Knowledge Life-Cycle Management over a Distributed Architecture

Marco Schorlemmer[1] Stephen Potter[1] David Robertson[1] Derek Sleeman[2]

[1]Centre for Intelligent Systems and their Applications
School of Informatics
The University of Edinburgh

[2]Department of Computing Science
University of Aberdeen

Abstract

In order to address problems stemming from the dynamic nature of distributed systems, there is a need to be able to express the often neglected notions of the evolution and change of the knowledge components of such systems. This need becomes more pressing when one considers the potential of the Internet for distributed knowledge-based problem solving — and the pragmatic issues surrounding knowledge integrity.

In this paper, we introduce a formal calculus for describing transformations in the 'life cycles' of knowledge components, along with ideas about the nature of distributed environments in which the ideas underpinning the calculus can be realised. The formality and level of abstraction of this language encourage the analysis of knowledge histories and allows useful properties about this knowledge to be inferred. These ideas are illustrated through the discussion of a particular case-study in knowledge evolution.

1 Introduction

The dynamic nature of knowledge has long been realised: knowledge evolves over time as experiences accumulate; it is revised and augmented in the light of deeper comprehension; entirely new bodies of knowledge are created while at the same time others pass into obsolescence. However, this dynamism has rarely been acknowledged in the engineering of knowledge-based systems: systems of static knowledge elements are assumed to exist in unchanging environments. The increasing desire to deliver knowledge across distributed environments such as the World-Wide Web highlights the extent of this gulf: the Web is ever-changing and systems must be able to accommodate change if they are to succeed and thrive.

To be able to describe the dynamics of knowledge, there seems to be a need for some level of formality. The role of formality is twofold: to give a concise account of what is going on, and to use this account for practical purposes in maintaining and analysing the 'life cycles' of knowledge components from their creation, through successive phases of modification, to their eventual retirement.

We start in Section 2 with a glimpse into a prototype environment we are currently implementing. Its constituent agents can interact and use their combined capabilities to solve problems, and it allows the life cycles of these agents to be described and maintained. Section 3 outlines the essential characteristics of such an environment, although consideration of the precise forms which these life-cycle descriptions might take is deferred until Section 4, where the notion of a *life-cycle calculus* of abstract knowledge transformations is introduced. These abstractions allow meaningful statements to be made about bodies of knowledge without sacrificing generality to the demands of any particular domain. By making explicit the histories of these knowledge components using terms based on this calculus, the sources of knowledge can be identified, assumptions traced, and useful properties inferred. To better illustrate these ideas, a case-study of the life cycle of one particular ontology is used. In Section 5 we return to this case-study

and describe how it might be enacted by the agents in our environment. In doing this, in Section 6 we make more concrete the ideas of formality for knowledge maintenance that lie at the heart of this research.

2 Ecolingua's Life Cycle

We shall motivate our research by using a real example — the life cycle of the Ecolingua ontology — and give a snapshot of the kind of environment we are currently developing so that a knowledge engineer can interact with the life cycle. In subsequent sections we shall discuss the details of such environment.

Ecolingua is an ontology engineered by Brilhante and Robertson for the description of ecological data [2, 4]. In order to reuse some of the ontologies made available by the Ontolingua Server [5] the new ontology 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 by the server's translation service (see Figure 1).

Because the outcome of the translation process was a large 5.3 Mb file, it was necessary to reduce the ontology in order to get a smaller and more manageable set of axioms. The definitions of proper Ecolingua classes use external classes defined in five other ontologies, and the translation service of the server just joins all ontologies together before performing the translation. This forced Brilhante and Robertson to implement filters 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. Finally, since the ontology was specified with KIF expressions (although in Prolog syntax) a final translation into Horn clauses was performed in order to execute the ontology with a Prolog interpreter, the preferred language of the authors of the ontology. Figure 1 shows this part of Ecolingua's life cycle.

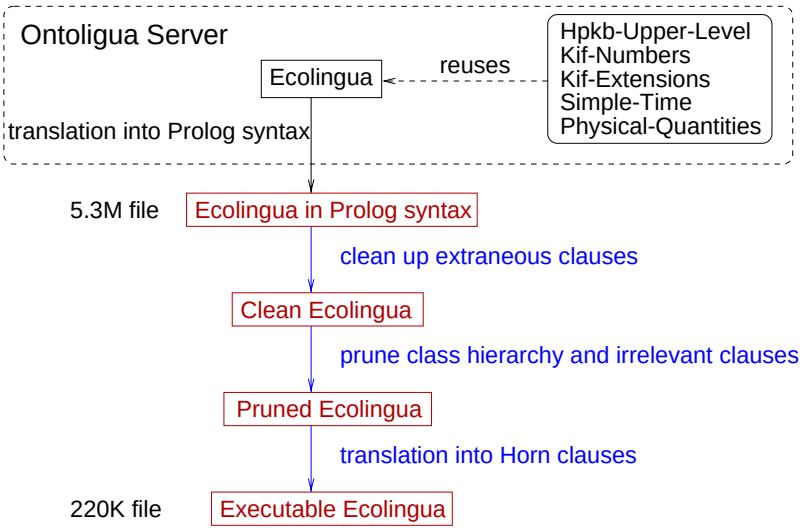

Figure 1: Ecolingua's life cycle

We postulate that particular sequences of life-cycle steps like those illustrated in Figure 1 might be common in particular domains and perhaps to particular forms of knowledge component. As such, the ability to generalise and 'compile' these sequences transforming them into *life-cycle patterns* and making them available within the environment as integrated competences would encourage more efficient behaviour when faced with the need to make similar modifications in the future.

Figure 2 shows a *life-cycle editor* that enables a knowledge engineer to analyse the life cycle of a knowledge component, extract its abstract pattern, and devise a formal representation of it. In particular, it shows Ecolingua's life cycle at the stage where the engineer is determining that the last transformation step is a *weakening* step of the ontology. The definition and choice of abstract life-cycle steps (such as *weakening*) is justified by a formal life-cycle calculus that we shall introduce later in Section 4 (see

