



Division of Informatics, University of Edinburgh

Centre for Intelligent Systems and their Applications

Formal Knowledge Management in Distributed Environments

by

Marco Schorlemmer, Stephen Potter, Dave Robertson, Derek Sleeman

Informatics Research Report EDI-INF-RR-0136

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

June 2002

Formal Knowledge Management in Distributed Environments

Marco Schorlemmer, Stephen Potter, Dave Robertson, Derek Sleeman

Informatics Research Report EDI-INF-RR-0136

DIVISION *of* INFORMATICS

Centre for Intelligent Systems and their Applications

June 2002

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 and trust this raises.

In this paper, we introduce a formal calculus for describing transformations in the ‘lifecycles’ 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 encourages 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.

Keywords :

Copyright © 2002 by The University of Edinburgh. All Rights Reserved

The authors and the University of Edinburgh retain the right to reproduce and publish this paper for non-commercial purposes.

Permission is granted for this report to be reproduced by others for non-commercial purposes as long as this copyright notice is reprinted in full in any reproduction. Applications to make other use of the material should be addressed in the first instance to Copyright Permissions, Division of Informatics, The University of Edinburgh, 80 South Bridge, Edinburgh EH1 1HN, Scotland.

Formal Knowledge Management in Distributed Environments

W. Marco Schorlemmer¹, Stephen Potter¹, David Robertson¹, and Derek Sleeman²

¹ Centre for Intelligent Systems and their Applications

Division of Informatics
The University of Edinburgh
80 South Bridge
Edinburgh EH1 1HN
Scotland, UK
{marco,dr}@dai.ed.ac.uk
stephenp@aiai.ed.ac.uk
Tel: +44 131 6502732
Fax: +44 131 6506513

² Department of Computing Science

University of Aberdeen
Aberdeen AB24 3UE
Scotland, UK
sleeman@csd.abdn.ac.uk
Tel: +44 1224 272295
Fax: +44 1224 273422

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 and trust this raises.

In this paper, we introduce a formal calculus for describing transformations in the ‘lifecycles’ 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 encourages 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 cope with this state of affairs and describe the dynamics of knowledge, there would seem 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 ‘lifecycles’ 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 lifecycles 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 lifecycle descriptions might take is deferred until Section 4, where the notion of a *lifecycle 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 lifecycle 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, we make more concrete the ideas of formality for knowledge maintenance that lie at the heart of this research.

2 Ecolingua’s Lifecycle

We shall motivate our research by using a real example — the lifecycle 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 lifecycle. 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 sharable ontologies made available by the Ontolingua Server [5] it 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 an overly 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 Brillhante 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 accordingly. 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 fragment of *Ecolingua*'s lifecycle.

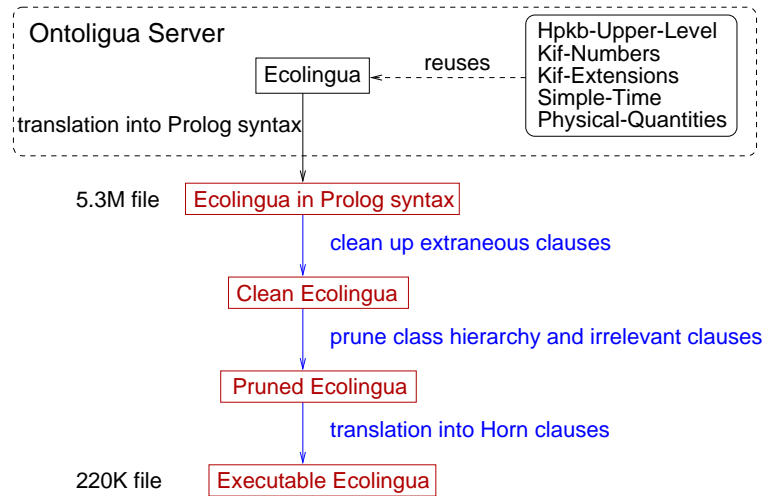


Fig. 1. *Ecolingua*'s lifecycle

We postulate that particular sequences of lifecycle 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 *lifecycle 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 *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 *Ecolingua*'s lifecycle 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 lifecycle steps (such as *weakening*) is justified by a formal lifecycle calculus that we shall introduce later in Section 4, where we also briefly discuss the formal representation of lifecycles (see also Figure 7, which gives the formal counterpart of the lifecycle in Figure 1). Notice that, although the

