Applying experienceware to support ontology deployment

Yannis Kalfoglou and David Robertson
University of Edinburgh,
Division of Informatics,
School of Artificial Intelligence,
Institute for Representation and Reasoning,
80 South Bridge, Edinburgh, EH1 1HN, Scotland
Email: {yannisk,dr}@dai.ed.ac.uk

Abstract

Experienceware is a paradigm which emerged in the late eighties and evolved during the nineties, resulting in technologies such as experience factories and their constituent experience bases. These are designed to manage experiences collected throughout the life-cycle of a software project. Ontologies emerged round about the same time as a way to represent consensual knowledge about a domain of interest in reusable and sharable formats. Despite their diverse origins and ways of development, there is an overlap of scope regarding one of their goals: to support reuse. In this paper we make use of this overlap by applying the experience factories paradigm to ontology deployment and in particular, to support ontology verification.

1. Motivation

In knowledge engineering, the study of ontologies is an attempt to prescribe appropriate, restricted formal languages for describing a target problem or domain. The idea is that by charting these in advance we make it easier for others to begin describing their problems in a way suited to standard forms of automation. These are now studied and applied in a variety of fields(see, for example, [21]).

Despite the increasing interest shown from various communities and the plethora of successful projects to which ontologies have been used there is still a lack of tools to facilitate deployment and maintenance. On the other hand, development has attracted the attention of researchers around the globe and we have already seen products such as: methodologies(i.e., [36], [16], [20]) and guidelines([18]) for building ontologies, online environments([14],[32],[12]) for collaborative construction, agent-based systems([4]) for supporting the selection task, and generic ontologies([28]) to be used as foundations for domain-specific ones.

However, to deploy ontologies correctly we need not only reuse-oriented tools and technologies as for example in the HPKB([11]) and IBROW([15]) projects. It is also necessary to record and organise our experiences in having applied them in order to improve future ontology deployment. It is hard to gain this sort of experience from the literature because few cases of comprehensive ontology reuse and deployment on a large scale are reported([35], [9], [38]). Even those which are reported do not normally discuss the hidden assumptions and tradeoffs identified during testing.

This paper explains how experience management may address this issue. We start by introducing the concepts of experience factories(hereafter, EF) and their constituent experience bases(hereafter, EB) in section 2. In section 3 we present a simple architecture that applies the experienceware[1] in a specific area of ontology deployment: verification. The whole section 4 is devoted to the implementation of the architecture in a step-by-step fashion. We further analyse the links with the EFs paradigm in section 4.6 and we speculate on the implications of the approach in section 5.

2. Managing experiences

In figure 1 we illustrate the organisation of an EF. EFs, which firstly investigated in the context of the TAME project[7] were further generalised in early nineties[6] as a mean to promote reuse of "all-kinds" of artefacts in an organisation. The core of an EF is the EB which acts as the organisational memory. The key idea is to install an organisational memory to support exchange of all kinds of experiences in the life cycle of a software project. While

[1]This expression was introduced in [39].

the structure of an EB might differ from that of an ontology, the purpose is the same: to support reuse. The main focus of an EF is to support 'learning from experience' on a technology-independent organisational level.

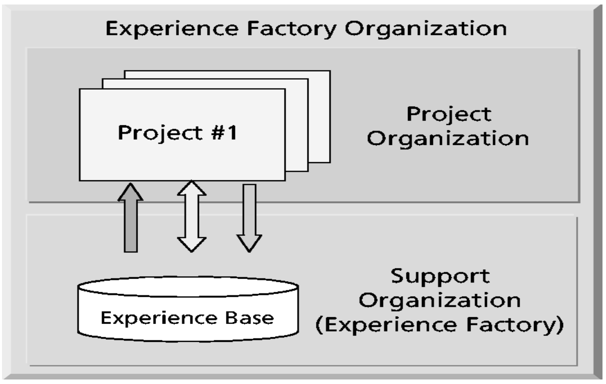

Figure 1. The *Experience Factory Organisation*.

As such, EFs do not provide design rationale as presented in [29] neither do they guide the software developer in design issues(examples of which are given in [40]). Rather, they focus on the preservation of prior knowledge in order to promote its reuse in future projects. An example of this is the EB on improvement ideas and problem-solution statements on supporting continuous improvement processes in hospitals which is described in [2]. Other example uses of EFs include their deployment to characterise and organise a range of products. In particular, in [8] the authors describe an EF based method for developing software systems that use a Case-Based Reasoning(CBR) approach. A publicly available result is the CBR Product Experience Base(PEB) accessible from the URL: http://demolab.iese.fhg.de:8080. In [33] the authors describe a representation formalism, called *Representation Formalism for Software Engineering Ontologies*(REFSENO), which supports the construction and manage reuse of EBs in the life-cycle of a software project. It was argued that this approach also provides knowledge maps for identifying sources of tacit knowledge in terms of authorship which later may be consulted for further guidance on the reuse process.

Such knowledge can also be described on different levels of abstraction. With ongoing experience collection(see, for example, [3], [5]), these different abstraction levels can be compared with one another. In this sense the EF approach could contribute to the question: "how ontologies should be?" for domains where there are no rich background sources of prior knowledge. While that issue is analysed in [22] here we focus on a particular aspect of managing experiences: how the experienceware(i.e., EFs and their constituents EBs) can be deployed to facilitate ontology use and in particular, ontology verification.

3. Ontology verification supported by experienceware

We argue that experienceware can be useful in ontology development and deployment as a way of managing the experiences collected from various agents participating in ontology building and usage. To make this idea more concrete, we built a simple architecture centred upon the notion of ontology verification. While we give the implementation details in the next section(4), here we focus on its operation. The architecture is given diagrammatically in figure 2.