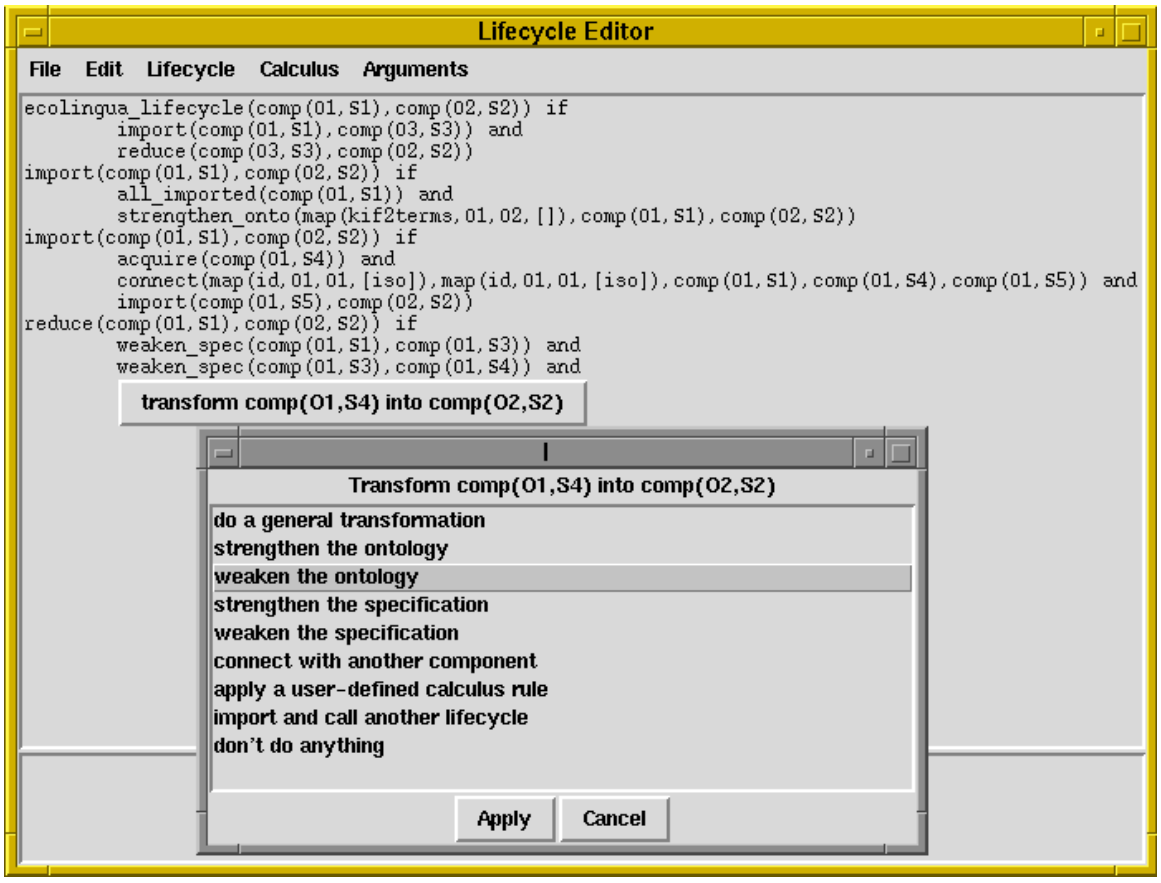

Figure 2: Editing Ecolingua's life cycle

also Figure 7, which gives the formal counterpart of this life cycle). Notice that, although the depicted life-cycle editor uses an explicitly formal notation, it is possible to hide much of the formality by using domain/task-specific editing operations.

Once the life cycle of a knowledge component like Ecolingua is formalised, we may publish it as a competence of a *life-cycle interpreter*, which is a meta-interpreter capable of executing the formal representation of a life cycle. This life cycle is described at a generic level; in order to be able to run the life cycle in a particular domain, the execution is carried out by means of a brokering service that searches for problem solvers capable of performing abstract life-cycle steps; these solvers should have previously advertised their capabilities. In this example we have decided to make the broker interactive to enable the knowledge engineer to choose among several alternative solvers that have the same capability to carry out particular life-cycle steps in a domain-specific fashion. One might, though, decide to have a non-interactive broker that has the sufficient information (or is able to acquire it) for choosing among alternative solvers. In Sections 3 and 5 we further explain how we have done this in a distributed environment.

Figure 3 shows a *task panel* during the execution of a particular life cycle, *ontology_reducer*, which reduces the Ecolingua ontology following the steps of the previously edited formal pattern of Ecolingua's life cycle. Figure 4 shows how, at a particular stage of the execution of the life-cycle pattern, the broker gives the choice of two alternative solvers with the capability of performing the *weaken* life-cycle step. At this point the user interactively chooses the solver to perform the domain-specific task required by the abstract life-cycle step.

Eventually the life-cycle execution ends, yielding as a response a term that expresses the *life-cycle history* of the knowledge component (Ecolingua in this case). This life-cycle history can later be used