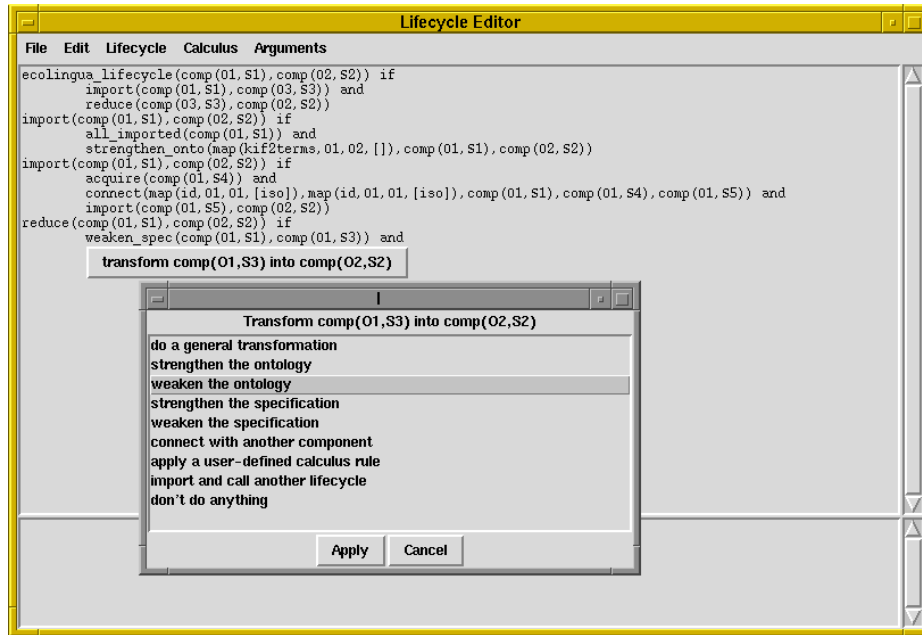


Fig. 2. Editing Ecolingua's lifecycle

depicted lifecycle editor uses an explicitly formal notation, it is possible to hide much of the formality by using domain/taste-specific editing operations.

Once the lifecycle of a knowledge component like Ecolingua is edited, we may publish it as a competence of a *lifecycle interpreter*, a meta-interpreter capable of executing the formal representation of a lifecycle. This lifecycle is described at a generic level; in order to be able to run the lifecycle in a particular domain, the execution is carried out by means of a brokering service that enables the knowledge engineer to choose among several solvers capable of performing abstract lifecycle steps; these solvers should have previously advertised their capabilities. 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 lifecycle, *ontology_reducer*, which reduces the Ecolingua ontology following the steps of the previously edited formal pattern of Ecolingua's lifecycle. Figure 4 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.

Eventually the lifecycle execution ends, yielding as a response the term that captures the *lifecycle history* of the knowledge component — Ecolingua in this case. This lifecycle history can later be used to infer properties of the components by looking at its evolutionary path rather than by inspecting the specification of the component itself.