The left-hand side of figure 2, depicts the task of verification. In particular, we are interested in verification of ontologies at the application level([26]). That is, we verify that ontological constructs are not misused by applications that adhere to an ontology. After applying our verification mechanism we accumulate, temporarily, the results in an EB. These are code-testing results and we regard them as experiences. The EB is then imported by an experiences editing tool which allows for further additions and modification of the description of existing experiences. It allows us to customise the experiences to the particular project as it provides a way of expressing information usually not obtainable through code-testing. We then select the experiences we want to validate and send them to a designated tool for verifying their correctness with respect to test results. This tool embodies the verification mechanism we deploy in the first step but here we apply it to verify the correctness of the results themselves. After the selected experiences have been validated we store them in the final EB to be part of an EF.

This cycle can be repeated as many times as we wish in the same or other ontologies to collect and manage the knowledge accumulated during verification and testing. Ultimately, this will result in an EF of ontology verification and testing that can be deployed in similar projects in order to facilitate ontology use.

4. Using EBs in ontology verification

To test this approach we focussed in a specific area of ontology use: verification at the application level. In [26] we elaborate on how this approach complements the proposed verification guidelines(see, for example, [17],[19]) which, however, are applicable only within the ontology development cycle. On the other hand, the verification at the application level we deploy requires the ontology to have been used in an application which we then check for occurrences of errors with respect to the ontological definitions. In that sense, we do not directly verify the underlying ontology but check the consistency of the developed model with respect to that ontology. However, our validation mechanism allow us to check the ontology the same since we reason

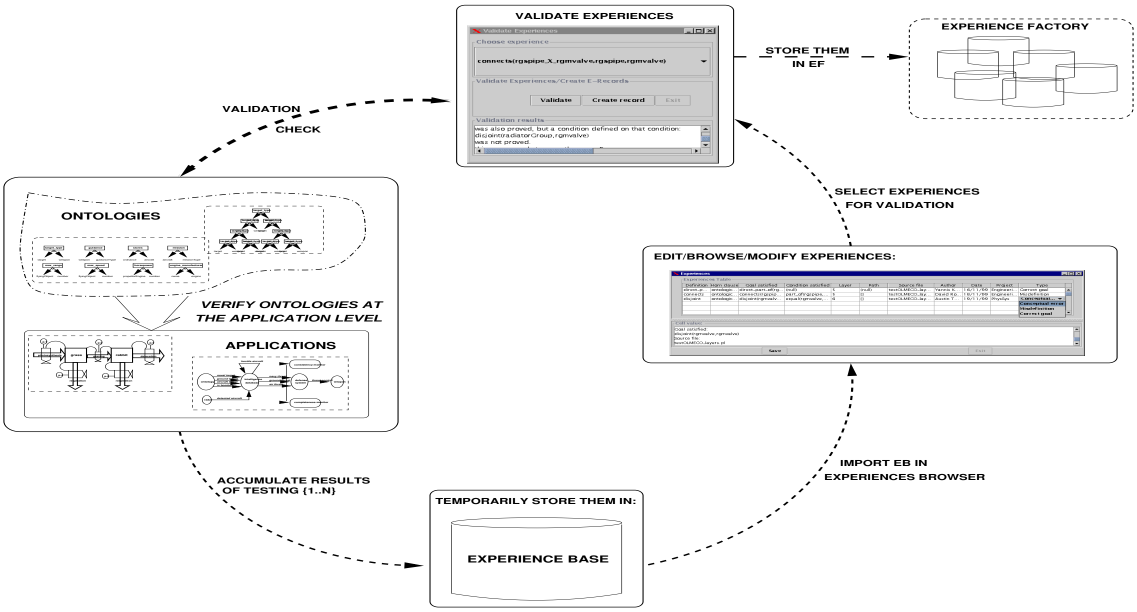

Figure 2. Experience-based architecture to support ontology verification.

about the correctness of ontological definitions as they used in the model. In most of the cases this model is been developed by those who build the original ontology as an exemplar system. An example of this is the *Supply Chain Processes*([30]) application developed as a testbed for the PIF([27]) ontology.

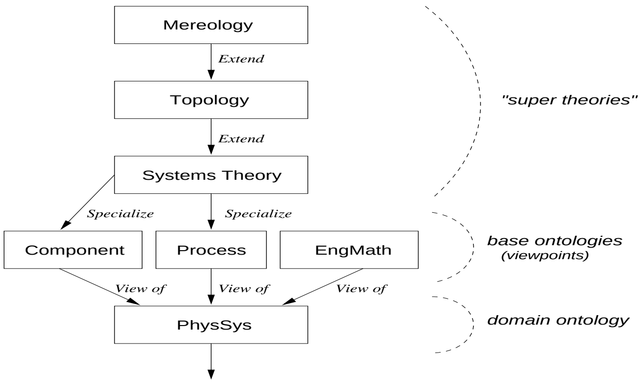

Figure 3. The inclusion lattice of PHYSSYS ontologies.

In this paper though, we use a more complex application developed by the PHYSSYS ontology authors to demonstrate its usage. PHYSSYS([9]) is a formal ontology based upon system dynamics theory and expresses different conceptual viewpoints on a physical system. It consists of three engineering ontologies formalising these viewpoints: a *component* ontology which represents the system layout; a *process* ontology for the physical processes that underlie behaviour; and an *EngMath* ontology which describes mathematical relations. The interdependencies between these ontologies are formalised as ontology projections and included in the PHYSSYS. The *component* ontology itself is constructed from mereology, topology and systems theory. A diagrammatic version of the ontology's inclusion lattice is given in figure 3, where the mereology, topology, and systems theory are 'super theories'; the component, process, and EngMath are base ontologies; and the produced PHYSSYS is a domain ontology.

We instantiated and executed the ontology using a *Hospital heating system* application developed by the PHYSSYS authors([9]). An illustration of the component view of the system appears in figure 4.