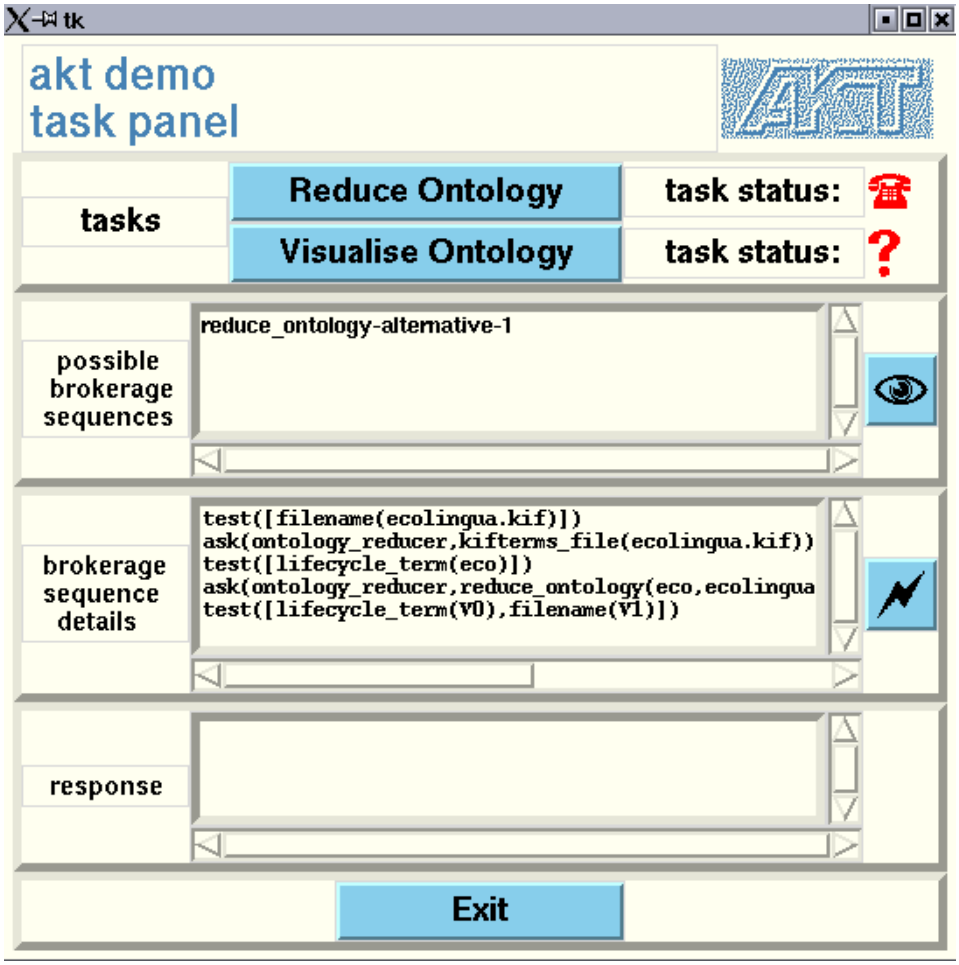

Figure 3: Executing Ecolingua's life cycle

to infer properties of the components by looking at its evolutionary path rather than by inspecting the specification of the component itself. In the next section we give a fuller description of an environment that facilitates the sort of knowledge management we have briefly illustrated so far.

3 A Knowledge Life-Cycle Management Environment

We shall now describe a generalised system in which formal notions of knowledge life cycles can be exploited to effect a coherent knowledge management environment. These life cycles correspond to sequences of transformations described by a formal calculus; however, discussion of the precise form of this calculus is deferred until Section 4 so as to emphasise that this environment is not predicated upon any particular life-cycle language.

We envisage a distributed agent-based environment which mirrors the most general structures of knowledge-intensive problem solving at both micro- and macro-levels (including the Internet). In general, a typical agent will contain:

- some knowledge component, along with a *life-cycle history* that describes the evolution of that component from (at least) the inception of the system: it is a requirement of the system that all agents store — and make available for inspection — the corresponding life-cycle history alongside their knowledge components.

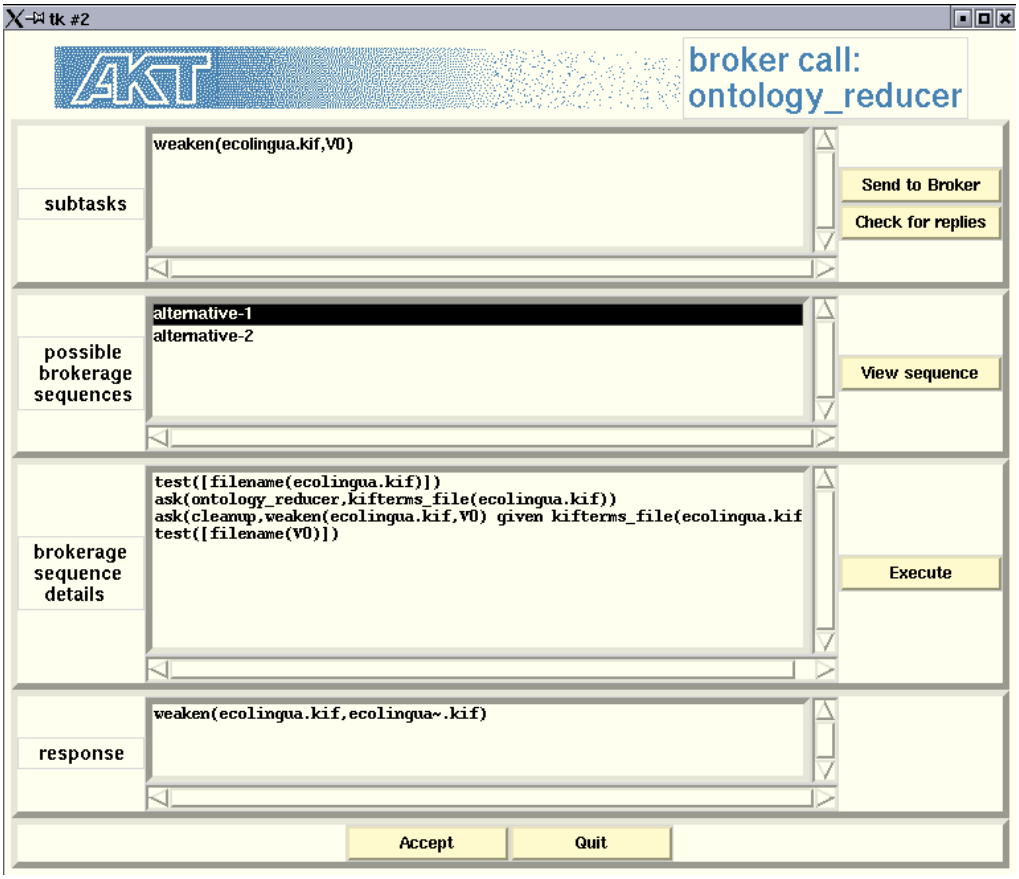

Figure 4: Choosing a particular solver for an abstract life-cycle step

- the means to communicate with its environment, sending messages addressed to other agents, and receiving messages addressed to it.
- the means to invoke its knowledge, as appropriate, for problem solving purposes.
- a succinct expression of the capabilities of its knowledge (the agent's *competences*), including any assumptions it makes and any requirements and constraints it imposes in the exploitation of that knowledge. An agent's competences are advertised as a statement of the services it is both willing and able to provide, under the stipulated conditions.

Given a common communication language with agreed semantics, some notion of how each agent may connect to the system and an appropriate mechanism for co-ordinating the competences of individual agents, a system of this sort would be able to solve problems that lie within its global competence. (Here we assume that this mechanism would be in the form of some sort of centralised brokering service, similar to that seen in the previous section, but more distributed models of co-operation could play this co-ordination role equally well.) In this manner, the system as a whole would be able to act as a conventional 'knowledge-based system', exploiting its collective knowledge-base to act intelligently in the face of (both internal and external) appeals to that knowledge. However, the availability of the formal life-cycle histories that accompany the knowledge components of the system adds an extra facet to this process and to the environment as a whole. We anticipate that the histories of the individual knowledge components can be inspected and reasoned with. Subsequently, properties of these components and of the system in its entirety can be proved and used to form a more complete picture of the state of the system, which enables the more considered exploitation of the resources it offers.