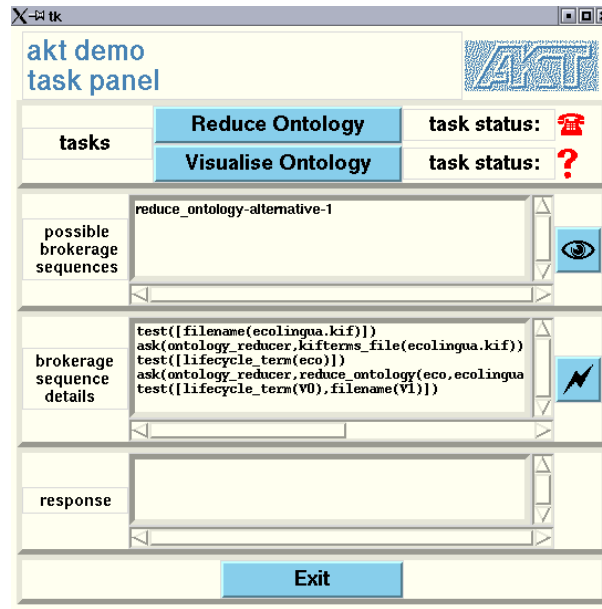


Fig. 3. Executing Ecolingua's lifecycle

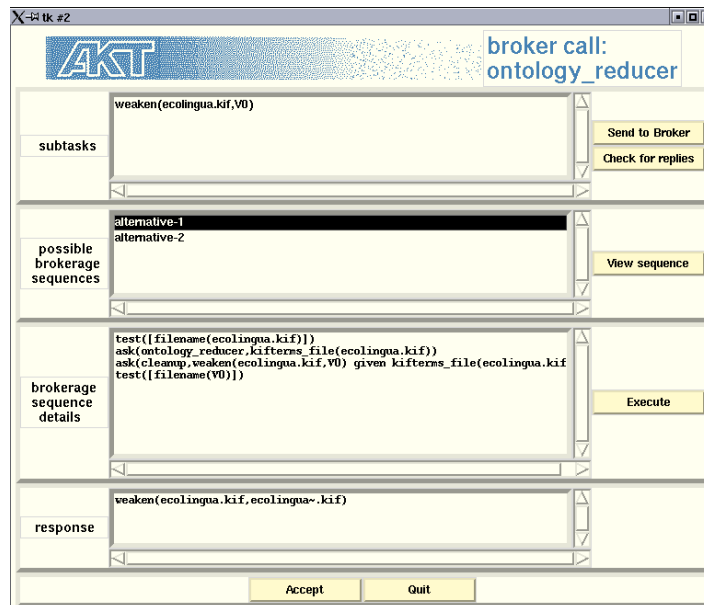


Fig. 4. Choosing a particular solver for an abstract lifecycle step

In the next section we give a deeper description of an environment that facilitates the sort of knowledge management we have briefly illustrated so far.

3 A Formal Knowledge Management Environment

We shall now describe a generalised system in which we postulate that formal notions of knowledge lifecycles can be exploited to effect a coherent knowledge management environment. These lifecycles correspond to sequences of the 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 lifecycle language.

We envisage that the environment will be a distributed, agent-based one so as to mirror 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 *lifecycle 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 lifecycle history alongside their knowledge components.
- 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 (taking in the ontology it uses) and any requirements and constraints it imposes in the exploitation of that knowledge. An agent's competences are advertised to the environment 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 lifecycle 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, and properties of these components — and thence, of the system in its entirety — can be proved and used to form a more complete picture of the state of the system, enabling 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):

- *s-agents*, agents whose competences include the ability to modify in some fashion the knowledge components of other agents. In other words, s-agents have the ability to perform one or more ‘lifecycle steps’ on these components. It is a requirement of the system that all s-agents that enter it agree to act in a principled manner: knowledge components must be altered in accordance with the calculus, and it is imperative that the s-agent also modifies appropriately the current lifecycle history associated with that component.
- *l-agents*, agents whose competences include the knowledge of how to alter knowledge components according to some pre-defined *lifecycle patterns* consisting of sequences of generalised lifecycle steps. The l-agent may itself execute one or more of these steps (in which case it would also be an s-agent, with the constraints that would entail), or it may rely on the capabilities of the s-agents in the system.

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 ‘lifecycle patterns’ and making them available within the environment as the competence of agents that can ‘execute’ these patterns would allow for the management of lifecycles when faced with the need to make similar transformations in the future. The role of the l-agent is to offer one or more of these lifecycle patterns to the system.

Using the s-agents, the knowledge components of other agents — including those of other s-agents — can be altered and recorded in a formalised manner. 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, trustworthiness and so on.

Figure 5 illustrates how a particular agent can interact with this environment. In the ‘problem-solving plane’, the agent’s knowledge component is used in combination with the knowledge of other agents to solve particular problems in a conventional agent-based manner. In the orthogonal ‘knowledge-evolving plane’, the successive operations performed upon the agent by the actions over time of s-agents in the environment result in the modification of its knowledge component — and the concomitant modification of its lifecycle history. It is the progress of knowledge in this plane that has hitherto been poorly documented, if at all; with the use of a lifecycle calculus, its path through time can be expressed and recorded, and this history made available when requested.

4 Formal Knowledge Lifecycles

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 lifecycles.