The system consists of two subsystems: one around the radiator and the other around the heater(*radiatorGroup* and *heaterGroup* in the figure) Its operation is briefly depicted in [9] as follows: "This system heats up the water up to a desired temperature regulated by the radiator. When the heater is on, its temperature first increases quickly because the water that flows into it is of almost the same temperature as the water that flows out of it. As the temperature increases the amount of heat that flows out of the heater will become larger than the heat carried by the water that flows into it. This will cause the radiator to become hot, and the temperature of the water from the radiator group return pipe(*rgrpipe* in the diagram) will increase and so will the temperature of the water that flows back into the heater. This will cause the heater temperature to increase

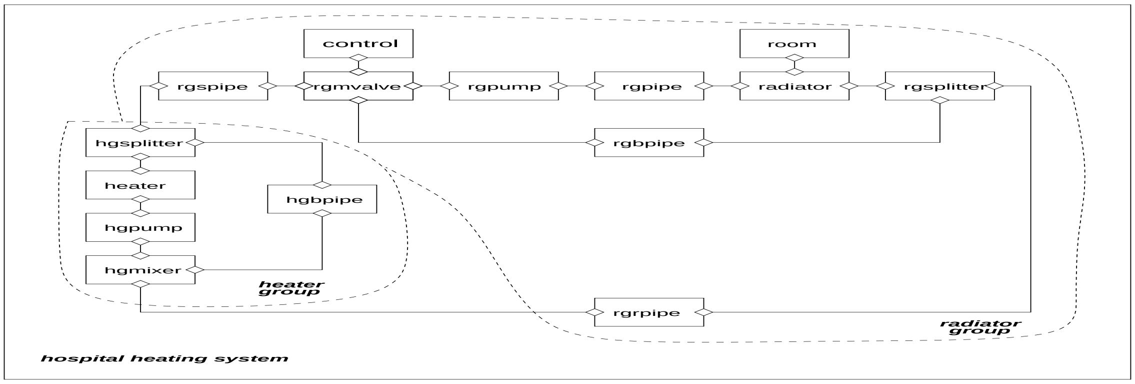

Figure 4. The component view of the *Schieden Hospital heating system*. Rectangle boxes used to represent system's components connected with straight lines. The dashed lines surrounding group of components denote the borders of the two subsystems: *radiatorGroup* and *heaterGroup*.

at a constant rate until the maximum heater temperature is reached and the heater is switched off." In the next section we describe the detection of misuse of ontological terms in the *Hospital heating system* by using the error checking mechanism we deploy([25]). Our original plan was to demonstrate its usage by introducing artificial errors in our applications([26]), but to our surprise we also detected an error in the original ontology. Recall from the introduction of this section, this is the result of validating the original ontology by using its instantiated model, the *Hospital heating system*. This case will be used to walk-through the architecture illustrated in figure 2. We start by explaining how our style of verification works with emphasis on accumulation of test results(section 4.1). These are then stored as experiences in a table where additional information regarding the project can be edited(section 4.4). In section 4.5 we present a mechanism for validating experiences which make it possible to detect the error occurred in the original ontology. All the information regarding error detection and validation outcome is then stored in an EB as we describe in section 4.6. The implementation uses `Prolog` as the inference engine during testing and `Java` for controlling user interaction.

4.1. Ontology verification

All the ontologies were implemented, originally, in Ontolingua using the Ontology server([13]). We translated them to the target language we use: `Prolog`. Many statements of the original ontologies are of the form $A \leftrightarrow B$ and this combines the implementation part which we write as: $A \leftarrow B$, and the constraint part which is $A \rightarrow B$. The former will be part of a specification of an application in Horn clauses, and since it is originating from the ontology we embed it within a predicate, `ontologicalDefinition/2` to distinguish it from the application specification statements which are within another predicate: `specification/2`. On the other hand, the latter represents a constraint on an ontological definition and we apply the equivalence: $A \rightarrow B \equiv \neg(A \wedge \neg B)$ to rewrite it as a goal appropriate for `Prolog`. We then record this as an error condition by losing the external negation and embedding the condition within the predicate `error/2`. Thus, if we have $error(A, \neg B)$ and can prove that $\neg B$ holds with respect to the specification being interpreted, then a potential misinterpretation of the ontological definition A has occurred.

We tested and evaluated the whole PHYSSYS ontology set, some of which are described in [26]. For the sake of brevity we will briefly present here only two of the seven ontologies comprising the set: mereology and topology. Further, we will demonstrate only a few of the devised cases where an ontological definition is being misused by the specification that adopts it. Our focus in this paper is on the usage of experienceware and its deployment in the context of ontologies and not to elaborate on the deployment of ontologies themselves. For the latter we point the interested reader to other publications, for example: [23], [26] and [24].

A surprising result as we mentioned earlier, was the detection of an ill-defined relation in the original ontology. In particular, we were interested to find whether the components *rgspipe* and *rgmvalve*(see figure 4) are linked together and by which connection. To test this we deploy the ontological definition, `connects/3`, which relates components to connections. As we can see from figure 4 these components are indeed linked through a connection which we define as `rgspipe_X_rgmvalve`. However, when we checked this definition against its ontological constraint we unexpectedly revealed an error. The error occurred in the mereology ontology and affected the `connects/3` definition which is using mereological definitions. However, the validation mechanism we deploy allowed us to show that the error was erroneously reported due to an ill-defined relation in the ontology.

