Automated Support for Composition of Transformational Components in Knowledge Engineering

by

Marco Schorlemmer, Stephen Potter, Dave Robertson

Informatics Research Report EDI-INF-RR-0137

Division of Informatics
http://www.informatics.ed.ac.uk/

June 2002

Automated Support for Composition of Transformational Components in Knowledge Engineering

Marco Schorlemmer, Stephen Potter, Dave Robertson

Informatics Research Report EDI-INF-RR-0137

DIVISION *of* INFORMATICS
Centre for Intelligent Systems and their Applications

June 2002

Abstract :

The knowledge engineering world provides a rich source of software components for transforming formally expressed knowledge on a large scale, such as induction systems, knowledge base refiners and ontology merging tools. Although most of these systems have been designed as stand-alone components, there is interest in making them accessible on the Web, with the ultimate goal in mind that a knowledge engineer should be able, with a small amount of intellectual effort, to locate and assemble sequences of these components to perform complex transformations on large repositories of knowledge. The sorts of transformations used in knowledge engineering are not always trustworthy: some may not preserve the semantics of the knowledge transformed; some may not be able to perform a given transformation reliably under all circumstances. Therefore, it is crucial to have ways of inspecting the key properties we expect to be preserved by each transformational component and of describing how these properties change as new transformations are applied.

We present initial experiments on a large-scale knowledge engineering problem and show how an abstract characterisation of knowledge-transformation steps, accompanied by a customisable editor, can allow a high degree of automation in this task. With such an editor we can analyse and represent sequences of general transformation steps and check if properties such as subsumption, completeness and soundness are preserved during different stages of the transformation, by analysing the structure of these sequences.

Keywords :

Automated Support for Composition of Transformational Components in Knowledge Engineering

W. Marco Schorlemmer David Robertson Stephen Potter

Centre for Intelligent Systems and their Applications
Division of Informatics
The University of Edinburgh

Abstract

The knowledge engineering world provides a rich source of software components for transforming formally expressed knowledge on a large scale, such as induction systems, knowledge base refiners and ontology merging tools. Although most of these systems have been designed as stand-alone components, there is interest in making them accessible on the Web, with the ultimate goal in mind that a knowledge engineer should be able, with a small amount of intellectual effort, to locate and assemble sequences of these components to perform complex transformations on large repositories of knowledge. The sorts of transformations used in knowledge engineering are not always trustworthy: some may not preserve the semantics of the knowledge transformed; some may not be able to perform a given transformation reliably under all circumstances. Therefore, it is crucial to have ways of inspecting the key properties we expect to be preserved by each transformational component and of describing how these properties change as new transformations are applied.

We present initial experiments on a large-scale knowledge engineering problem and show how an abstract characterisation of knowledge-transformation steps, accompanied by a customisable editor, can allow a high degree of automation in this task. With such an editor we can analyse and represent sequences of general transformation steps and check if properties such as subsumption, completeness and soundness are preserved during different stages of the transformation, by analysing the structure of these sequences.

1. Introduction

One of the new challenges for automated software engineering raised by the Web is that of managing lifecycles of large volumes of formally expressed knowledge. It is now straightforward to access through a Web-browser systems which will, for example, allow us to assemble knowledge bases and ontology descriptions from libraries of components. Often, the knowledge we create in this way is too extensive to adapt by hand (the example we use in this paper is a 5.3 Mb ontology) so we must deploy various automated tools to trim it down, partition it and integrate it with other knowledge bases and reasoning systems. This is an exercise in combining large-scale transformation systems. It is difficult from an engineering point of view because typically we apply many individual transformations over the lifespan of a knowledge-based system, so it is easy to lose track of the "big picture" as far as key properties of the knowledge we are manipulating is concerned.

In this paper we present one way of addressing this problem. We assume that the results of transformation systems of interest are too large, and their effects too varied and complex to analyse *ab initio* from their source code but that it is possible to specify properties which they preserve of the knowledge they transform. For example, ID3 [16] and AQ11 [15] both are algorithms for performing rule induction. There are various implementations of each of these algorithms but all of them accept as input a database in a given syntactic form; produce a rulebase in a different syntactic form; and claim to preserve the property that the output rulebase allows at least those conclusions which would have been obtainable from the input database. These sorts of properties often are the primary concern of knowledge engineers, who may have no interest in the algorithmic or coding details of the Web-based generalisation they applied to their knowledge base as long as they know it actually was a generalisation (perhaps of a given kind). We supply a language — a *lifecycle calculus* — in which transformations are classified in terms of these sorts of key properties. We then build mechanisms for supporting large scale transformation where transformation steps are specified in the lifecycle calculus but applied, via automated systems, through selection of transformational components which are consistent with that specification. This provides a way of controlling sequences of transformation through lifecycle specifications and of expressing standard practice in large-scale transformation using libraries of such specifications designed for common tasks.

The lifecycle calculus which we present is the most general and abstract we could devise for this purpose. This is an ideal vehicle for research prototyping but in practice we do not expect it to be used in its general form. Instead, its use is controlled by the ways in which it is visualised, the combinations of calculus steps we allow, and the kinds of transformational component made available. In Section 2 we show one such way of applying the calculus in a controlled fashion. There is, however, a spectrum of ways of using it, ranging from a high degree of automated support through to no automated support (with the calculus serving only as a way of recording on paper the knowledge transformations applied). Our concern in this paper is with highly automated support. The overall motivation for emphasising automation is similar to that for coarse-grain synthesis systems such as Amphion [13, 12]: we wish to provide tools which make it tractable to cope with large scale software engineering problems. Unlike systems such as Amphion, we are concerned with transformations of knowledge rather than synthesis of code.

2. Formal Lifecycles: an Example

A typical problem faced by Internet-based knowledge engineers is the following. Suppose we have an open agent architecture in which suppliers of programs for transforming knowledge in different ways supply these as “services”. Such services need to be brokered in some way (otherwise they would be hard to find and use). Brokering requires some specification of the competences of each service, which could be expressed simply using some propositional, domain-specific language or could be a more complex input/output specification. Imagine that in such an open architecture we discover a very large and complex knowledge base which has been produced by one of these services and which appears to meet our current needs. But does it, in fact, possess the sort of qualities that we demand of the knowledge we use? That will be impossible to decide by reading the knowledge base; it will probably be impossible to decide by static analysis, and a decision by testing will probably be inconclusive. If we know the competence specification of the service which constructed it then this might reveal more about why and how it was built. It will not, however, tell us if some other related knowledge base might have been better. To know that we must know the history of transformations which link the knowledge bases built by all our services. This is the information provided by a lifecycle history. A lifecycle calculus allows us to construct and reason about such histories.

