of delegation in object-oriented programming (the target 'delegates' the problem to the base, which solves it and passes the solution back [Gamma et al., 1995, p20]). This variant approach for using patterns would allow some run-time flexibility, but would be more complex to implement and computationally more expensive at run-time.

Finally, work on microtheories and contexts (e.g., [Buvac, 1995, Blair et al., 1992]) is also related, where a microtheory (context) can be thought of as a pattern, and lifting axioms provide the mapping between predicates in the microtheory and the target KB which is to incorporate it. However, this work has typically been used to solve a different problem, namely breaking a large KB into a set of smaller, simpler (and thus more maintainable) pieces, rather than making recurring axiom patterns explicit, and it does not account for mapping the same microtheory multiple times (and in different ways) into the same target KB. Reasoning with lifting axioms can also be computationally expensive except in the simplest cases.

5 Discussion

For many representational tasks, inheritance provides a straightforward way of encoding relationships between domain-specific and abstract concepts, and the additional machinery of patterns is not necessary. Specifically, this is the case when there is a single, obvious way in which a specific concept instantiates a general one, and when all the general properties of concepts in the general theory transfer to the domain-specific ones. Psychological work on Category Structures provides support for this claim [Smith and Medin, 1981], and, computationally, inheritance is simple to understand and use.

However, as we have argued in Section 1, inheritance becomes inappropriate when a general theory can be applied in multiple ways, and when we wish to restrict how and which properties transfer to more domain-specific concepts. The pattern approach also addresses the issue of multiple theory applications through its use of morphisms (one for each application), and can also selectively transfer information, by hiding relations that do not have correlates in the target KB (Section 2). In addition, it is relatively intuitive, efficient, and easy to use.

However, our approach also has limits. First, it does not allow a system to make run-time modeling decisions, as general theories are morphed when the knowledge base is loaded. Second, it does not address the issue of *finding* relevant knowledge patterns in the first place, or deciding the appropriate boundaries of patterns (this is left to the knowledge engineer). Finally, we do not address the issue of finding the appropriate mappings between patterns and the domain; this again is left to the knowledge engineer. As mentioned earlier, this is a primary focus of research in the related field of analogical reasoning [Falkenhainer et al., 1986].

Note that patterns are not an essential prerequisite for building a knowledge-based system. In the PHaSE application, for example, we could have simply defined the PHaSE electrical circuit, implemented axioms about the behavior of electrical circuits, and answered circuit questions, all within the electrical vo-

cabulary. This would be a completely reasonable approach for a single-task system; however, to achieve reuse within a multifunctional system (such as KB-PHaSE), or between systems, it becomes preferable to extract the more general abstractions, as this paper has described. Patterns do not enable better reasoning, rather they are to help reuse.

6 Summary

In this paper, we have highlighted both the need for and difficulty of capturing and applying general theories as modular units in a KB. We have described and critiqued an approach for doing this, based on capturing those theories as "patterns" and incorporating them by morphing, and we have described an application system assembled in this way.

The significance of this approach is that it allows us to better modularize a knowledge-base and isolate general theories as self-contained units for reuse. It also allows us to control and vary the way those theories are mapped onto an application domain, and it better separates the "computational clockwork" of a general theory from the domain phenomena which it is considered to reflect. In addition, the approach is technically simple and not wedded to a particular implementation language. In the long-term, we hope this will help foster the construction of reusable theory libraries, an essential requirement for the construction of large-scale knowledge-based systems.

Acknowledgements

Thanks to Tim Menzies for valuable comments, in particular on the relationship of knowledge patterns to problem-solving methods and design patterns. Thanks also to Mike Healy, Rob Jasper, Mike Uschold, and Keith Williamson for their categorical assistance.

References

[Blair et al., 1992] Blair, P., Guha, R. V., and Pratt, W. (1992). Microtheories: An ontological engineer's guide. Tech Rept CYC-050-92, MCC, Austin, TX.

[Burstall and Goguen, 1977] Burstall, R. M. and Goguen, J. A. (1977). Putting theories together to make specifications. In *IJCAI-77*, pages 1045–1058.

[Buvac, 1995] Buvac, S., editor (1995). *Proc AAAI-95 Fall Symposium on Formalizing Context*. AAAI. http://www-formal.stanford.edu:80/bovac/95-context-symposium/.