We will use this case to walk-through the architecture depicted in figure 2 in order to answer questions such as: where it occurred?(section 4.2), why it was erroneously reported as an error?(section 4.3), and how the ill-definition was detected?(section 4.5). As we are going through each step we also collect and store as experiences the results of testing. In particular, during the ontology verification task(see figure 2) we record the following information:

- *ontological definition*: that is the ontological term(i.e.: *connects*) which will be formalised as a Horn clause in the specification;
- *Horn clause*: this is the formalised representation of the ontological definition in Horn logic. Examples of this are the clauses described in section 4.2;
- *Goal satisfied*: this is the goal we used for testing which has succeeded during execution;
- *Condition satisfied*: the error condition that has been satisfied during testing of a particular goal. In cases where no conditions where proved the goal is correct and no information is recorded;
- *Layer*: the layer to which the particular goal for testing belongs. In our ontology verification task we use a multi-layer approach in which each layer corresponds to a different ontology. This makes it easier to decompose complex ontological structures, as the PhysSys, and deploy them more easily in a specification. For example, in the PhysSys structure we have defined layer 4 for topology and 5 for mereology. These indices are given as the first argument of the predicates we mentioned above and explained in detail in [26];
- *Path*: the path which has been followed in proving a particular goal. This is useful to track and locate errors in the specification;
- *Source file*: the file which was used for testing.

4.2. Ontological definitions

The first question we answer is "where it occurred?" with respect to the ontological definition of `connects/3`. This is given in the topology ontology. However, the first ontology in the PhysSys inclusion lattice depicted in figure 3 is the **mereology**. It provides definitions for mereological relations to specify decomposition and the properties that any decomposition should have. We list below a part of the mereology ontology corresponding to the topological definition of our case in the form of the designated Horn clauses:

```
ontologicalDefinition(5,(proper_part_of(X,Z) ←
       part_of(X,Z))).
ontologicalDefinition(5,(proper_part_of(X,Z) ←
       part_of(X,Y) ∧ proper_part_of(Y,Z))).
ontologicalDefinition(5,(disjoint(X,Y) ←
    ¬ (equal(X,Y) ∨
    (proper_part_of(Z,X) ∧ proper_part_of(Z,Y))))).
```

The clauses can be given a declarative reading: The first two clauses formalise the notion of *proper_part_of* and are written as the recursive predicate `proper_part_of/2`. The base case states that when an individual `X` is part of individual `Z` then the `proper_part_of` relation holds. The recursive definition of `proper_part_of` realises the transitivity property. We use the `part_of/2` predicate to express static relations which hold with respect to a system that uses this ontology. In our case, as we can see from figure 4, these are ten components belonging to the *radiatorGroup* and five part of the *heaterGroup*. These two are then defined as parts of the *hospital heating system*. The `disjoint/2` clause holds for individuals that are not mereologically equal or do not share a part. The relation `equal(X,Y)` defines which individuals are considered to be mereologically equal and usually holds for `equal(X,X)`. These are static definitions tailored to the system at question, and in our case are all the components(square boxes) easily identified from figure 4.

The second ontology we present here is **topology**. It provides the means to express that individuals are connected. Axioms ensure that only sound connections can be made. We apply the principles given earlier to transform the Ontolingua syntax to Horn clauses. These are given below:

```
ontologicalDefinition(4,(connection(C) ←
          connects(C,X,Y))).
ontologicalDefinition(4,(connects(C,X,Y) ←
          (connect(C,X,Y) ∧ connect(C,Y,X)))).
```

The first clause, states that a connection `C` connects two individuals `X` and `Y`. The second clause, `connects/3`, realises the symmetrical property that holds for the *connects* relation. It uses the predicate `connect/3` to express instances with respect to the system that uses the topology ontology. In our case this is used to define the following connectors between the corresponding components identified from their name. For example the connector between components *rgspipe* and *rgmvalve*(figure 4) is defined as: `connect(rgspipe_X_rgmvalve,rgspipe,rgmvalve).`

In addition to these ontological definitions we also find, via translation from the Ontolingua syntax the ontological error conditions given below:

```
error(6,proper_part_of(X,Y),proper_part_of(Y,X)).
error(6,disjoint(X,Y), (equal(X,Y) ∨
     (proper_part_of(Z,X) ∧ proper_part_of(Z,Y)))).
```

Note that the `disjoint` error condition is the dual of its corresponding ontological definition which we gave earlier. This is because each error/precondition pair was obtained by "splitting" a double implication, as described earlier. The `proper_part_of` error condition is derived directly from original ontological definition.

In the topology ontology the following constraints are defined:

```
error(5,connection(C), ¬ connects(C,X,Y)).
error(5,connects(C,X,Y), ¬ connects(C,Y,X)).
error(5,connects(C,X,Y),((part_of(X,Y) ∨
            part_of(Y,X)) ∧ connect(C,X,Y))).
error(5,connects(C,X1,Y1),(connects(C,X2,Y2) ∧
            ((disjoint(X1,X2) ∧ disjoint(X1,Y2)) ∧
            (disjoint(Y1,X2) ∧ disjoint(Y1,Y2))))).
error(5,connects(C,X,Y), (part_of(X,Z) ∧
            disjoint(Z,Y) ∧ ¬ connects(C,Z,Y))).
```

The first two conditions are used to trap side-effects of the symmetrical property that holds for system's connections as well as invalid definitions of connections. The third condition prohibits a part from being connected to itself or its whole and uses the static definition `connect/3` to refer to specific system instances described above. The next error condition is used to detect errors when a connection connects an entirely separated pair of individuals. It uses the relation *disjoint* that has already been defined in mereology. The last error condition is designed to ensure that when a part whose whole is disjoint with an individual connected to the part, then that whole is also connected to the individual.

4.3. Detecting errors

Using the error detection mechanism we describe in [25] and the underlying theory which is given in section 4.5, we are able to detect when a misuse of ontological definition occurs in the specification. We present numerous example-cases in [26] and [24] but here we focus on one case, the erroneously reported `connects/3` relation, addressing the question introduced in section 4.1 "why it was erroneously reported as an error?".