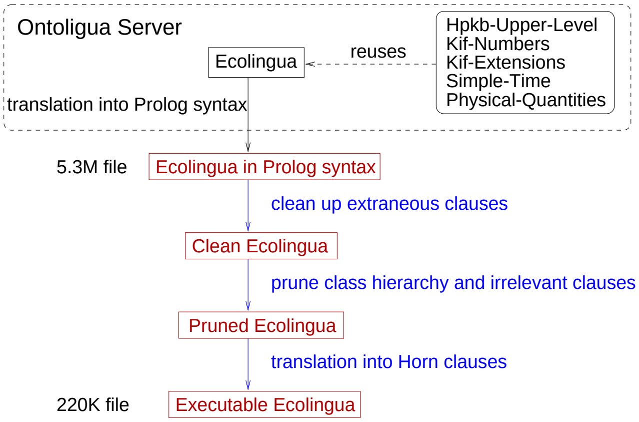


Figure 1. Ecolingua’s lifecycle

2.1. Ecolingua’s Lifecycle

In order to illustrate the need for a formal, abstract account of knowledge lifecycles and to show the advantage such an account yields, we shall use a real knowledge-engineering scenario that our colleagues encountered here at The University of Edinburgh. It consists of the lifecycle of an ontology for ecological meta-data, called Ecolingua [3, 5].

With the intention of specifying an ontology for the description of ecological data, Brilhante and Robertson chose not to build the ontology from scratch, but to do it the “proper way”, that is, by reusing publicly available ontologies of related fields and incorporating their specifications into Ecolingua. For this reason they turned to the library of sharable ontologies provided by the Ontolingua Server [8]. Hence, Ecolingua was first constructed with the server’s editor by reusing classes from other ontologies in the server’s library, and then automatically translated into Prolog syntax, Brilhante and Robertson’s preferred language, by using the server’s translation service.

To their dismay the outcome of the translation was an overly large 5.3 Mb file. That was so because the definitions of proper Ecolingua classes use external classes defined in five other ontologies, and because the translation service of the server blindly joins the specification of all ontologies together, including those of very general ontologies like Simple-Time and Physical-Quantities[1], before performing the actual translation into Prolog syntax. There was no way that the ontology as produced by the Ontolingua Server was going to be useful, unless it was significantly reduced in order to get a smaller and more manageable set of axioms. This reduction process could not be done manually, since the logical integrity of the ontology may have been affected. This forced Brilhante and Robertson to implement filters

[1] Both ontologies are available from the Ontolingua Server.

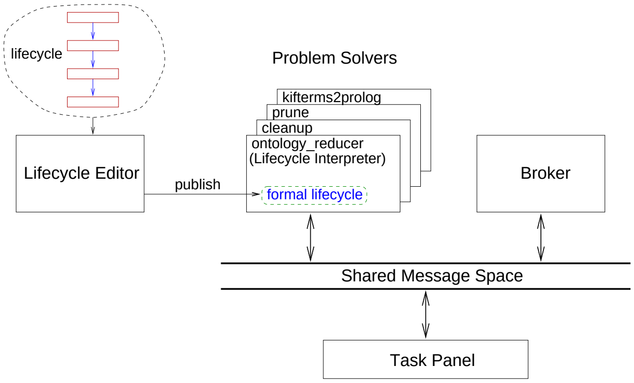
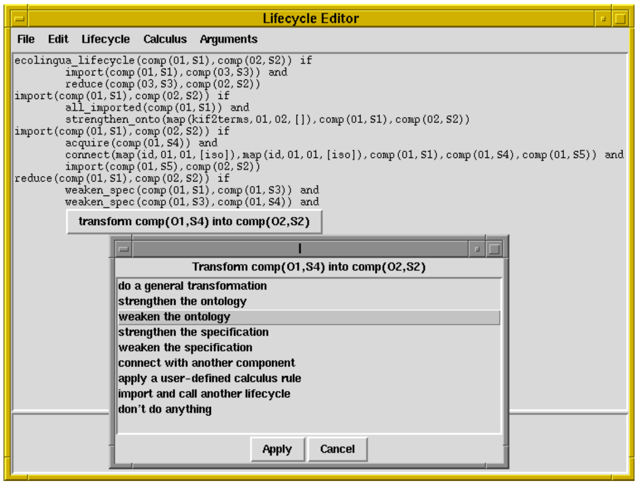


Figure 2. Overview of the architecture


Lifecycle Editor
File
Edit
Lifecycle
Calculus
Arguments
transform comp(O1,S4) into comp(O2,S2)
Transform comp(O1,S4) into comp(O2,S2)
do a general transformation
strengthen the ontology
weaken the ontology
strengthen the specification
weaken the specification
connect with another component
apply a user-defined calculus rule
import and call another lifecycle
don't do anything
Apply
Cancel

```
ecolingua_lifecycle(comp(O1,S1),comp(O2,S2)) if
        import(comp(O1,S1),comp(O3,S3)) and
        reduce(comp(O3,S3),comp(O2,S2))
import(comp(O1,S1),comp(O2,S2)) if
        all_imported(comp(O1,S1)) and
        strengthen_onto(map(kif2terms,O1,O2,[]),comp(O1,S1),comp(O2,S2))
import(comp(O1,S1),comp(O2,S2)) if
        acquire(comp(O1,S4)) and
        connect(map(id,O1,O1,[iso]),map(id,O1,O1,[iso]),comp(O1,S1),comp(O1,S4),comp(O1,S5)) and
        import(comp(O1,S5),comp(O2,S2))
reduce(comp(O1,S1),comp(O2,S2)) if
        weaken_spec(comp(O1,S1),comp(O1,S3)) and
        weaken_spec(comp(O1,S3),comp(O1,S4)) and
```

Figure 3. Editing Ecolingua's lifecycle

that first deleted all extraneous clauses (over-general facts, definitions of self-subclasses, duplicated classes), and then pruned the class hierarchy and removed irrelevant clauses accordingly. Finally, since the translation service did not provide executable Prolog code (it only yielded the ontological axioms originally specified in KIF [9] as Prolog terms) a final translation into Horn clauses was necessary, in order to be able to execute the ontology with a Prolog interpreter. Figure 1 shows this fragment of Ecolingua's lifecycle. The whole reduction process is explained in detail in [5].