Now consider this system to contain additional agents that have some very particular properties (while still conforming to the general description of agents given above):

knowledge transformers: These are agents whose competences include the ability to perform some transformation on a knowledge component, like cleaning up extraneous clauses or pruning the class hierarchy in the Ecolingua example of Section 2 — they have the ability to perform one or more *life-cycle steps* on these components. It is a requirement of the system that all its knowledge transformers agree to act in a principled manner: knowledge components must be altered in accordance with the calculus, and it is imperative that the knowledge transformer also modifies appropriately the current life-cycle history associated with that component.

life-cycle interpreters: These are agents capable of following the sequence of generalised life-cycle steps specified in some predefined *life-cycle pattern*, like the one shown in Figure 7. Although they are able to follow the sequence, in general they will not be able to alter knowledge components. For that reason they will have to ask for the service of knowledge transformers that are capable of performing such transformations.

Should a life-cycle interpreter itself be capable of executing one or more life-cycle steps, then it must also be considered a knowledge transformer (with all the constraints on its behaviour this entails).

As mentioned in Section 2, we claim that particular sequences of transformations applied to certain forms of knowledge components might be common to specific domains. Generalising and compiling these sequences into 'life-cycle patterns' and making them available within the environment as the competence of agents that can 'execute' these patterns would allow for the management of life cycles when faced with the need to make similar transformations in the future. The role of the life-cycle interpreter is to offer one or more of these life-cycle patterns to the system.

The knowledge components of other agents — including those of other knowledge transformers — can be altered and recorded in a formalised manner using knowledge transformers. As a result, the system is able to coordinate and control the evolution of its knowledge, enabling its global competences to evolve and develop in accordance with the changing demands of its problem-solving domain. The steps in the development of the various knowledge components would be traceable from their inception to their final retirement from the system. Users of the services provided by the system would be fully aware of the provenance of that service, and able to make informed judgements about its reliability, inherent assumptions and trustworthiness.

4 Formal Knowledge Life Cycles

In Section 5 we describe a concrete system of this kind, but first we need to introduce the necessary formal machinery to capture and represent the abstract structure of knowledge life cycles.

4.1 Life-Cycle Calculus

We capture the structure of the life cycle through which a knowledge component like Ecolingua has gone by means of abstract transformations drawn from the set of life-cycle calculus rules given in Figure 5. We wanted these rules to be general, since we attempt to capture only the essentials of the knowledge engineering activity and do not want to describe formally the details of numerous individual techniques. On the other hand, we wanted them not to be so general that they do not have any relevance to the technologies we wish to analyse. We claim to have achieved a sensible level of abstraction as the example of distributed life-cycle management described in Section 5 illustrates. Nevertheless we envisage particular specialisations of this general calculus that are closer to particular classes of transformation techniques and, subsequently, capture more properties of these techniques.

The rules of the calculus are to be interpreted as particular kinds of 'inference rules' with premises (or input components) above the horizontal line, and conclusions (or output components) below it. The life-cycle rules operate on abstract knowledge components; 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. If the knowledge component currently under consideration is itself an ontology, as it is the case

Ontology Strengthening (OS)	$\frac{\langle O, S\rangle}{\langle O', f[S]\rangle}$	if $O \xrightarrow{f} O'$
Ontology Weakening (OW)	$\frac{\langle O, S\rangle}{\langle O', f^{-1}[S]\rangle}$	if $O \xleftarrow{f} O'$
Specification Strengthening (SS)	$\frac{\langle O, S\rangle}{\langle O, S'\rangle}$	if $S \sqsubseteq S'$
Specification Weakening (SW)	$\frac{\langle O, S\rangle}{\langle O, S'\rangle}$	if $S \sqsupseteq S'$
Component Connection (CC)	$\frac{\langle O, S\rangle \quad \langle O', S'\rangle}{\langle O'', f[S] \sqcup g[S']\rangle}$	if $O \xrightarrow{f} O'' \xleftarrow{g} O'$

Figure 5: Life-cycle calculus

of Ecolingua, then S is the content of the ontology, while O is actually the meta-ontology in which the ontology is specified.

The application of these life-cycle rules is further conditioned by either the existence of an order $\sqsubseteq$ between specifications (as in the rules of *Specification Strengthening/Weakening*) or of particular mappings $O \xrightarrow{f} O'$ between ontologies (as in the rules of *Ontology Strengthening/Weakening* and *Component Connection*).

Ontology Strengthening captures, for instance, transformations that add ontological constraints or that translate the component into a more expressive logical language, and *Ontology Weakening* captures transformations that remove ontological constraints or translate the component into a less expressive logical language. In these cases, the specification is changed according to the change of ontology, which is denoted with $f[S]$ and $f^{-1}[S]$, the set image and inverse set image of a specification with respect to an ontology mapping, respectively. The mathematical principles upon which the calculus is founded have been discussed in [6] and are beyond the scope of this paper. They are based on a mathematical theory of information flow proposed by Barwise and Seligman in [1].

Specification Strengthening captures, for example, transformations that add axioms to the component's specification or that generalise it, and *Specification Weakening* captures transformations that remove axioms from the component's specification or that specialise it. This is modelled with an order relation $\sqsubseteq$ on specifications. Finally, *Component Connection* captures transformations that take two knowledge components (perhaps expressed using different ontologies O and O') and merge them into a unified component, whenever their respective ontologies can be mapped into a common global ontology O''. Here the join operator $\sqcup$ models the merging operation, and is usually defined as a least upper bound in the order $\sqsubseteq$ of specifications.

Figure 6 shows the application of several life-cycle calculus rules as they occurred in Ecolingua's life cycle (Figure 1) in form of an inference tree. In the tree, O_{ol}, O_{kif}, O_{pl} denote the ontologies for Ontolingua, KIF terms, and Prolog clauses, respectively; S is the proper specification of Ecolingua, while $S_1, \ldots, S_5$ are the specifications of the ontologies that are reused.

During the first steps, several *Component Connection* rules are applied until an *Ontology Strengthening* step is performed to get the whole specification written with KIF terms in Prolog syntax. These are the steps that were performed at the Ontolingua Server. They are justified by the fact that all ontologies are written in Ontolingua, hence no translation (or only identity (id) translations) had to be done before the specifications can be merged. The translation into KIF terms is justified by the existence of a translator, that we abstractly denote as $O_{ol} \xrightarrow{ol2kif} O_{kif}$.