The results returned from testing whether *rgspipe* and *rgmvalve* components are connected through connection `rgspipe_X_rgmvalve` reveal that there is an error occurrence. This happened because one the error conditions defined on the `connects/3` relation was satisfied:

```
error_condition_satisfied(5,
    connects(rgspipe_X_rgmvalve,rgspipe,rgmvalve),
 (part_of(rgspipe,radiatorGroup) ∧
 disjoint(radiatorGroup,rgmvalve) ∧
¬connects(rgspipe_X_rgmvalve,radiatorGroup,rgmvalve)))
```

The error condition $part\_of(X,Z) \wedge disjoint(Z,Y) \wedge \neg\ connects(C,Z,Y)$ on the $connects(C,X,Y)$ relation was satisfied because one of the components checked, the *rgspipe*, is part of another component, the *radiatorGroup*, which is disjoint from the second component checked, the *rgmvalve*, and *radiatorGroup* is not connected with *rgmvalve* through the `rgspipe_X_rgmvalve` connector.

In section 4.5 we will see how we can find that this error is erroneously reported. At this point we also collect the information described in 4.1 which will be imported by the experience table described below.

4.4. Editing experiences

To manage the experiences obtained from testing we have build table which allows us to add extra information regarding the tests. A screenshot of the *experiences table* is given in figure 5.

Using this table we can add information regarding the *author* of the experience, the *date* and the *project* used to derive the experience, and its *type*. We identify three possible types: *conceptual error* for goals that violated an error condition, that is, errors originating in the specification; *misdefinition* for goals that violated an error condition but this might happened due to an error originating in the ontological constraints; and *correct goal* for goals that do not violated error conditions. In figure 5 we present various results on testing. For example, the definition *disjoint* which belongs to the *testOLMECO_layers.pl* source file, is reported from experience author *Austin* on *19/11/99* as part of the project *PhysSys* and characterised as *conceptual error*. In the same manner definitions *direct_part_of* and *connects* are reported by authors *Yannis* and *David* as *correct goal* and *misdefinition*, respectively.

In addition to this direct information, we also found it useful to include the following, with respect to ontology characterisations(originally introduced in [34] and further analysed in [37]). We instantiated the proposed framework with two of the PHYSSYS ontologies: mereology and topology. Each item represents a category, followed by its instantiation in the particular context. We also include a short explanation of each category in a parenthesis following its name.

- `Purpose:`*(the purpose of the ontology)* In our tests, both mereology and topology provide the building blocks for PHYSSYS ontology. They formalise generic mereotopological relations as described in the literature(i.e., [31] and [10]);
- `Representation languages and paradigms:`*(Ontolingua? Description Logics? FLogic? CycL? Prolog? Clips? XML?)* The mereology and topology ontology were implemented in Ontolingua, we translated them in the implementation language we use: `Prolog`;
- `Meaning and formality:`*(to what extent and how formal is the specification of the meaning of each*

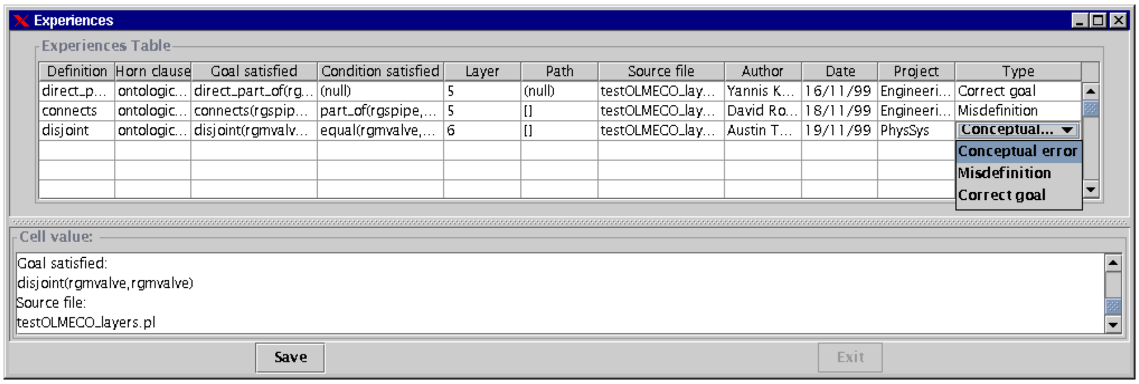

Figure 5. The Experiences Table

term?) The Ontolingua versions of mereology and topology provide primitive terms with axioms restricting their use by placing constraints on relationships between types of entities. The implemented versions we used(in `Prolog`), include this information along with application specific constructs;

- `Subject Matter`:*(is it domain-specific or general?)* Both mereology and topology provide generic relations and axioms to be used in the PhysSys ontologies set;

- `Scale`:*(how big is the ontology?)* Both ontologies are quite small, each of them formalises less than 10 relations;

- `Development`:*(the degree to which the application is specified, developed and/or fielded)* The mereology and topology ontologies are research prototypes but the PhysSys which includes them has been used in the context of the OLMECO project([9]);

- `Conceptual architecture`:*(what are the main components in the ontology application, and how do they relate to each other)* Two research prototype systems were used, the *hospital heating system* and the *air pump system*, both implemented in `Prolog`, and include ontological constructs;

- `Mechanisms and techniques`:*(what specific mechanisms and techniques were invoked to make use of the ontology)* We deployed a multi-layered architecture([26]) to embed ontological constructs in the applications. We translated ontological constructs to the target implementation language: `Prolog`, and translated ontological axioms to the constraints format used to detect discrepancies with respect to misuse of ontological constructs;

- `Implementation platform`:*(the particular implementation platform and context)* The ontologies are described textually in the literature and were also written in Ontolingua syntax,which we translated in `Prolog`. The testbed applications were also implemented in `Prolog`. The tests were executed through a `Java` front-end.

These characterisations provide ontology-specific information which can be used for organising and retrieving experiences on testing in similar ontology deployment efforts.

In a typical experiences recording framework we would proceed to store the information accumulated so far in an EB. However, in our approach we want to validate this information before storing it. This is a requirement in the ontology exercise where the existing ontological structures might be erroneously defined, as indeed happened in our case.