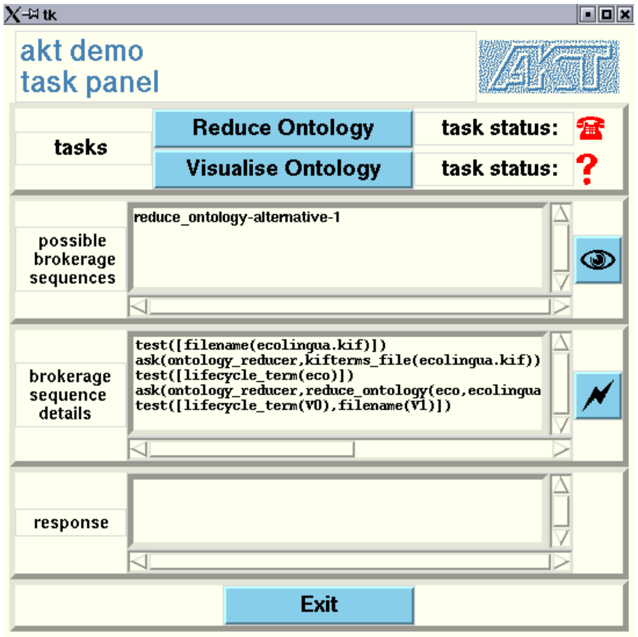
We postulate that particular sequences of transformation steps like those illustrated in Figure 1 may be common in particular domains, and perhaps to particular forms of knowledge component. In fact, the same problem encountered by Brilhante and Robertson with Ecolingua, and the same need for automatic ontology reduction will be faced by any knowledge engineer who attempts to specify a particular ontology in the Ontolingua Server, by reusing classes from sharable ontologies, and who asks for a translation into Prolog syntax. As such, the ability to generalise and compile these sequences of transformations into formally represented specifications of lifecycle patterns, and making these available within a distributed environment as integrated competences of an interpreter of such patterns, would encourage more efficient behaviour when faced with the need to make similar modifications in the future.

2.2. Editing and Executing Lifecycle Patterns

Let us now illustrate what a prototype for editing and executing lifecycle patterns looks like. Figure 2 shows the overview of an agent architecture we have been investigating. In such an architecture different knowledge engineers could edit a formal representation of abstract lifecycle patterns by means of a *lifecycle editor*, could then publish these patterns as "competences" of a *lifecycle interpreter* — an agent with the capability of executing lifecycle patterns — and could finally execute these patterns through a *task panel*, which allows them to chose between the alternative problem solvers that perform the knowledge transformations captured in the abstract pattern in a domain-specific way, if more than one exist.

Figure 3 shows the *lifecycle editor* that enables a knowledge engineer to analyse the lifecycle of a knowledge component, extract its abstract pattern, and devise a formal representation of it. In particular, it shows the formal counterpart of Ecolingua's lifecycle as portrayed in Figure 1 at the stage where the engineer is about to determine that the last transformation step should be an 'Ontology Weakening' step. In our prototype we chose a representation based on Horn clauses, where each clause represents a lifecycle pattern or sub-pattern, and each literal in the body of the clause represents the "call" to an abstract lifecycle step. In Section 3.2 we justify our choice of these steps, such as 'Ontology Weakening', and give definitions for a formal lifecycle calculus. Notice that, although the depicted lifecycle editor uses an explicit formal notation as Horn clauses, it is possible to hide much of the formality by using domain- or taste-specific editing operations. For example, if we built an editor that was especially for ontology refinement, then we might have translations between formal languages as "primitive steps", and these would fit into the basic underlying calculus as instantiations of abstract transformation steps .

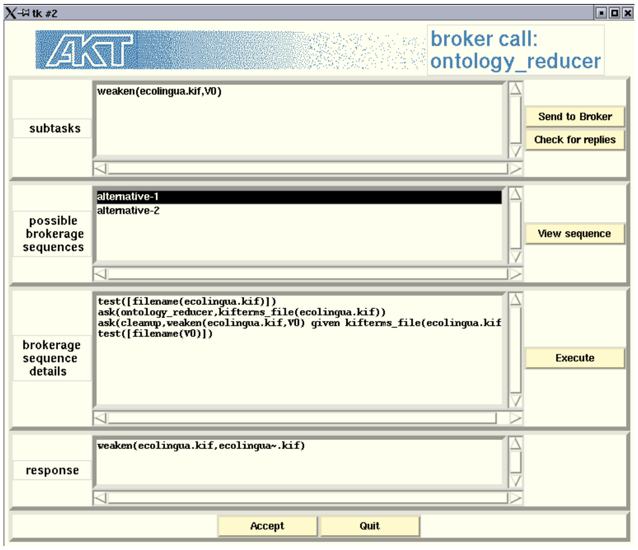
Once the lifecycle of a knowledge component like Ecolingua is edited, we may publish it as a "competence" of a *lifecycle interpreter*, i.e., a meta-interpreter capable of executing the formal representation of a lifecycle. This lifecycle is described at a generic level, and in order to be able to run the lifecycle in a particular domain, specific solvers capable of performing abstract lifecycle steps are needed. As a possible realisation for lifecycle execution we have been investigating the use of an agent-based environment provid-



```
reduce_ontology-alternative-1
```

```
test([filename(ecolingua.kif)])
ask(ontology_reducer,kifterms_file(ecolingua.kif))
test([lifecycle_term(eco)])
ask(ontology_reducer,reduce_ontology(eco,ecolingua
test([lifecycle_term(V0),filename(V1)])
```

Figure 4. Executing Ecolingua's lifecycle



```
weaken(ecolingua.kif,V0)
```

```
alternative-1
alternative-2
```

```
test([filename(ecolingua.kif)])
ask(ontology_reducer,kifterms_file(ecolingua.kif))
ask(cleanup,weaken(ecolingua.kif,V0) given kifterms_file(ecolingua.kif
test([filename(V0)])
```

```
weaken(ecolingua.kif,ecolingua~.kif)
```

Figure 5. Choosing a particular solver for an abstract lifecycle step

ing a brokering service that offers a knowledge engineer the option of choosing among any relevant solvers for each abstract lifecycle transformation [17].

Figure 4 shows a *task panel* during the execution of a particular lifecycle, dubbed *ontology_reducer*, which reduces the Ecolingua ontology following the steps of the previously edited formal pattern of Ecolingua's lifecycle. Figure 5 shows how, at a particular stage of the execution of the lifecycle pattern, the broker gives the choice of two alternative solvers with the capability of performing the abstract *weaken* lifecycle step. At this point the user interactively chooses the solver that is to perform the domain-specific task required by the abstract lifecycle step.