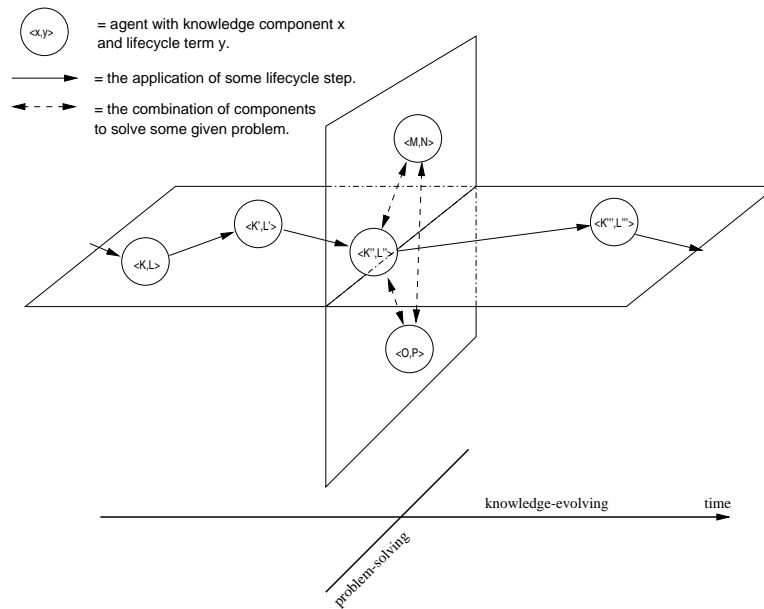


Fig. 5. Representation of the progress of an agent in the environment. See text for explanation.

4.1 Lifecycle Calculus

We capture the structure of the lifecycle through which a knowledge component like *Ecolingua* has gone by means of abstract transformations drawn from the set of lifecycle calculus rules given in Figure 6. The mathematical principles upon which the calculus is founded have been discussed in [6], and are based on a mathematical theory of information flow proposed by Barwise and Seligman in [1]. On one hand we want the rules to be general, since they attempt to capture only the essentials of the knowledge engineering activity and not to describe formally the details of numerous individual techniques. On the other hand we want them not to be so general that they do not have any link to the technologies we wish to analyse. We claim to have achieved a sensible level of abstraction as the example of distributed lifecycle 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 hence, that capture more properties of these techniques.

The lifecycle rules operate on abstract knowledge components; each component is described as the 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 of *Ecolingua*, then S is the content of the ontology, while O is actually the meta-ontology in which the ontology is specified. *Ontology Strengthening* captures, e.g., transformations that add ontological constraints or that translate the component into a more expressive logical language,

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

Fig. 6. Lifecycle calculus

and *Ontology Weakening* captures, e.g., transformations that remove ontological constraints or translate the component into a less expressive logical language. *Specification Strengthening* captures, e.g., transformations that add axioms to the component's specification or that generalise it, and *Specification Weakening* captures, e.g., transformations that remove axioms from the component's specification or that specialise it. Finally, *Component Connection* captures transformations that take two knowledge components (perhaps expressed using different ontologies) and merge them into a unified component, whenever their respective ontologies can be mapped into a common global ontology.

4.2 Lifecycle Patterns

We have implemented a prototype tool that supports the generation of patterns of lifecycles based on the lifecycle calculus discussed in Section 4.1. Figure 7 shows the formal representation as a set of Horn clauses generated for the pattern of Ecolingua's lifecycle discussed in Section 2 (see Figure 1) and displayed by the lifecycle editor in Figure 2.

The recursive *import* predicate captures the part of the lifecycle Ecolingua went through in the Ontolingua Server, while the *reduce* predicate captures the subsequent reduction performed on the output file of the server. The goal *weaken_spec* within the *reduce* predicate represents the clean-up and pruning steps the ontology went through and corresponds to a particular application of the *Specification Weakening* rule of the calculus. It takes knowledge component $\langle O_1, S_1 \rangle$ and delivers component $\langle O_1, S_3 \rangle$, where S_3 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 $\langle O_1, S_3 \rangle$