The information which will be used for validating an experience is the *Goal satisfied*. The remaining information does not need validation and will be part of the experiences record which will be created in the next phase(section 4.6).

4.5. Validating experiences

In order to reason about the conceptual error occurrences found in the specification with respect to the underpinning ontology, we deploy an error detection mechanism. This mechanism is based on the logic programming technique of meta-interpretation. We give the implementation details elsewhere([25] and [24]) and provide the theoretical foundations of error detection in [26]. In this section we focus on the validation of an erroneously reported error and thus addressing the third question introduced in 4.1: "how the ill-definition was detected?".

Recall from section 4.4, the information which we will validate is the *Goal satisfied*. In our testing these were: the *connects(rgspipe_X_rgmvalve,rgspipe,rgmvalve)* from topology ontology in layer 4, along with two more goals of mereology topology not described here. These goals are then imported in a Java-based application which uses the theory described in [26]. A screenshot of this application is shown in figure 6.

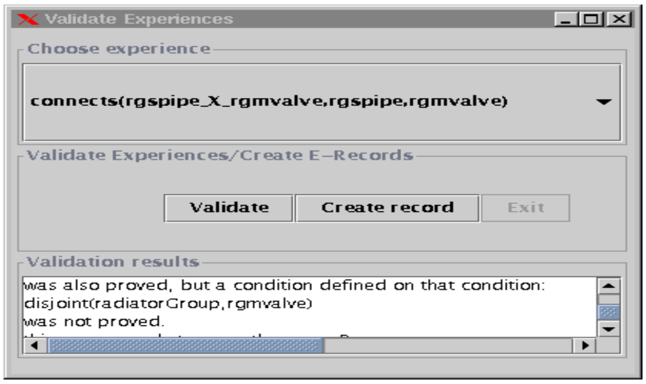

Figure 6. Validating experiences

Let us examine closely the validation of goal *connects(rgspipe_X_rgmvalve,rgspipe,rgmvalve)*, shown in the choice box of figure 6. In section 4.3 we saw that this goal is provable, and a quick look in the diagram of figure 4 verifies its correctness. However, it was found that it violates an error condition, therefore is considered to be a conceptual error. This situation can be clarified by deploying the error detection mechanism which reveals that an error has occurred in the original ontology.
In particular, the error was reported because the above goal satisfied the condition:

```
error(5,connects(C,X,Y), (part_of(X,Z) ∧
              disjoint(Z,Y) ∧ ¬ connects(C,Z,Y))).
```

However, there is another error condition:

```
error(6,disjoint(X,Y),(equal(X,Y)
                        ∨ (proper_part_of(Z,X) ∧
                        proper_part_of(Z,Y)))).
```

defined over the mereological definition 5a(given in section 4.2) *disjoint*, which is used as a subgoal of the *connects* goal given above and instantiated as follows:

`disjoint(radiatorGroup,rgmvalve)`
which is not provable.

In [26] we identified this situation as a definition of an erroneous error condition. The first part of that definition states that: “there exists an error condition, *E1*, defined on the given goal for proof, *G*, which is provable”, and in our case instantiated as follows:

```
part_of(rgspipe,radiatorGroup) ∧
 disjoint(radiatorGroup,rgmvalve) ∧
¬connects(rgspipe_X_rgmvalve,radiatorGroup,rgmvalve))).
```

and *G* is instantiated to:
the second part states that: “and there does not exist an error condition, *E2*, defined on that condition, *E1*, such that *E2* is provable.”. In our case *E2* is instantiated to:

```
equal(radiatorGroup, rgmvalve) ∨
      (proper_part_of(Z,radiatorGroup) ∧
      proper_part_of(Z,rgmvalve))
```

we should mention here that this condition is defined over a part of condition *E1* given above. This usually happens when we have a composite condition as in our case with three goals comprising condition *E1*: *part_of*, *disjoint*, and *¬connects*. So in the second part, we use a part of condition $E1$ which refers to the goal *disjoint* and instantiated as follows:
`disjoint(radiatorGroup,rgmvalve)`

As can be seen from the instantiations, the error condition *E2*, defined on a part of *E1*, was not proved because the *radiatorGroup* is not *equal* to *rgmvalve* neither they share a part(satisfying the *proper_part_of* relation). According to our error detection theory([26], this suggests that there is an erroneous error condition in topology ontology: *E1* because another condition, *E2*, defined on one of its parts(*disjoint*), was not proved. The side-effect is that we can prove that *radiatorGroup* is disjoint from *rgmvalve* and, consequently, condition $E1$ is provable since its third goal, *¬connects(rgspipe_X_rgmvalve, radiatorGroup, rgmvalve)*, is also provable, as it should. Therefore, the given goal *G* is reported as an error although it is actually correct.

The information collected from this validation task will be used in the next section where we store in the final EB all the experiences collected through the ontology verification cycle we described here.

4.6. Storing in EBs and linking to the EF paradigm

At this step we collect the information returned from validation, like the corresponding theoretical foundation cases(described in [26]), the conditions that were found erroneous(if any), and information already accumulated before(see sections 4.1 and 4.4), to build the experiences record.

Each record contains the following fields: *Ontological definition*, *Horn clause*, *Goal satisfied*, *Condition satisfied*, *Layer*, *Path*, *Source file*, accumulated from the verification task; *Condition not satisfied*, *Erroneous condition*, *Erroneous specification*, *Dependent condition*, *Error theory correspondence*, collected from the validation task; and *Author*, *Date*, *Project*, and *Report type* as entered in the experiences table(section 4.4).