2.3. Checking Properties with Lifecycle Histories

Eventually the lifecycle execution terminates, yielding as a response a term that captures the *lifecycle history* of Ecolingua. This lifecycle history can then be used to prove arguments about properties of knowledge components by looking at their evolutionary paths rather than by inspecting the specification of the components themselves. For instance, in the case of ontology transformation, a key question is where in the lifecycle of transformations we begin to lose the soundness of an original general ontology as we refine it for a specific purpose. In Section 4 we explain how *lifecycle histories* allow us to answer this question automatically.

Now that we have motivated our approach we shall describe next its theoretical foundations.

3. Abstract Knowledge Transformation

In order to give an abstract characterisation of knowledge transformation steps we start from the assumption that knowledge transformation presupposes flow of information, and that we need to base any notion of knowledge lifecycle on a sound mathematical theory of information and information flow. There is no such theory of information yet, but there have been several efforts in that direction [7, 1, 6, 2].

In our opinion, channel theory, as proposed by Barwise and Seligman in [2], is currently the most promising approach, as it is inspired by category theory, and we think that this is a major advantage. Category theory stems from the observation that many properties of mathematical systems can be unified and simplified by describing mathematical structures only through structure-preserving relationships between them [14], and this observation provided deep insights in the fields of mathematical logic and computer science. The same approach is taken in channel theory when tackling information: it focuses on the flow of information, i.e., the information transformation, rather than on the highly elusive notion of information itself.

Within this section we outline a formal language for the specification of knowledge lifecycles based on channel theory. It is based on our view of the activity of knowledge artifact synthesis as the connection of different knowledge components —each one with its own laws and regularities described within its own system of ontological commitments— in order to build a *distributed information system* in which the links between these components favour the flow and use of information, and hence of knowledge. Thus, first we need to introduce some basic definitions of channel theory.

3.1. Channel Theory and the Flow of Information

An in-depth discussion of channel theory is impractical here, but a short introduction will suffice. Channel theory has been developed based on the understanding that information flow results from regularities in a distributed system, and that it is by virtue of regularities among the connections that information of some components of the system carries information of other components; furthermore it is the instances that carry information, so that information flow crucially involves both types (i.e., the terminology to describe components) and instances.

Central to channel theory is the idea of a *local logic*. Components of a distributed information system will typically be described using different vocabularies, i.e., using different systems of *types*, and *instances* will be *classified* according to these types in quite different ways. In addition, each component will have its own particular *constraints* that describe its local behaviour. Ideally, all instances will adhere to these constraints, they will be *normal*, although exceptions may occur. A local logic brings all these ideas together:

Definition 1 A *local logic* is a quintuple $\mathcal{L} = (I, T, \models, \vdash, N)$, where

- I is a set of *instances*;
- T is a set of *types*;
- $\models$ is a *classification relation*, a binary relation between elements of I and T;
- $\vdash$ is a *consequence relation*, a binary relation between subsets of T;
- $N$is a subset of I, the set of *normal instances*.

There are two parts of a local logic that are of particular importance in the channel theory framework. The first one is the triple $(I, T, \models)$, and is called the *classification* of the local logic, because the binary relation $\models$ determines a classification of instances in I with respect to types in T. Thus, $i \models t$ means that instance $i \in I$ is classified as of type $t \in T$.

The second important part is the tuple $(T, \vdash)$, which is called the *theory* of the local logic. This theory is specified by a set of *sequents* $\Gamma \vdash \Delta$, where $\Gamma, \Delta \subseteq T$. The set of types Γ is to be interpreted conjunctively, the set Δ disjunctively, so that an instance $i \in I$ *satisfies* a sequent $\Gamma \vdash \Delta$ provided that, if i is of *every* type in Γ, then i is of *some* type in Δ. Sequents that belong to the theory of a logic are called *constraints*. Theories of local logics must satisfy the following properties, and hence are said to be *regular*[2]:

1. Identity: $t \vdash t$, for all $t \in T$;
2. Weakening: If $\Gamma \vdash \Delta$ then $\Gamma, \Gamma' \vdash \Delta, \Delta'$, for all $\Gamma, \Gamma', \Delta, \Delta' \subseteq T$;
3. Global Cut: If $\Gamma, T'_0 \vdash \Delta, T'_1$ for each partition[3] $\langle T'_0, T'_1 \rangle$ of $T' \subseteq T$, then $\Gamma \vdash \Delta$, for all $\Gamma, \Delta \subseteq T$.

Finally, normal instances must satisfy all constraints of the local logic. The idea of normal instances is needed if we want to model reasonable but unsound transformations of information. Hence, we say that a logic is *sound* when all its instances are normal, i.e., $N = I$. A logic is *complete* if every sequent satisfied by every normal instance is a constraint of the logic.

3.2. A Lifecycle Calculus

We describe the lifecycle of knowledge components by means of abstract transformations rules, where the semantics of these rules is to be found in channel theory. On one hand we want these transformation rules to be general, since we attempt to capture only the essentials of any knowledge engineering activity: we don't want the rules to formally describe details of numerous individual techniques. For this reason our rules originate from the abstract operations on local logics provided by channel theory. On the other hand we do not want these rules to be so general that they do not have any link to the technologies we wish to analyse, and in practice we shall always have concrete lifecycles in mind, like the one of Ecolingua. We claim to have achieved a sensible balance for the level of abstraction as the examples of property checking in Section 4.2 illustrate. Nevertheless, we envisage particular specialisations of these general rules that are closer to particular classes of transformation techniques, and hence capture more properties of them. For instance, there will be many transformational components used by knowledge engineers that will not conform to any of the abstract calculus rules below, but are in fact sequences of abstract transformations themselves. Still, investigation into these sort of 'customised' or 'specialised' transformation rules can follow from an abstract description as presented in this paper.

Each of the lifecycle rules introduced below takes one or more abstract knowledge components as input and yields

[2] Regularity arises from the observation that, given a classification of instances to types, the set of all sequents that are satisfied by all instances do fulfill these properties.