The final three steps in the tree — two applications of *Specification Weakening* plus one application of *Ontology Weakening* — were performed afterwards in order to clean up, prune, and finally translate

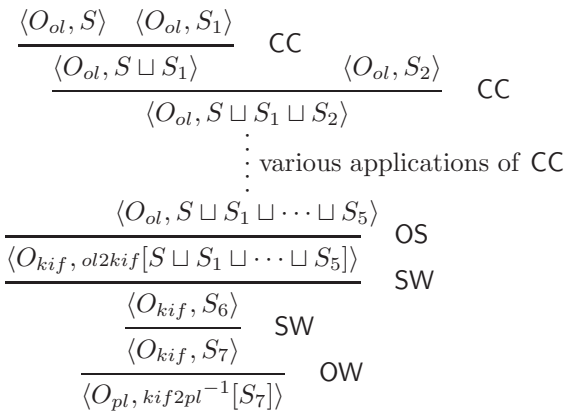

Figure 6: Inference tree for Ecolingua's life cycle

the component into Prolog clauses. Here, S_6 and S_7 denote the specification of our component after the clean-up and pruning transformations, respectively (i.e., $ol2kif[S \sqcup \cdots \sqcup S_5] \sqsupseteq S_6 \sqsupseteq S_7$). The translation into Prolog clauses is justified by the existence of a translator, that we abstractly denote as $O_{kif} \overset{kif2pl}{\longleftarrow} O_{pl}$. (The fact that the arrow is represented in the opposite direction of the translation and also that its inverse is used in the inference tree is justified by the underlying formal semantics of the life-cycle calculus, which lies outside the scope of this paper.)

4.2 Life-Cycle Patterns

We have implemented a prototype tool that supports the generation of patterns of life cycles based on the life-cycle calculus discussed in Section 4.1. Figure 7 presents a set of Horn clauses corresponding to the life cycle of Ecolingua discussed in Section 2 (see Figure 1 and the inference tree in Figure 6); this is the life cycle shown in the editor in Figure 2.

$$
\begin{aligned}
&\mathit{ecolingua\_lifecycle}(\langle O_1, S_1\rangle, \langle O_2, S_2\rangle) \longleftarrow \\
&\qquad \mathit{import}(\langle O_1, S_1\rangle, \langle O_3, S_3\rangle) \wedge \\
&\qquad \mathit{reduce}(\langle O_3, S_3\rangle, \langle O_2, S_2\rangle)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{import}(\langle O_1, S_1\rangle, \langle O_2, S_2\rangle) \longleftarrow \\
&\qquad \mathit{all\_imported}(\langle O_1, S_1\rangle) \wedge \\
&\qquad \mathit{strengthen\_onto}(O_1 \overset{ol2kif}{\rightarrow} O_2, \langle O_1, S_1\rangle, \langle O_2, S_2\rangle)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{import}(\langle O_1, S_1\rangle, \langle O_2, S_2\rangle) \longleftarrow \\
&\qquad \mathit{acquire}(\langle O_1, S_4\rangle) \wedge \\
&\qquad \mathit{connect}(O_1 \overset{id}{\rightarrow} O_1, O_1 \overset{id}{\rightarrow} O_1, \langle O_1, S_1\rangle, \langle O_1, S_4\rangle, \langle O_1, S_5\rangle) \wedge \\
&\qquad \mathit{import}(\langle O_1, S_5\rangle, \langle O_2, S_2\rangle)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{reduce}(\langle O_1, S_1\rangle, \langle O_2, S_2\rangle) \longleftarrow \\
&\qquad \mathit{weaken\_spec}(\langle O_1, S_1\rangle, \langle O_1, S_3\rangle) \wedge \\
&\qquad \mathit{weaken\_spec}(\langle O_1, S_3\rangle, \langle O_1, S_4\rangle) \wedge \\
&\qquad \mathit{weaken\_onto}(O_1 \overset{kif2pl}{\leftarrow} O_2, \langle O_1, S_4\rangle, \langle O_2, S_2\rangle)
\end{aligned}
$$

Figure 7: Formal representation of Ecolingua's life cycle as a life-cycle pattern

The recursive *import* predicate captures that part of the life cycle Ecolingua underwent in the Ontolingua Server and corresponds to the sequence of applications of *Component Connection* and the application

of *Ontology Strengthening* illustrated in the inference tree of Figure 6. At each call of *import* a new component is acquired and subsequently connected to our current component, until some termination condition is satisfied. The *reduce* predicate captures the subsequent reduction performed on the output file of the server and corresponds to the two applications of *Specification Weakening* and the application of *Ontology Weakening* illustrated in the inference tree of Figure 6.

Focusing on the *reduce* predicate, for instance, the two goals *weaken_spec* represent the clean-up and pruning steps the ontology went through and correspond to particular applications of the *Specification Weakening* rule of the calculus. It takes knowledge component $\langle O_1, S_1\rangle$ and delivers component $\langle O_1, S_4\rangle$, where S_4 is going to be a weaker specification than S_1, according to the calculus rule, since, after cleaning up and pruning, the ontology will have less constraints. The *weaken_onto* goal captures the final translation into Prolog clauses and corresponds to a particular application of *Ontology Weakening*. It takes knowledge component $\langle O_1, S_4\rangle$ of the previous goal and delivers component $\langle O_2, S_2\rangle$, where O_2 is the new meta-ontology (in this particular example it is that of Prolog clauses) while S_2 is the translation of S_4 into the new syntax. Since Horn-clause logic is a less expressive logical language than KIF, which is essentially predicate calculus, the output component $\langle O_2, S_2\rangle$ will be in principle weaker than component $\langle O_1, S_4\rangle$.

We have chosen a Horn-clause representation of knowledge life cycles to readily implement a meta-interpreter that allows the execution of a life cycle, so that we can recreate the same pattern in different knowledge-engineering scenarios. The life-cycle calculus, however, is independent of the enactment system in which it may be embedded. We consider that, when acquiring ontologies from sources such as the Ontolingua Server in the future, it is quite likely that we will have to perform similar importing and reducing operations, and so it is worthwhile formalising these steps and publishing them as a generic life-cycle pattern.

4.3 Managing Formal Life Cycles

Having the essentials of the knowledge-engineering activity expressed as formal life-cycle patterns allows the study and analysis of life cycles as first-class citizens: we may consider life cycles as knowledge components themselves and reuse them in other knowledge management activities. This allows us to devise frameworks — like the one envisaged in Section 3 and further developed in Section 5 — in which we can keep track of the several life-cycle transformations a knowledge component goes through, and to infer properties that are preserved during different stages of a life cycle.

For instance, transforming a knowledge component by weakening its specification preserves its soundness but not the completeness of the logic that models the component. On the other hand weakening the ontology of a knowledge component by representing it in a less expressive logical language preserves the completeness but not the soundness of the logic that models the component. These statements are justified by the underlying semantics of the life-cycle calculus based on channel theory, the theory of information flow proposed in [1]. Such properties would be difficult to prove, let alone infer, by inspecting the specification of the knowledge components, because it would require the cumbersome task of reasoning with the axioms that constitute the specification. Instead, they are easily proved by a theorem prover on a 'life-cycle level', namely by inspecting the structure of the life cycle and using knowledge about channel-theoretic operations and knowledge (or assumptions) of the initial properties of the component.