In addition, each record is also linked to the characterisations of ontology entered in the experiences table. In the sequel, every record is stored in the final EB which in turn will be part of an EF on ontology verification. The structure of that EF could follow the architecture described in [1] where the authors describe an structure which consists of conceptualisations and characterisations of experiences. This structure helps us to browse through the collected experiences and ultimately supports the idea of "learning from experiences" which is an acknowledged need in ontology deployment.

5. Implications

In this paper we investigated the use of experienceware in the ontology field and in particular, in ontology verification. We conclude the paper by elaborating on the impact that experienceware can make in ontological engineering and vice versa by examining the benefits of this approach for each side:

- Firstly, we examine the software engineering side. In particular, we answer the question: "what do software engineers have to gain from the use of experienceware in ontologies?". Ontologies provide a rich field with diverse characteristics which can be used as a testbed for EFs. There are various ways in which this can benefit EFs, for example the use of ontologies as a testbed can improve EFs in order to cope with the intricate problems encountered when deploying ontologies. Furthermore, new uses of EFs are envisaged, as for example deploy EFs to structure and organise ontologies. In return, ontologies the means for software engineers to agree upon the structure of prospective EFs. The overlap of the fields can facilitate the understanding of the role of ontologies in software engineering which will be of benefit for both communities.

- On the ontology side we answer the question: "what ontological engineers have to gain from experienceware?". We support the view that the EF paradigm can play a vital role in ontology exercise. We can facilitate and improve ontology deployment when we collect and organise our experiences with ontologies. We envisage a plethora of potential applications in the whole ontology development and deployment lifecircle. Apart from EBs on testing, an example of which was shown in this paper, we envisage EBs on development, on maintenance, on reuse, etc. All these EBs can be part of an EF tailored to a particular ontology or a particular project. Having these EFs available in online libraries, alongside with their ontology counterparts, can help us to develop them better, deploy them faster, and reuse them more easily.

Acknowledgements The research described in this paper is supported by a European Commission's Marie Curie Fellowship(programme: Training and Mobility of Researchers).

References

[1] K-D. Althoff, A. Birk, S. Hartkopf, W. Muller, M. Nick, D. Surmann, and C. Tautz. Managing Software Engineering Experience for Comprehensive Reuse. In *Proceedings of the 11th International Conference on Software Engineering and Knowledge Engineering, SEKE'99, Kaiserslauten, Germany*, pages 10–19, June 1999.

[2] K-D. Althoff, F. Bomarius, W. Muller, and M. Nick. Using Case-Based Reasoning for Supporting Continuous Improvement Processes. In P. Perner, editor, *Proceedings of German Workshop on Machine Learning(FGML'99), Leipzig, Germany*, IBal Report, pages 54–61. Institute for Image Processing and Applied Informatics, 1999. ISSN: 1431-2360.

[3] K.-D. Althoff, M. Nick, and C. Tautz. Improving organizational memories through user feedback. In F. Bomarius, editor, *Proceedings of the Learning Software Organizations(LSO'99) workshop, Kaiserslauten, Germany*, pages 27–44, June 1999.

[4] J. Aspirez, A. Gomez-Perez, A. Lozano, and S. Pinto. (onto)2agent: An ontology-based www broker to select ontologies. In *Proceedings of the Workshop on Applications of Ontologies and Problem-Solving Methods, ECAI'98, Brighton, England*, pages 16–24, August 1998.

[5] R.V. Basili, F. Shull, and F. Lanubile. Building Knowledge through Families of Experiments. *IEEE Transactions on Software Engineering*, 25(4):456–473, August 1999.

[6] V. Basili, G. Caldiera, and D. Rombach. Experience Factory. In J. Marciniak, editor, *Encyclopedia of Software Engineering*, volume 1, pages 469–476. John Wiley & Sons, 1994.

[7] V.R. Basili and H.D. Rombach. The TAME Project: Towards Improvement Oriented Software Environments. *Transactions on Software Engineering*, SE-14(6):758–773, June 1988.

[8] R. Bergmann and K-D. Althoff. Methodology for Building Case-Based Reasoning Applications. In M. Lenz, B. Bartsch-Sporl, H-D. Burkhard, and S. Wess, editors, *Case-Based Reasoning Technology - From Foundations to Applications*, number 1400 in LNAI, pages 299–326. Springer Verlag, 1998.

[9] P. Borst, H. Akkermans, and J. Top. Engineering Ontologies. *International Journal of Human-Computer Studies*, 46:365–406, 1997.

[10] B.L. Clarke. A calculus of individuals based on "connection". *Notre Dame Journal of Formal Logic*, 22:204–218, 1981.

[11] P. Cohen, R. Schrag, E. Jones, A. Pease, A. Lin, B. Starr, D. Gunning, and M. Burke. The DARPA High Performance Knowledge Bases project. *AI Magazine*, 19(4):25–49, 1998.

[12] J. Domingue. Tadzebao and WebOnto: Discussing, Browsing, and Editing Ontologies on the Web. In *Proceedings of the 11th Knowledge Acquisition, Modelling and Management Workshop, KAW'98, Banff, Canada*, April 1998.

[13] A. Farquhar, R. Fikes, W. Pratt, and J. Rice. The Ontolingua Server: a Tool for Collaborative Ontology Construction. In *proceedings of the 10th Knowledge Acquisition Workshop, KAW'96,Banff,Canada*, November 1996. Also available as KSL-TR-96-26.

[14] A. Farquhar, R. Fikes, and J. Rice. The ontolingua server: a tool for collaborative ontology construction. *International Journal of Human-Computer Studies*, 46(6):707–728, June 1997.

[15] D. Fensel, V.R. Benjamins, E. Motta, and B. Wielinga. UPML: A Framework for knowledge system reuse. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence, IJCAI'99, Stockholm, Sweden*, pages 16–21, August 1999.

[16] M. Fernandez, A. Gomez-Perez, and N. Juristo. METHONTOLOGY: From Ontological Arts Towards Ontological Engineering. In *Proceedings of the AAAI-97 Spring Symposium Series on Ontological Engineering, Stanford, CA, USA*, pages 33–40, March 1997.