[3] A partition of T' is a pair $\langle T'_0, T'_1 \rangle$ of subsets of T', such that $T'_0 \cup T'_1 = T'$ and $T'_0 \cap T'_1 = \emptyset$; T'_0 and T'_1 may themselves be empty (hence it is actually a quasi-partition).

an abstract component as output; each component is characterised as a pair $\langle O, S\rangle$, consisting of the specification S of the component and the ontology O in which the specification is given. We shall model a knowledge component by a *local logic*, where its ontology O specifies the *classification* upon which the local logic will be built, and its specification S determines the *theory* of the logic. Normal instances will depend on the soundness and completeness of the knowledge component. It is beyond the scope of this paper to provide a detailed and formally argued justification of this view of knowledge components; this will be given elsewhere [18]. In the meantime we move on, and distinguish between the following kinds of transformations, based on the above characterisation of knowledge components.

Transformations on the specification

Transforming the specification but not the ontology of a knowledge component will affect the set of constraints of the local logic that characterises it, leaving the classification of instances to types unchanged. Some of these transformations will yield "stronger" specifications, with fewer models, while others will yield "weaker" ones, with more models. Thus, there is a natural partial order on local logics defined as follows:

Definition 2 Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be two local logics on the same classification; $\mathcal{L}_1 \sqsubseteq \mathcal{L}_2$ if and only if

- all constraints of $\mathcal{L}_1$ are also constraints of $\mathcal{L}_2$
- all normal instances of $\mathcal{L}_2$ are also normal instances of $\mathcal{L}_1$

Stronger logics — those larger in the partial order — have more constraints but fewer normal instances. More constraints may result in some instances not satisfying all of the constraints any more, reducing the number of normal instances, while fewer constraints may have the reverse effect. This is a 'counter-variance' that results from the duality between types and instances, and we shall come back to it later when we introduce infomorphisms in Definition 3. We shall distinguish, thus, between two specification transformation rules:

$$\begin{array}{ccc} \text{Specification Strengthening} & \dfrac{\langle O, S\rangle}{\langle O, S'\rangle} & \text{if} \quad S \sqsubseteq S' \\ \text{Specification Weakening} & \dfrac{\langle O, S\rangle}{\langle O, S'\rangle} & \text{if} \quad S \sqsupseteq S' \end{array}$$

In the rules above, the annotations $S \sqsubseteq S'$ and $S \sqsupseteq S'$ should to be understood as that the application of the rules is conditioned to the existence of a partial order between the local logics that characterise specifications S and S'. This partial order shows the *subsumption* of one component by another.

Transformational systems that perform Specification Strengthening are, for instance, systems like ID3 [16] or AQ11 [15], machine learning systems that perform generalisation by rule induction on a set of examples. On the other hand, systems that perform Specification Weakening are, for instance, specialisation operators of knowledge-base refiners [4], or the filters implemented by Brilhante and Robertson for trimming down the Ecolingua ontology, as explained in Section 2.

Transformations on the ontology

There are various ways to transform the ontology of a knowledge component: we may include or remove concepts, modify the concept hierarchy, rename concepts and relations, or modify the set of ontological constraints, for instance. In terms of our semantics based on channel theory, transforming the ontology of a knowledge component will essentially affect the system of classifications, but in a sensible way, namely maintaining the flow of information between ontologies. This latter is captured with the idea of an infomorphism:

Definition 3 A *infomorphism* $f : A \rightleftarrows B$ from classification $A = (I_A, T_A, \models_A)$ to classification $B = (I_B, T_B, \models_B)$ is a contravariant pair of functions $f = \langle f^\star, f_\star\rangle$, where $f^\star : T_A \to T_B$ and $f_\star : I_B \to I_A$, and such that, for $b \in I_B$ and $\alpha \in T_A$,

$$f_\star(b) \models_A \alpha \quad \text{iff} \quad b \models_B f^\star(\alpha) \ .$$

In Section 4.1, we shall pay special attention to infomorphisms whose function of instances is surjective. We say that there is an *ontology morphism* between two ontologies if there is a infomorphism between the classifications that characterise them.

By transforming the ontology of a component we will need to adapt its specification accordingly. That is, we will have to *move* the constraints of the original logic to a new set of instances, types and a classification relation between them. Depending if this is done forwards or backwards along the infomorphism, we will have two ways of moving logics: by taking images and inverse images. We start with the latter first:

Definition 4 Let $f : A \rightleftarrows B$ be an infomorphism, and let $\mathcal{L}$ be a local logic on classification B. The *inverse image* of $\mathcal{L}$ under f, written $f^{-1}[\mathcal{L}]$, is the local logic on classification A whose constraints are defined as follows:

$$\Gamma \vdash_{f^{-1}[\mathcal{L}]} \Delta \quad \text{iff} \quad f[\Gamma] \vdash_{\mathcal{L}} f[\Delta]$$

and whose normal instances are:

$$\{a \in I_A \mid a = f_\star(b) \quad \text{for some } b \in N_{\mathcal{L}}\}$$

In the definition above $f[\Gamma]$ and $f[\Delta]$ are the set-images of the set of types Γ and Δ under function $f^{\star}$. Analogously, we shall use their inverses below, in Definition 5.

Definition 5 Let $f : A \rightleftarrows B$ be an infomorphism, and let $\mathcal{L}$ be a local logic on classification A. The *image* of $\mathcal{L}$ under f, written $f[\mathcal{L}]$, is the local logic on classification B whose constraints are defined as follows:

$$\Gamma \vdash_{f[\mathcal{L}]} \Delta \quad \text{iff} \quad f^{-1}[\Gamma] \vdash_{\mathcal{L}} f^{-1}[\Delta]$$

closed under Identity, Weakening and Global Cut[4], and whose normal instances are:

$$\{b \in I_B \mid f_{\star}(b) \in N_{\mathcal{L}}\}$$

These two ways of moving logics along infomorphisms give rise to two ontology transformation rules:

Ontology Strengthening $\quad \dfrac{\langle O, S\rangle}{\langle O', f[S]\rangle} \quad$ if $\; O \xrightarrow{f} O'$

Ontology Weakening $\quad \dfrac{\langle O, S\rangle}{\langle O', f^{-1}[S]\rangle} \quad$ if $\; O \xleftarrow{f} O'$

In the rules above, the annotations $O \xrightarrow{f} O'$ and $O \xleftarrow{f} O'$ mean that the application of the rules is conditioned to the existence of a ontology morphism between the ontologies, and hence of an infomorphisms between the classification systems of the local logics that characterise the knowledge components involved.

That a knowledge transformation, due to a forward-translation (an image) of the ontology is called Ontology Strengthening is justified later when we discuss ontologies as knowledge components. Analogously, a transformation based on a backward-translation (an inverse image) is Ontology Weakening.