Although the life-cycle calculus rules given in Figure 5 suffice for the purposes of high-level knowledge management on the 'life-cycle level', calculus-specific goals like *weaken_spec* or *weaken_onto* are far too general for a meta-intepreter to be able to solve them in any particular domain by applying the transformation steps they capture. Suppose we wanted to recreate the execution of the reduction fragment of Ecolingua's life cycle as represented in the last clause of Figure 7. Within a distributed environment, the meta-interpreter should appeal to the local competences of knowledge transformers to execute this life cycle.

5 Executing Ecolingua's Life Cycle

We present a description of how agents might interact in the environment outlined in Section 3, based upon the *reduce* life-cycle pattern represented in the last clause of Figure 7. In the environment, the

Ecolingua ontology, in all its successive guises, would be the knowledge component owned by an agent, called, say, ECOLINGUA. Hence, initially this agent would have as its knowledge component $\langle O, S\rangle$, where O represents the meta-ontology over which the specification, S, of Ecolingua is given.

The constructs used to build the corresponding life-cycle history are shown in Table 1. They correspond to the transformation rules of the calculus (now expressed in terms of the agent-based environment), along with an additional construct, acq, used to describe the acquisition of some knowledge component, and its introduction into the system. This acquisition step is not covered by a calculus rule since it represents merely the introduction of some knowledge component into the domain of discourse, and not some particular transformation of it. However, it *is* needed in practical situations where we admit components that have histories external to the system.

In accordance with these constructs, then, the initial life-cycle history of component $\langle O, S\rangle$ is acq(ONTOLINGUA-SERVER, t_0), indicating that the first recorded step (at time t_0) in the life of this component, as far as this environment is concerned, has been its acquisition from the Ontolingua Server. It is necessary to record the time of each life-cycle step since the precise behaviour of the transformation described may itself be time dependent. Configured in this manner, then, ECOLINGUA is available for general problem-solving in the system; as its competence it may offer, for example, query services and property-checking upon the Ecolingua ontology.

construct	*description*
acq($Source, T$)	The knowledge component has been acquired from source *Source* at time T.
ss($LC, Agent, T$)	The specification of the component, previously having life-cycle history LC, has been strengthened by the actions of knowledge transformer *Agent* at time T.
sw($LC, Agent, T$)	The specification of the component, previously having life-cycle history LC, has been weakened by the actions of knowledge transformer *Agent* at time T.
os($LC, Agent, T$)	The ontology of the component, previously having life-cycle history LC, has been strengthened by the actions of knowledge transformer *Agent* at time T.
ow($LC, Agent, T$)	The ontology of the component, previously having life-cycle history LC, has been weakened by the actions of knowledge transformer *Agent* at time T.
cc(LC, SA, CA, T)	The component, previously having life-cycle history LC, is connected to the component of agent CA through the actions of knowledge transformer SA at time T.

Table 1: Life-cycle history constructs (times may be absolute or relative to the environment).

Now, assuming the system also contains the following knowledge transformers:

- CLEANUP, which offers the competence:

  $$weaken\_spec(A)$$

  where A is the name of some agent that is constrained to hold an appropriate knowledge component. The effect of the operation is to 'clean up' that ontology, with the removal of extraneous clauses in the manner described above in Section 2. Notice that this operation is equivalent to a *Specification Weakening* step in the reduction of the Ecolingua ontology.

- PRUNE, which also offers the competence:

  $$weaken\_spec(A)$$

where A is the name of some agent that is constrained to hold an appropriate knowledge component. The effect of the operation is to 'prune' it of irrelevant clauses in the manner described above in Section 2. Notice that this operation is also equivalent to a *Specification Weakening* steps in the reduction of the Ecolingua ontology.

- KIFTERMS2PROLOG, offering the competence:

$$weaken\_onto(A)$$

With the application of this competence, the terms of the specification of the knowledge component of the agent named A are transformed into Horn clauses — the *Ontology Weakening* step in the transformation of the Ecolingua ontology.

Notice that the number of arguments in the competences above does not coincide with the number of those of the predicate in the life-cycle pattern of Figure 7; this is because the competence indicates that knowledge transformations are performed on the knowledge component owned by A, who provides the input to the transformation, and collects the output.

In addition, the environment contains the life-cycle interpreter named ONTOLOGY-REDUCER, offering the competence:

$$reduce\_ontology(A)$$

This competence corresponds to the life-cycle sub-pattern specified only by the last clause of Figure 7 above, which represents the part of Ecolingua's life cycle after the ontology was obtained from the Ontolingua Server. Hence, this consists of two *Specification Weakening* steps followed by one *Ontology Weakening* step.

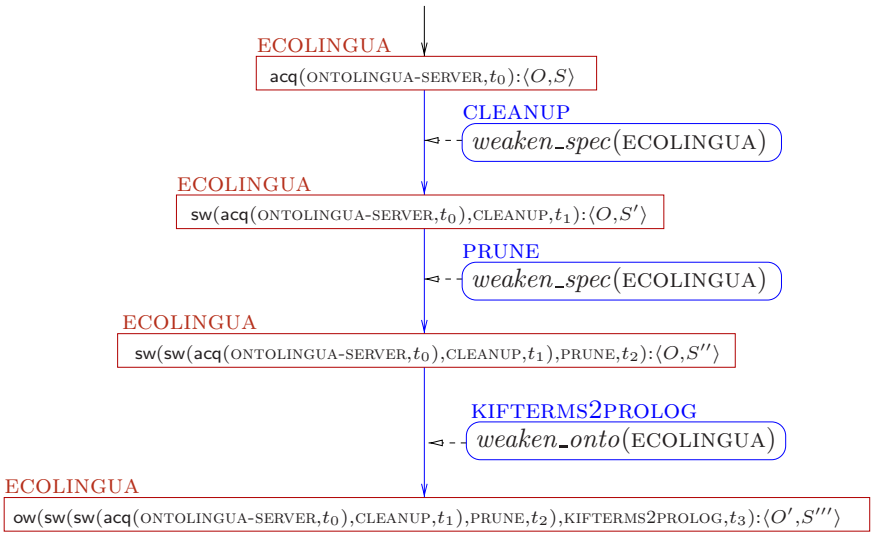

Figure 8: Evolution of the ECOLINGUA agent as transformed by knowledge transformers CLEANUP, PRUNE and KIFTERMS2PROLOG, following the life cycle of Figure 1

This competence of the ONTOLOGY-REDUCER life-cycle interpreter is domain-independent, and has been 'published' into the system since the ability to apply to other Ontolingua ontologies transformations akin to those undergone by Ecolingua was thought to be a potentially useful capability. However, since it is domain-independent, by itself it is incapable of effecting the particular transformation brought about on the Ecolingua ontology: for this, the domain-specific knowledge transformers defined above are necessary: these are able to perform the particular life-cycle steps required (these knowledge transformers can also be invoked independently of the life-cycle interpreter if their individual abilities are required elsewhere).