[17] A. Gomez-Perez. Towards a Framework to Verify Knowledge Sharing Technology. *Expert System with Applications*, 11(4):519–529, 1996.

[18] T.R. Gruber. Towards principles for the design of ontologies used for knowledge sharing. *International Journal of Human-Computer Studies*, 43:907–928, 1995.

[19] M. Gruninger. Designing and Evaluating Generic Ontologies. In *Proceedings of the workshop on ontological engineering, ECAI'96, Budapest, Hungary*, August 1996.

[20] M. Gruninger and M.S. Fox. Methodology for the Design and Evaluation of Ontologies. In *Proceedings of the IJCAI'95 Workshop on Basic Ontological Issues in Knowledge Sharing, Montreal, Quebec,Canada*, August 1995.

[21] N. Guarino, editor. *Formal Ontology In Information Systems*, Frontiers in Artificial Intelligence and Applications. IOS Press, June 1998. ISBN: 90-5199-399-4.

[22] Y. Kalfoglou, T. Menzies, K-D. Althoff, and E. Motta. Meta-knowledge in systems design: panacea...or undelivered promise? *The Knowledge Engineering Review(submitted)*, 2000.

[23] Y. Kalfoglou and D. Robertson. A Case Study in Applying Ontologies to Augment and Reason about the Correctness of Specifications. In *Proceedings of the 11th International Conference on Software Engineering and Knowledge Engineering, SEKE'99, Kaiserslauten, Germany*, pages 64–71, June 1999. Also as: Research Paper No.927, Dept. of AI, University of Edinburgh.

[24] Y. Kalfoglou and D. Robertson. Managing Ontological Constraints. In *Proceedings of the IJCAI-99 Workshop on Ontologies and Problem-Solving Methods(KRR5), Stockholm, Sweden*, http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-18/, August 1999. Also as: Research Paper No.948, Dept. of AI, University of Edinburgh.

[25] Y. Kalfoglou and D. Robertson. Use of Formal Ontologies to Support Error Checking in Specifications. In D. Fensel and R. Studer, editors, *Proceedings of the 11th European Workshop on Knowledge Acquisition, Modelling and Management(EKAW99), Dagstuhl, Germany*, pages 207–220, May 1999. Also as: Research Paper No.935, Dept. of AI, University of Edinburgh.

[26] Y. Kalfoglou, D. Robertson, and A. Tate. Using Meta-Knowledge at the Application Level. *Journal of Artificial Intelligence Research, submitted*, 2000. Also as: Research Paper No.956, Dept. of AI, University of Edinburgh.

[27] J. Lee, M. Gruninger, Y. Jin, T. Malone, A. Tate, G Yost, and other members of the PIF working group. The PIF Process Interchange Format and framework. *Knowledge Engineering Review*, 13(1):91–120, February 1998.

[28] D.B. Lenat and R.V. Guha. *Building large knowledge-based systems. Representation and inference in the Cyc project*. Addison-Wesley, Reading, Massachusetts, 1990.

[29] T.P. Moran and J.M. Carroll. *Design Rationale: Concepts, Techniques, and Use*. Lawerence Erlbaum Associates, 1996. ISBN: 0-8058-1567-8.

[30] S. Polyak, J. Lee, M. Gruninger, and C. Menzel. Applying the Process Interchange Format(PIF) to a Supply Chain Process Interoperability Scenario. In A. Gomez-Perez and R. Benjamins, editors, *Proceedings of Workshop on Applications of Ontologies and Problem Solving Methods, ECAI'98, Brighton, England*, August 1998.

[31] P. Simons. *Parts: A Study in Ontology*, pages 5–128. Oxford: Clarendon Press, 1987.

[32] B. Swartout, R. Patil, K. Knight, and T. Russ. Toward Distributed Use of Large-Scale Ontologies. In *Proceedings of the 10th Knowledge Acquisition, Modeling and Management Workshop(KAW'96),Banff,Canada*, November 1996.

[33] C. Tautz and C.G. von Wangenheim. REFSENO: A Representation Formalism for Software Engineering Ontologies. In *Proceedings of the 5th German Conference on Knowledge-Based Systems(XPS99), Wurzburg, Germany*, pages 61–71, 1999.

[34] M. Uschold. Where are the Killer Apps? In Gomez-Perez,A. and Benjamins,R., editor, *Proceedings of Workshop on Applications of Ontologies and Problem Solving Methods, ECAI'98, Brighton, England*, August 1998.

[35] M. Uschold, P. Clark, M. Healy, K. Williamson, and S. Woods. An Experiment in Ontology Reuse. In *Proceedings of the 11th Knowledge Acquisition Workshop, KAW98, Banff, Canada*, April 1998.

[36] M. Uschold and M. Gruninger. Ontologies: principles, methods and applications. *The Knowledge Engineering Review*, 11(2):93–136, November 1996.

[37] M. Uschold and R. Jasper. A Framework for Understanding and Classifying Ontology Applications. In *Proceedings of the IJCAI-99 Workshop on Ontologies and Problem-Solving Methods(KRR5), Stockholm, Sweden*, August 1999. http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-18/.

[38] A. Valente, T. Russ, R. MacGrecor, and W. Swartout. Building and (Re)Using an Ontology for Air Campaign Planning. *IEEE Intelligent Systems*, 14(1):27–36, January 1999.

[39] C. von Wangenheim, A. von Wangenheim, and R.M. Barcia. Case-Based Reused of Software Engineering Measurement Plans. In *proceedings of the 10th International Conference on Software Engineering and Knowledge Engineering(SEKE'98), San Fransisco, CA, USA*, pages 193–200, June 1998.

[40] T. Winograd. *Bringing Design to Software*. Addison-Wesley, 1996. ISBN: 0-201-85491-0.