[Chandrasekaren, 1986] Chandrasekaren, B. (1986). Generic tasks in knowledge-based reasoning: High-level building blocks for expert system design. *IEEE Expert*, pages 23–30.

[Clark and Porter, 1997] Clark, P. and Porter, B. (1997). Building concept representations from reusable components. In *AAAI-97*, pages 369–376, CA. AAAI. (www.cs.utexas.edu/users/pclark).

[Clark and Porter, 1999] Clark, P. and Porter, B. (1999). KM – the knowledge machine: Users manual. Technical report, AI Lab, Univ Texas at Austin. (http://www.cs.utexas.edu/users/mfkb/km.html).

[Clark et al., 1998] Clark, P., Thompson, J., and Dittmar, M. (1998). KB-PHaSE: A knoweldge-based training tool for a space station experiment. Technical Report SSGTECH-98-035, Boeing Applied Research and Technology, Seattle, WA. (http://www.cs.utexas.edu/users/pclark/papers).

[Clark et al., 1999] Clark, P., Thompson, J., and Porter, B. (1999). A knowledge-based approach to question-answering. In Fikes, R. and Chaudhri, V., editors, *Proc. AAAI'99 Fall Symposium on Question-Answering Systems*. AAAI. (http://www.cs.utexas.edu/users/pclark/papers).

[Cycorp, Inc., 1996] Cycorp, Inc. (1996). The cyc public ontology. (http://www.cyc.com/public.html).

[Falkenhainer and Forbus, 1991] Falkenhainer, B. and Forbus, K. (1991). Compositional modelling: Finding the right model for the job. *Artificial Intelligence*, 51:95–143.

[Falkenhainer et al., 1986] Falkenhainer, B., Forbus, K. D., and Gentner, D. (1986). The structure-mapping engine. In *AAAI-86*, pages 272–277.

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns*. Addison-Wesley.

[Goguen, 1986] Goguen, J. A. (1986). Reusing and interconnecting software components. *Computer*, pages 16–28.

[Jullig et al., 1995] Jullig, R., Srinivas, Y. V., Blaine, L., Gilham, L.-M., Goldberg, A., Green, C., McDonald, J., and Waldinger, R. (1995). Specware language manual. Technical report, Kestrel Institute. (www.kestrel.edu).

[Mahesh and Nirenburg, 1995] Mahesh, K. and Nirenburg, S. (1995). A situated ontology for practical NLP. In *Proc. IJCAI-95 Workshop on*

Basic Ontological Issues in Knowledge Sharing. (http://crl.NMSU.Edu/Research/Projects/mikro/).

[Menzies, 1997] Menzies, T. (1997). Object-oriented patterns: Lessons from expert systems. *Software – Practice and Experience*, 27(12):1457–1478. (http://www.csee.wvu.edu/~timm/).

[Noy and Hafner, 1998] Noy, N. F. and Hafner, C. D. (1998). Representing scientific experiments: Implications for ontology design and knowledge sharing. In *AAAI-98*, pages 615–622.

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

[Porter et al., 1988] Porter, B. W., Lester, J., Murray, K., Pittman, K., Souther, A., Acker, L., and Jones, T. (1988). AI research in the context of a multifunctional knowledge base: The botany knowledge base project. Tech Report AI-88-88, Dept CS, Univ Texas at Austin.

[Rickel and Porter, 1997] Rickel, J. W. and Porter, B. W. (1997). Automated modeling of complex systems to answer prediction questions. *Artificial Intelligence*.

[Smith and Medin, 1981] Smith, E. E. and Medin, D. L. (1981). *Categories and Concepts*. Harvard Univ., Cambridge, Ma.

[Srinivas and Jullig, 1995] Srinivas, Y. V. and Jullig, R. (1995). Specware: Formal support for composing software. In *Proc. Conf. on the Mathematics of Program Construction*, Kloster Irsee, Germany. (Also Kestrel Tech Rept KES.U.94.5, http://www.kestrel.edu/HTML/publications.html).

[Wielinga et al., 1992] Wielinga, B. J., Schreiber, A. T., and Breuker, J. A. (1992). KADS: A modelling approach to knowledge engineering. *Knowledge Aquisition*, 4(1).

[Williamson et al., 2000] Williamson, K., Healy, M., and Barker, R. (2000). Reuse of knowledge at the appropriate level of abstraction. In *Proc. Sixth Int. Conf. on Software Reuse (ICSR'6)*.