Figure 3 shows the *task panel* where some request is placed for the following operation:

$$reduce\_ontology(\text{ECOLINGUA})$$

that is, it is required that the Ecolingua ontology held in the agent ECOLINGUA be reduced, the brokering mechanism (assuming this is the implemented means of orchestrating problem-solving in the system) recognises this as a competence offered by the ONTOLOGY-REDUCER life-cycle interpreter. Accordingly this capability of ONTOLOGY-REDUCER is invoked.

Now, the first step in the *reduce_ontology*(ECOLINGUA) operation is a *weaken_spec*(ECOLINGUA) step, which must be performed in a domain-dependent fashion. Accordingly, a request is placed with the broker for satisfaction of the goal:

weaken_spec(ECOLINGUA)

Figure 4 shows how, at this particular stage of the execution of the life-cycle pattern, the broker gives the choice of two alternative solvers with the capability of performing the *weaken* life-cycle step. At this point the user interactively chooses the knowledge transformer CLEANUP, which offers this service: it is able to perform the domain-specific task required by the abstract life-cycle step.

By choosing CLEANUP, this knowledge transformer is invoked and it performs its operation, updating both the knowledge component of ECOLINGUA and the accompanying life-cycle history. For the subsequent request for satisfying the goal

weaken_spec(ECOLINGUA)

we would proceed analogously, but choosing the PRUNE knowledge transformer, instead. Figure 8 shows the effect of these two steps. Square boxes show the content of ECOLINGUA agent in the format

$$\boxed{\textit{life-cycle history} : \textit{knowledge component}}$$

and the round boxes show knowledge transformers CLEANUP, PRUNE and KIFTERMS2PROLOG in the format

$$(\underline{\textit{competence}})$$

acting upon ECOLINGUA. The specification S of the knowledge component is modified to S' and S'', and the life-cycle history reflects the fact that two *Specification Weakening* (SW) steps have been applied, by agent CLEANUP and PRUNE at time t_1 and t_2, respectively, to the initial component as it was acquired from the Ontolingua Server.

In similar fashion, the KIFTERMS2PROLOG knowledge transformer is called to realise the third step of the ontology reduction, namely:

weaken_onto(ECOLINGUA)

This is done, with the knowledge component of ECOLINGUA being modified again — as is the life-cycle history (see again Figure 8: this is an *Ontology Weakening* (OW) step).

Following these operations, then, the life-cycle history accompanying the Ecolingua ontology now details the sequence of domain-specific steps that have transformed this knowledge in the course of its existence in the system.

6 The Added Value of Life-Cycle Histories

In this section we show two examples of how formal life-cycle histories, accompanying the corresponding knowledge components, provide additional capabilities within the knowledge-engineering task.

In the first example we show how this additional information contained in life cycles can be readily made accessible in a human-oriented fashion for knowledge engineers to use. The second example further highlights the advantage of a formal semantics for knowledge life cycles as we show how we can mechanically check specific properties of knowledge components by inspecting their life-cycle histories.

6.1 Displaying Components at Different Life-Cycle Stages

When a knowledge component such as Ecolingua comes with a formal representation of its life-cycle description (its *life-cycle history*), we may make use of this additional information to display the content of a particular component at different stages of its life cycle.

We have added to the task panel illustrated in Figure 3 the capability to visualise the ontology that has been reduced according to a predefined life cycle, like the one we have been discussing in Section 5. This visualisation is realised by automatically synthesising web pages out of the structure of the specification of the ontology, which is contained in the output file of the reduction process. Figure 9 shows the web page for the final Ecolingua ontology.