$$\begin{aligned}
& \text{ecolingua_lifecycle}(\langle O_1, S_1 \rangle, \langle O_2, S_2 \rangle) \leftarrow \\
& \quad \text{import}(\langle O_1, S_1 \rangle, \langle O_3, S_3 \rangle) \wedge \\
& \quad \text{reduce}(\langle O_3, S_3 \rangle, \langle O_2, S_2 \rangle) \\
& \text{import}(\langle O_1, S_1 \rangle, \langle O_2, S_2 \rangle) \leftarrow \\
& \quad \text{all_imported}(\langle O_1, S_1 \rangle) \wedge \\
& \quad \text{strengthen_onto}(O_1 \xrightarrow{\text{ol2kif}} O_2, \langle O_1, S_1 \rangle, \langle O_2, S_2 \rangle) \\
& \text{import}(\langle O_1, S_1 \rangle, \langle O_2, S_2 \rangle) \leftarrow \\
& \quad \text{acquire}(\langle O_1, S_4 \rangle) \wedge \\
& \quad \text{connect}(O_1 \xrightarrow{\text{id}} O_1, O_1 \xrightarrow{\text{id}} O_1, \langle O_1, S_1 \rangle, \langle O_1, S_4 \rangle, \langle O_1, S_5 \rangle) \wedge \\
& \quad \text{import}(\langle O_1, S_5 \rangle, \langle O_2, S_2 \rangle) \\
& \text{reduce}(\langle O_1, S_1 \rangle, \langle O_2, S_2 \rangle) \leftarrow \\
& \quad \text{weaken_spec}(\langle O_1, S_1 \rangle, \langle O_1, S_3 \rangle) \wedge \\
& \quad \text{weaken_onto}(O_1 \xrightarrow{\text{kif2pl}} O_2, \langle O_1, S_3 \rangle, \langle O_2, S_2 \rangle)
\end{aligned}$$

Fig. 7. Formal representation of Ecolingua’s lifecycle

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_3 into the new syntax. Since Horn 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 $\langle O_1, S_3 \rangle$.

We have chosen a Horn-clause representation of knowledge lifecycles to readily implement a meta-interpreter that allows the execution of a lifecycle, so that we can recreate the same pattern in different knowledge-engineering scenarios. The lifecycle 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 lifecycle pattern.

4.3 Managing Formal Lifecycles

Having the essentials of the knowledge-engineering activity expressed as formal lifecycle patterns allows the study and analysis of lifecycles as first-class citizens: we may consider lifecycles 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 lifecycle transformations a knowledge component goes through, and to infer properties that are preserved during different stages of a lifecycle.

For instance, transforming a knowledge component by weakening its specification preserves its soundness but not the completeness of the logic that models the compo-

ment. 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 lifecycle 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 ‘lifecycle level’, namely by inspecting the structure of the lifecycle and using knowledge about channel-theoretic operations and knowledge, or assumptions, of the initial properties of the component.

Although the lifecycle calculus rules given in Figure 6 suffice for the purposes of high-level knowledge management on the ‘lifecycle level’, calculus-specific goals like *weaken_spec* or *weaken_onto* are far too general for a meta-intpreter 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 lifecycle as represented in the last clause of Figure 7. Within a distributed environment, the meta-interpreter should appeal to the local competences of s-agents to execute this lifecycle.

5 Executing the Lifecycle

Therefore we present a description of how agents might interact in the environment outlined in Section 3, based upon the *reduce* lifecycle pattern represented in the last clause of Figure 7. In the environment, the *Ecolingua* ontology, in all its successive guises, would be contained within an agent, called, say, *ecoagent*. 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 this ontology is given.

The constructs used to build the corresponding lifecycle history are shown in Table 1. These constructs correspond to the transformations in 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.¹ In accordance with these constructs, then, the initial lifecycle 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.² Configured in this manner, then, *ecoagent* is available for general problem-solving in the system; as its competence it may offer, for example, query services and property-checking upon the ontology.

Now, assuming the system also contains the following s-agents:

¹ 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.

² It is necessary to record the time of each lifecycle step since the precise behaviour of the transformation described may itself be time dependent — the knowledge of an s-agent performing a step can itself be modified.

Table 1. The general lifecycle history constructs available (times may be absolute or relative to the environment). The last five constructs correspond to applications of the five lifecycle rules of Figure 6.

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

- *cleanup_and_prune*, which offers the competence:

`agent-weakenspec (Agent)`

where *Agent* is the name of some agent that is constrained to hold an appropriate ontology knowledge component. The effect of the operation is to ‘clean up’ that ontology, with the removal of extraneous clauses, and then to ‘prune’ it of irrelevant clauses in the manner described above in Section 4. In other words, this operation is equivalent to the specification weakening step in the reduction of the Ecolinga ontology.

