Patterns for Interfacing between Logic Programs and Multiple Ontologies

Luís Cruz-Filipe[1,2,4], Isabel Nunes[3,4] and Graça Gaspar[3,4]
[1]*Dept. of Maritime Engineering, Escola Superior Náutica Infante D. Henrique, Paço d'Arcos, Portugal*
[2]*CMAF, Lisbon, Portugal*
[3]*Dept. Informatics, Faculty of Sciences, University of Lisbon, Lisbon, Portugal*
[4]*LabMag, Lisbon, Portugal*

Keywords: Design Patterns, Hybrid Semantic Web Languages, Dl-programs.

Abstract: Originally proposed in the mid-90s, design patterns for software development played a key role in object-oriented programming not only in increasing software quality, but also by giving a better understanding of the power and limitations of this paradigm. Since then, several authors have endorsed a similar task for other programming paradigms, in the hope of achieving similar benefits.
In this paper we discuss design patterns for hybrid semantic web systems combining several description logic knowledge bases via a logic program. We introduce eight design patterns, grouped in three categories: three elementary patterns, which are the basic building blocks; four derived patterns, built from these; and a more complex pattern, the study of which can shed some insight in future syntactic developments of the underlying framework. These patterns are extensively applied in a natural way in a large-scale example that illustrates how their usage greatly simplifies some programming tasks, at the level of both development and extension.
We work in a generalization of dl-programs that supports several (possibly different) description logics, but the results presented are easily adaptable to other existing frameworks such as multi-context systems.

1 INTRODUCTION

In the mid-nineties, the Gang of Four's work on software design patterns (Gamma et al., 1995) paved the way for important advances in software quality; presently, many valuable experienced designers' "best practices" are not only published but effectively used by the software development community. From very basic, abstract, patterns that can be used as building blocks of several more complex ones, to business-specific patterns and frameworks, dozens of design patterns have been proposed, e.g. (Adams et al., 1996; Meyer, 1997; Schmidt et al., 2000; Fowler, 2002; Larman, 2004; Mattson et al., 2005; Erl, 2009), establishing a kind of common language between development teams, which substantially enriches their communication, and hence the whole design process.

Most of the work around design patterns has been focused in the object-oriented paradigm, although some of the patterns are fundamental enough to be independent of the used modeling and programming paradigms. Some effort has also been made in adapting some of these best practices to other paradigms and in finding new paradigm-specific patterns (Antoy and Hanus, 2002; Sterling, 2002; Oliveira and Gibbons, 2005; Gibbons, 2006). In this spirit, we carried the task of identifying several basic and other, more complex, patterns in the paradigm of dl-programs (Eiter et al., 2008) – which join description logics with rules (expressed as a Datalog-like logic program) –, a powerful and expressive approach to reasoning over general knowledge bases or ontologies.

To the best of our knowledge, design patterns have not been studied in the framework of dl-programs, in spite of their importance. That is the goal of this paper. In order to enhance both the modular nature of dl-programs and the relevance of some important design principles, we resort to multi description logic programs (Mdl-programs), a straightforward generalization of dl-programs to accommodate for several description logics.

This work should be seen as quite distinct from that on ontology design patterns (Gangemi and Presutti, 2009), a catalogue of which is maintained at http://ontologydesignpatterns.org/. As their name

Cruz-Filipe L., Nunes I. and Gaspar G..
Patterns for Interfacing between Logic Programs and Multiple Ontologies.
DOI: 10.5220/0004544100580069
In *Proceedings of the International Conference on Knowledge Engineering and Ontology Development* (KEOD-2013), pages 58-69
ISBN: 978-989-8565-81-5

suggests, ontology design patterns are used in ontology development, both to build new ontologies from scratch or from other ontologies, and to update existing ones. In the setting of Mdl-programs, the ontologies are seen as immutable, being used and not changed; therefore, our patterns focus almost exclusively on the rule part of the Mdl-program, which is the entity responsible for coordinating the ontologies. Thus, ontology design patterns and design patterns for Mdl-programs should in general be seen as two complementary techniques, and not as alternatives.