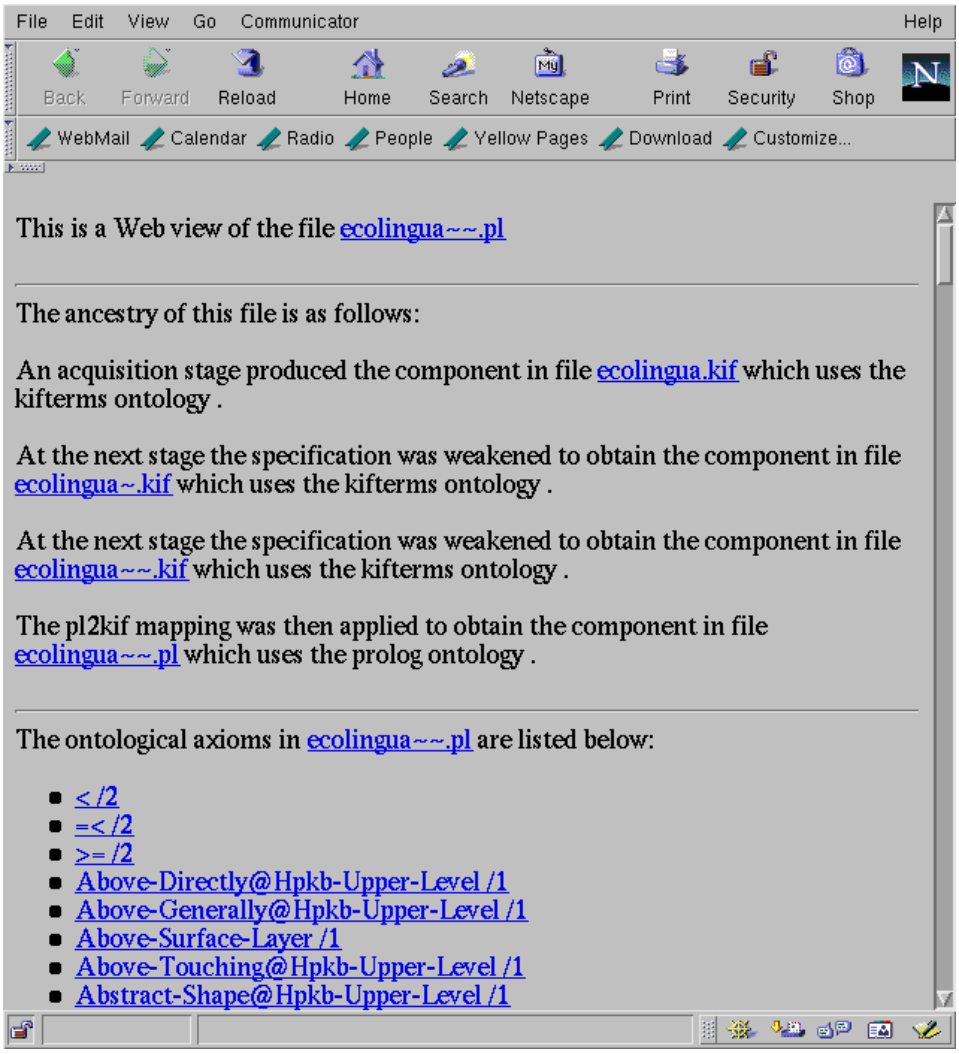

Figure 9: Synthesised web page displaying the life-cycle history

But in addition to the content of the knowledge component, the web page also describes the life-cycle steps the component has gone through, and links to the corresponding web pages that will show the component at previous stages of its life cycle. These pages will again be automatically synthesised out of the previous components and their corresponding life-cycle histories.

With this additional information knowledge engineers would be able to browse and analyse knowledge components at various stages of their evolution, hence allowing them to choose among a specific version of the component if they need so. One reason to do so would be if some particular property of the component is required. Formal life-cycle histories could also help an engineer to automatically check whether a knowledge component has a certain property. This will be our next example.

6.2 Checking Properties by Inspecting Life-Cycle Histories

Figure 10 shows our *life-cycle editor* again; here the knowledge engineer is about to ask for the validity of an argument concerning the soundness of Ecolingua. Specifically, with the premise that the *untransformed* ontology (identified here by the constant *eco*) is sound, the engineer would like to know if it can be proved that this property is maintained in the ontology *after* the transformation steps have been applied. The transformed ontology is expressed through its life-cycle history, as displayed in the bottom part of the window; this corresponds to the execution of the formal life-cycle pattern as it was represented with Horn clauses, and captures the structure of Ecolingua's life cycle as illustrated in Figure 1. (In this particular case, the soundness of the transformed ontology cannot be proved.)

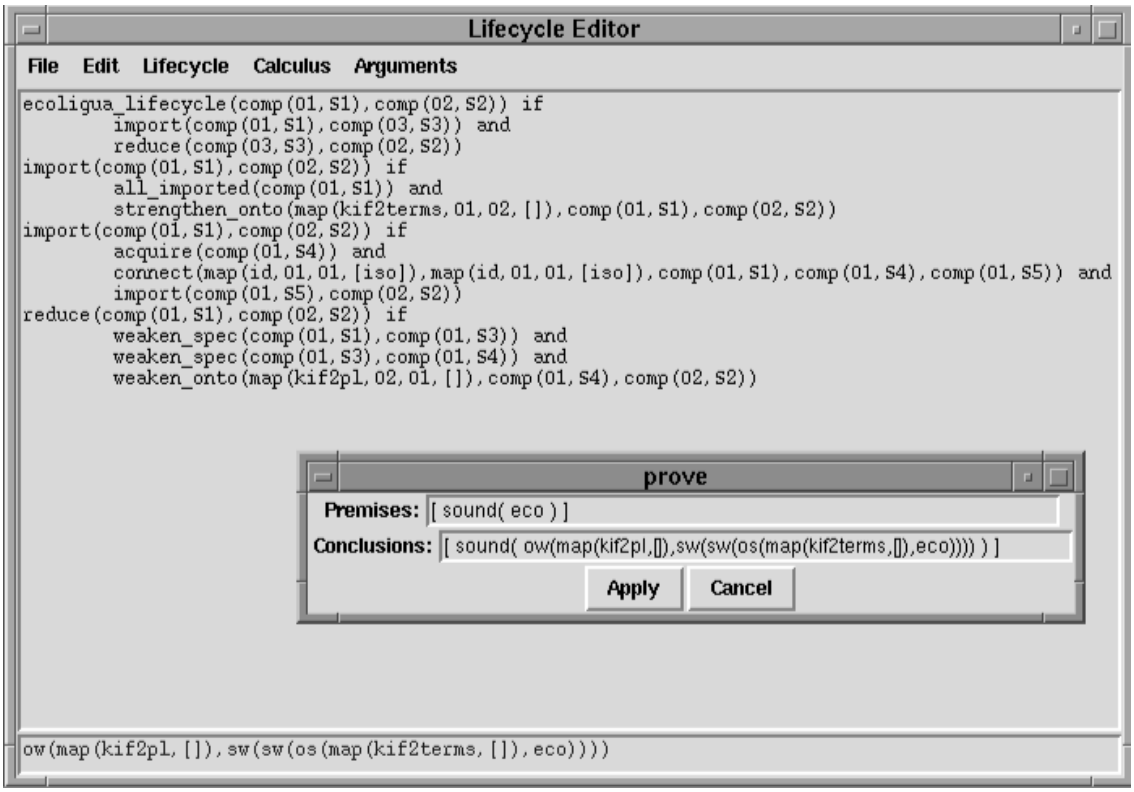

Figure 10: Proving arguments on Ecolingua's life cycle

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. Thus, this is something that would be impossible to know if we didn't have a life-cycle history that recorded the sort of transformations applied to Ecolingua. It is only practical to learn it through use of life-cycle 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.

7 Conclusions

Formal knowledge management must operate in a distributed setting if it is to work on the World-Wide Web. It will become important to be aware of the provenance and design history of the knowledge components we use. The life-cycle calculus is a formal language for describing the history of design components. It is general purpose but, as we have shown, it can be embedded within the sorts of specialist algorithms used in large-scale, automated knowledge management tasks.

The examples given in this paper are based on results for the calculus in use as part of a prototype agent-based brokering system. This uses Prolog as the agent implementation language and Linda [3] as the medium for asynchronous message passing. The concepts described, however, could equally well have been implemented using any other appropriate agent platform (e.g., Java agents and the JADE message passing environment).

Future work will develop our understanding of the interaction between life cycles and brokering of agent competences. We also shall be investigating the use of our calculus as an aid to tracking the evolution of versions of an ontology or knowledge base.

Acknowledgments

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. Seligman. *Information Flow: The Logic of Distributed Systems*. Cambridge University Press, 1997.

[2] 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.

[3] N. Carreiro and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4), 1989.

[4] 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.

[5] 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.

[6] M. Schorlemmer. Duality in knowledge sharing. In *Seventh International Symposium on Artificial Intelligence and Mathematics*, 2002.