- *kifterms2prolog*, offering the competence:

`agent-weakenspec (Agent)`

With the application of this competence, the terms of the specification of the ontology of the agent named *Agent* are transformed into Horn clauses — the ontology weakening step in the transformation of the Ecolinga ontology.

In addition, the environment contains the l-agent named *ontology_reducer*, offering the competence:

`reduce_ontology (Agent)`

This competence represents, in general terms, the lifecycle pattern represented by the last clause of Figure 7 above, as applied to the Ecolinga ontology obtained from the

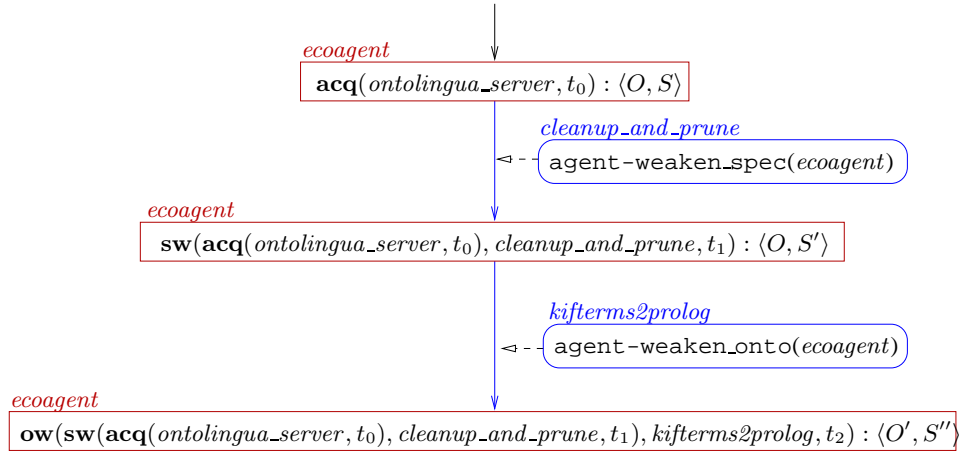


Fig. 8. Evolution of the *ecoagent* agent as transformed by s-agents *cleanup_and_prune* and *kifterms2prolog*, following the lifecycle of Figure 1

Ontolingua Server. Hence, this consists of a specification weakening step followed by an ontology weakening step.

This competence of the *ontology_reducer* l-agent 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 s-agents defined above are necessary: these are able to perform the particular lifecycle steps required (these s-agents can also be invoked independently of the l-agent if their individual abilities are required elsewhere).

So, assuming some request is placed in the system for the following operation:

`reduce_ontology(ecoagent)`

that is, it is required that the *Ecolingua* ontology held in the agent *ecoagent* 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* agent. Accordingly this capability of *ontology_reducer* is invoked.

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

`agent-weakenspec(ecoagent)`

The agent *cleanup_and_prune* offers this service; it is invoked, and performs its operation, updating both the knowledge component of *ecoagent* and the accompanying lifecycle. Figure 8 shows the effect of this step. Square boxes show the content of *ecoagent* in the format $\boxed{\text{lifecycle history} : \text{knowledge component}}$, and the round boxes

show s-agents *cleanup_and_prune* and *kifterms2prolog* in the format *competence* acting upon *ecoagent*. The specification S of the knowledge component is modified to S' , and the lifecycle history reflects the fact that a *Specification Weakening* (**sw**) step has been applied, by agent *cleanup_and_prune* at time t_1 , to the initial component as it was acquired from the Ontolingua Server.

In similar fashion, the *kifterms2prolog* agent is called to realise the second step of the ontology reduction, namely:

`agent-weaken_onto(ecoagent)`

This is done, with the knowledge component of *ecoagent* being modified again — as is the lifecycle history (see again Figure 8: this is an *Ontology Weakening* (**ow**) step).

Following these operations, then, the lifecycle 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 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 lifecycle 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 lifecycles and brokering of agent competences. We also shall be investigating the use of calculus as an aid to non-automated lifecycle tracking, for example, when one wishes to trace 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. W. M. Schorlemmer. Duality in knowledge sharing. In *Seventh International Symposium on Artificial Intelligence and Mathematics*, 2002.