A transformational system that performs both Ontology Strengthening and Ontology Weakening is for example Ontolingua [10], which uses KIF [9] as an expressive *interlingua* into which knowledge components that are to share knowledge are translated back and forth. Another example of a transformational system that does Ontology Weakening is the translator into Horn clauses implemented by Brilhante and Robertson for Ecolingua (see Section 2).

Connection of components

Finally, knowledge components are frequently joined together. If they happen to be specified in the same ontology, only the specifications need to be merged in a sensible way. But in general knowledge components will be specified using different ontologies, and some sort of *bridge* between ontologies will be needed. An abstract way to look at a bridge is as a common ontology O_0 that each of the ontologies of two components conforms to and possibly extends [11]. This is characterised by the existence of ontology morphisms $f_1 : O_0 \rightarrow O_1$ and $f_2 : O_0 \rightarrow O_2$. The *global ontology* that captures the original ontologies is then characterised by a *pushout* construction [14]:

$$\begin{tikzcd}
& \mathbf{O} & \\
\mathbf{O}_1 \arrow[ur, "\iota_1"] & & \mathbf{O}_2 \arrow[ul, "\iota_2"'] \\
& \mathbf{O}_0 \arrow[ul, "f_1"] \arrow[ur, "f_2"'] &
\end{tikzcd}$$

This construction determines the necessary infomorphisms to move the original logics, by taking their respective images into the global ontology O, and then merging them by taking the *join* of logics defined next:

Definition 6 Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be two local logics on the same classification; their *join* $\mathcal{L}_1 \sqcup \mathcal{L}_2$ is the least upper bound in the partial order $\sqsubseteq$ of local logics.

This gives rise to the following abstract transformation rule:

Component Connection $\quad \dfrac{\langle O_1, S_1\rangle \quad \langle O_2, S_2\rangle}{\langle O, \iota_1[S_1] \sqcup \iota_2[S_2]\rangle} \quad$ if $\; O_1 \xleftarrow{f_1} O_0 \xrightarrow{f_2} O_2$

In the rule above, the annotation $O_1 \xleftarrow{f_1} O_0 \xrightarrow{f_2} O_2$ means that the application of the rules is conditioned to the existence of a bridge O_0 for ontologies O_1 and O_2, and ι_1 and ι_2 are the *injections* of the resulting pushout construction with vertex O.[5] The resulting knowledge component is then characterised by the least upper bound of the images of the original logics into the global ontology.

Examples of transformational systems that perform Component Connection are those of standard knowledge engineering practices that link a knowledge base with a problem solver or with ontological axioms.

Ontologies as abstract knowledge components

If we are interested in the lifecycle of transformations performed on a knowledge component $\langle O, S\rangle$ that is actually an ontology, as with Ecolingua's lifecycle discussed in Section 2, then S has to be understood as the content, i.e., the specification, of the ontology, while O is actually its meta-ontology in which the ontology itself is specified.

Now, by looking at the channel-theoretic characterisation of ontologies as classifications, we will discover the

[4] This is necessary as there is no guarantee that the consequence relation as defined by the inverse images of sequents is regular.

[5] Actually this will be a pushout construction on a pair of infomorphisms; this pushout always exists in the category of classifications and infomorphisms [2].

following: If an ontology $O = \langle O_{meta}, S\rangle$, viewed as a component over meta-ontology O_{meta}, is strengthened to an ontology $O' = \langle O_{meta}, S'\rangle$, i.e., $S \sqsubseteq S'$, then there is an ontology morphism $O \xrightarrow{f} O'$. Analogously, if an ontology $O = \langle O_{meta}, S\rangle$ is weakened to an ontology $O' = \langle O_{meta}, S'\rangle$, i.e., $S \sqsupseteq S'$, then there is an ontology morphism $O \xleftarrow{f} O'$:

$$\frac{S \sqsubseteq S'}{\langle O_{meta}, S\rangle \xrightarrow{f} \langle O_{meta}, S'\rangle} \qquad \frac{S \sqsupseteq S'}{\langle O_{meta}, S\rangle \xleftarrow{f} \langle O_{meta}, S'\rangle}$$

Hence, when the ontology Ecolingua is transformed by Ontology Weakening, it is not that Ecolingua, as it is specified by its axioms, is weakened, but its meta-ontology. This happens, for instance, by translating its axioms from KIF, which is essential first-order predicate calculus, into less expressive Horn clauses (that is the last transformation step of the lifecycle depicted in Figure 1). The transformations that clean up and prune Ecolingua are transformations that remove ontological axioms and hence weaken its specification. Consequently they are Specification Weakening steps.

4. Proving Arguments about Properties

With an abstract characterisation of knowledge transformation embodied in the rules of the above lifecycle calculus we are already capable of checking key properties such as soundness and completeness of local logics, and partial order between two local logics. For this we use our understanding of channel-theoretic operations like, for instance, those of moving logics, and that form the semantics of the rules of our lifecycle calculus, under the assumption that the specification supplied for each individual transformation step is correct.

4.1. About Moving Logics

Recall from Section 3.2 that by strengthening or weakening the ontology of a knowledge component we are actually *moving* the local logic that characterises the component along an infomorphism. Images and inverse images of logics have an effect on the preservation and loss of properties such as the soundness of the completeness of a local logic:

Proposition 1 ([2]) *Let $f : A \rightleftarrows B$ be an infomorphism.*

1. *Let $\mathcal{L}$ be a local logic on a classification A.*
   - *If $\mathcal{L}$ is sound, then $f[\mathcal{L}]$ is sound.*
   - *If f is surjective on instances and $\mathcal{L}$ is complete, then $f[\mathcal{L}]$ is complete.*
2. *Let $\mathcal{L}$ be a local logic on a classification B.*
   - *If $\mathcal{L}$ is complete, then $f^{-1}[\mathcal{L}]$ is complete.*
   - *If f is surjective on instances and $\mathcal{L}$ is sound, then $f^{-1}[\mathcal{L}]$ is sound.*

Consequently, strengthening an ontology preserves the soundness of a knowledge component, but not its completeness, and weakening an ontology preserves the completeness of the component, but not its soundness. Strengthening would preserve completeness, and weakening soundness, were the infomorphism surjective on instances.

It is also interesting to know how moving logics influences the partial-order relation between logics and the join of logics:

Proposition 2 ([2]) *Let $f : A \rightleftarrows B$ be an infomorphism.*

1. *For logics $\mathcal{L}_1$ and $\mathcal{L}_2$ on A,*
   (a) *if $\mathcal{L}_1 \sqsubseteq \mathcal{L}_2$ then $f[\mathcal{L}_1] \sqsubseteq f[\mathcal{L}_2]$;*
   (b) *$f[\mathcal{L}_1 \sqcup \mathcal{L}_2] = f[\mathcal{L}_1] \sqcup f[\mathcal{L}_2]$.*
2. *For logics $\mathcal{L}_1$ and $\mathcal{L}_2$ on B,*
   (a) *if $\mathcal{L}_1 \sqsubseteq \mathcal{L}_2$ then $f^{-1}[\mathcal{L}_1] \sqsubseteq f^{-1}[\mathcal{L}_2]$;*
   (b) *$f^{-1}[\mathcal{L}_1 \sqcup \mathcal{L}_2] = f^{-1}[\mathcal{L}_1] \sqcup f^{-1}[\mathcal{L}_2]$.*

Consequently, strengthening or weakening the ontology of two knowledge components preserves the partial-order relation between two components as well as the the joins, i.e., when components are connected together.

Finally, it is also interesting to know what happens to a knowledge component when it is translated to and from a weaker or stronger ontology. The resulting component will itself be weaker or stronger due to the following proposition:

Proposition 3 ([2]) *Let $f : A \rightleftarrows B$ be an infomorphism.*

1. *For any logic $\mathcal{L}$ on A, $\mathcal{L} \sqsubseteq f^{-1}[f[\mathcal{L}]]$.*
2. *For any logic $\mathcal{L}$ on B, $f[f^{-1}[\mathcal{L}]] \sqsubseteq \mathcal{L}$.*

4.2. Inspecting Lifecycle Histories

Although we are dealing with a small set of transformation rules in our lifecycle calculus, and in addition these are quite abstract, we are already able to prove some key properties that are useful to the task of a knowledge engineer. For example, suppose we want to combine two knowledge components C_1 and C_2 which are both specified over the same ontology O, and we lack a transformational system capable of combining knowledge components specified in O, but instead we have at our disposal:

- a 'translator' that can translate specifications between an ontology O and a more expressive ontology O', in either direction (e.g., the system Ontolingua and the interlingua KIF); and

- a ‘merger’ that combines components expressed in the more expressive ontology O'.

We may decide to use the translator to translate components C_1 and C_2 into ontology O', then use the merger to combine them, and finally translate the resulting component back into O using the translator again. Can we know if our resulting component subsumes both C_1 and C_2?

Of course, we could decide to supply a theorem prover with the specifications of C_1, C_2 and C and try proving if the axioms of the former follow from the axioms of the latter. But this is a very resource-expensive task and will be quite often impracticable when dealing with large-scale knowledge components. Instead, we can inspect the sequence of abstract transformations we have been applying to C_1 and C_2 in terms of the lifecycle calculus proposed in Section 3.2, and use the propositions of channel theory given above to perform the same check, hence staying at the abstract ‘transformation level’ only, and not getting involved with component specifications at all. This ‘transformation level’ is characterised by the histories of knowledge components as they are transformed during their lifecycles. We describe them by means of terms and the lifecycle calculus.

Definition 7 A *lifecycle history* is a term, constructed as follows:

1. A component identifier C is itself a lifecycle history (the empty history).

2. If h, k are lifecycle histories, f, g are ontology morphisms, and O_0 is a bridge, then $\mathsf{ss}(h)$, $\mathsf{sw}(h)$, $\mathsf{os}(f, h)$, $\mathsf{ow}(f, h)$, and $\mathsf{cc}(O_0, f, g, h, k)$ are lifecycle histories.

In the definition above ss, sw, os, ow, and cc stand for applications of the five calculus rules given in Section 3.2 (taking the obvious correspondences). Where a calculus rule involves ontology morphisms or a bridge of ontologies, the morphisms and the bridge are explicitly stated in the term. So, $\mathsf{cc}(O_0, f_1, f_2, C_1, C_2)$ would represent a component connection involving components C_1 and C_2 after they have been expressed in a global ontology determined by the bridge O_0 and ontology morphisms f_1 and f_2. We shall use id to represent the identity ontology morphism, when no translation is needed.

The lifecycle history for our component C of the example above is, thus,

$$C = \mathsf{ow}(f, \mathsf{cc}(O', id, id, \mathsf{os}(f, C_1), \mathsf{os}(f, C_2)))$$

In terms of its semantics in channel theory, if $\mathcal{L}_1$ and $\mathcal{L}_2$ are the local logics characterising knowledge components C_1 and C_2, respectively, and if f is the infomorphism characterising the information flow from O to O', then the local logic for the resulting component C will be $f^{-1}[f[\mathcal{L}_1] \sqcup f[\mathcal{L}_2]]$. The subsumption of component C_1 by component C, for instance, is proved as follows: By Proposition 3,

$$\mathcal{L}_1 \sqsubseteq f^{-1}[f[\mathcal{L}_1]]$$

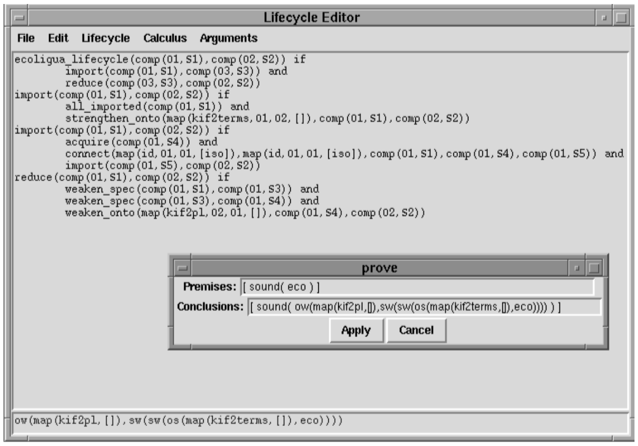
and, by Definitions 2 and 6,

$$f^{-1}[f[\mathcal{L}_1]] \sqsubseteq f^{-1}[f[\mathcal{L}_1]] \sqcup f^{-1}[f[\mathcal{L}_2]] \ .$$

Finally, by Proposition 2,

$$f^{-1}[f[\mathcal{L}_1]] \sqcup f^{-1}[f[\mathcal{L}_2]] = f^{-1}[f[\mathcal{L}_1] \sqcup f[\mathcal{L}_2]] \ .$$

By Proposition 1, completeness and soundness of the resulting combined component cannot be guaranteed, unless we know that the ontology morphisms involved — the infomorphisms – are surjective on the instances (that populate our ontologies).



```
ecoligua_lifecycle(comp(01,S1),comp(02,S2)) if
        import(comp(01,S1),comp(03,S3)) and
        reduce(comp(03,S3),comp(02,S2))
import(comp(01,S1),comp(02,S2)) if
        all_imported(comp(01,S1)) and
        strengthen_onto(map(kif2terms,01,02,[]),comp(01,S1),comp(02,S2))
import(comp(01,S1),comp(02,S2)) if
        acquire(comp(01,S4)) and
        connect(map(id,01,01,[iso]),map(id,01,01,[iso]),comp(01,S1),comp(01,S4),comp(01,S5)) and
        import(comp(01,S5),comp(02,S2))
reduce(comp(01,S1),comp(02,S2)) if
        weaken_spec(comp(01,S1),comp(01,S3)) and
        weaken_spec(comp(01,S3),comp(01,S4)) and
        weaken_onto(map(kif2pl,02,01,[]),comp(01,S4),comp(02,S2))
```

Premises: [sound(eco)]

Conclusions: [sound(ow(map(kif2pl,[]),sw(sw(os(map(kif2terms,[]),eco)))))]

```
ow(map(kif2pl,[]),sw(sw(os(map(kif2terms,[]),eco))))
```

Figure 6. Proving arguments on Ecolingua’s lifecycle

4.3. Back to Ecolingua

Figure 6 shows our *lifecycle editor* of Section 2 again (compare with Figure 3), but at the moment where the knowledge engineer is about to ask for the validity of an argument concerning the soundness of Ecolingua. In particular, the engineer attempts to prove if, assuming that the version of the ontology *before* the sequence of transformations — identified here with the constant eco — is sound, we can conclude that the ontology *after* the transformation steps — identified by the lifecycle history displayed in the bottom part of the window — is also sound. The lifecycle history displayed in this window corresponds to the execution of the formal lifecycle pattern as it was represented with Horn clauses, and captures the structure of Ecolingua’s lifecycle as illustrated in Figure 1. In this particular case, the argument we attempt to prove will be shown to be invalid.

Thus, if soundness is a property to be preserved, any further transformations that are to be applied to Ecolingua need to take the output of the pruning transformation step (the last Specification Weakening step) since after Ontology Weakening, which translates axioms into Horn clauses, soundness cannot be guaranteed. Here is something that would be impossible to know if we had not a lifecycle history that recorded the sort of transformations applied to Ecolingua. It is only practical to learn it through use of lifecycle histories and these must be supported through automated synthesis in order to deal with the scaling problem of recording and reasoning about such information in large, open systems.

5. Conclusion

Although in this paper we have been mainly concerned with the task of large-scale knowledge engineering, rather than with the automated generation of code, we have tackled a problem that is of great importance to software engineers, namely the task of combining transformational systems and providing automated support for performing this task in large-scale and open environments such as the World-Wide Web. We have introduced a semantically precise language to describe knowledge transformations in abstract terms, and showed how we can automatically check for the validity of arguments concerning key properties of knowledge components along different stages of the components' lifecycles. We have also given a glimpse into an agent-based architecture in which we can use a customised lifecycle editor to analyse and execute sequences of abstract transformations in domain-specific ways. We plan to further study specialised environments based on the ideas presented in this paper, so that we can provide additional automated support to the task of large-scale knowledge and software engineering.

Acknowledgements

This work is supported under the Advanced Knowledge Technologies (AKT) Interdisciplinary Research Collaboration (IRC), which is sponsored by the UK Engineering and Physical Sciences Research Council under grant number GR/N15764/01. The AKT IRC comprises the Universities of Aberdeen, Edinburgh, Sheffield, Southampton and the Open University.

References

[1] J. Barwise and J. Perry, editors. *Situations and Attitudes*. MIT Press, 1983.

[2] J. Barwise and J. Seligman. *Information Flow: The Logic of Distributed Systems*. Cambridge University Press, 1997.

[3] V. Brilhante and D. Robertson. Metadata-supported automated ecological modelling. In C. Rautenstrauch and S. Patig, editors, *Environmental Information Systems in Industry and Public Administration*. Idea Group Publishing, 2001.

[4] L. Carbonara and D. H. Sleeman. Effective and efficient knowledge base refinement. *Machine Learning*, 37:143–181, 1999.

[5] F. S. Corrêa da Sliva, W. W. Vasconcelos, D. S. Robertson, V. Brilhante, A. C. de Melo, M. Finger, and J. Agustí. On the insufficiency of ontologies: problems in knowledge sharing and alternative solutions. *Knowledge-Based Systems*, 15(3):147–167, 2002.

[6] K. Devlin. *Logic and Information*. Cambridge University Press, 1991.

[7] F. Dretske. *Logic and the Flow of Information*. MIT Press, 1981.

[8] A. Farquhar, R. Fikes, and J. Rice. The Ontolingua Server: a tool for collaborative ontology construction. *International Journal of Human-Computer Studies*, 46(6):707–727, 1997.

[9] M. R. Genesereth and R. E. Fikes. Knowledge interchange format (KIF). Draft proposed American National Standard, NCITS.T2/98-004, 1998. Available at http://logic.stanford.edu/kif/dpan.html.

[10] T. Gruber. A translation approach for portable ontology specifications. *Knowledge Engineering*, 5(2), 1993.

[11] R. E. Kent. The information flow foundation for conceptual knowledge organization. In *Sixth International Conference of the International Society for Knowledge Organization*, 2000.

[12] M. Lowry and J. V. Baalen. Meta-Amphion: Synthesis of efficient domain-specific program synthesis systems. *Automated Software Engineering*, 4, 1997.

[13] M. Lowry, A. Philpot, T. Pressburger, and I. Underwood. A formal approach to domain-oriented software design environments. In *Proceedings of the 9th Knowledge-Based Software Engineering Conference*, pages 48–57, 1994.

[14] S. Mac Lane. *Categories for the Working Mathematician*. Springer, second edition, 1998.

[15] R. S. Michalski and J. Larson. Incremental generation of VL_1 hypotheses: the underlying methodology and the description of program AQ11. Technical Report ISG 83-5, Department of Computer Science, University of Illinos at Urbana-Champaign, 1983.

[16] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[17] W. M. Schorlemmer, S. Potter, D. Robertson, and D. Sleeman. Formal knowledge engineering in distributed environments. In *ECAI 2002 Workshop on Knowledge Transformation for the Semantic Web*, 2002.

[18] W. M. Schorlemmer, D. Robertson, and S. Potter. Formal knowledge lifecycles. Technical report, Division of Informatics, The University of Edinburgh, 2002.