The remainder of the paper is structured as follows. Section 2 explains the context and goals of this work in more detail. Section 4 presents seven different design patterns, and Section 5 illustrates their combined use by means of a larger example. Section 6 explores limitations and future directions of research, and Section 7 summarizes the contributions presented earlier.

2 MOTIVATION

The usefulness of combining description logics with rule-based reasoning systems led to the introduction of dl-programs (Eiter et al., 2008; Eiter et al., 2011), that couple a description logic knowledge base with a generalized logic program, interacting by means of special atoms, the *dl-atoms*.

Although the two components of a dl-program are kept independent, giving dl-programs nice modularity properties, there is a bidirectional flow of information via dl-atoms.

The purpose of this paper is to study design patterns in dl-programs, as was previously done for other programming paradigms – object-oriented (Gamma et al., 1995), service-oriented (Erl, 2009), functional (Norvig, 1996; Antoy and Hanus, 2002; Gibbons, 2006), logic (Sterling, 2002) and others. As several of these authors observed, studying design patterns in different programming paradigms is far from being a trivial task: each paradigm has its specific features, meaning that patterns that are very straightforward in one paradigm can be very complex in another, and vice-versa.

Looking at dl-programs, it is clear that they represent a completely different programming paradigm – not only are they closely related to the logic programming paradigm, but they involve description logic knowledge bases, in the presence of which the study of design patterns attains a different quality: on the one hand, some patterns become trivial (such as FAÇADE) or meaningless (such as DYNAMIC BINDING or SINGLETON), on the other hand some patterns pose totally new problems that have not been addressed in other paradigms where they do not arise (such as PROXY, which we will discuss in Section 6).

We claim that dl-programs, being a dual-component system, with a description logic and a logic-based rule language, provide the adequate setting for the study of design patterns for the Semantic Web. We will work with a straightforward generalization that we will define precisely in the next section: Mdl-programs (for Multi Description Logic programs), which incorporate not one, but a finite set of description logic knowledge bases. Mdl-programs are, therefore, multi-component systems connected by a logic program. This makes perfect sense when we consider some of the more elaborate design patterns, whose real power can only be appreciated in this more general setting. Furthermore, most theoretical results known for dl-programs immediately translate to similar results about Mdl-programs, fully justifying the use of the latter.

There are two questions that immediately come to mind, though. In the first place, if one wishes to consider several ontologies, what are the advantages of using Mdl-programs instead of merging all the desired ontologies and use the result in a standard dl-program? Also, why work within the framework of Mdl-programs instead of within other existing frameworks, such as the more general HEX-programs (Eiter et al., 2006a) or multi-context systems (Brewka and Eiter, 2007)? We feel it is important to provide satisfactory answers to these two pertinent issues before proceeding.

As regards the first point, there are a number of advantages to keeping ontologies separate, which essentially coincide with the reasons universally invoked to defend modularity of large-scale systems. Not only is it much more convenient to have independent knowledge bases (which might even be physically separated, or independently managed) than a single, gigantic one, but merging ontologies is in itself a mighty task with its own specific problems (Grau et al., 2005; Bruijn et al., 2006).

Furthermore, ontologies are typically very wide-spectrum, and in practice one does not need to work with each of them in its entirety. Therefore, keeping them separate also reduces the number of compatibility issues one has to deal with to the essential minimum. Clearly, if there are two concepts in two different ontologies that should be identified (from a particular application's perspective) but are logically inconsistent, this will always be a problem. However, if that particular inconsistency is irrelevant for the goal at hand (because the concepts involved are not used), then *not* merging the ontologies will simply not raise

that question – and our approach allows us to bypass it altogether.

Also, the separation of the knowledge bases allows us to make the most of the positive aspects of each knowledge base – which is relevant if, say, one of them is very efficient at performing specific reasoning tasks, while another features richer concept and role constructions.

For these reasons, we feel that a setting that keeps all relevant knowledge bases separate (where their interaction is externally controlled) is preferable not only for the purposes of this paper, but also as a general principle.

As regards the second point, dl-programs (and Mdl-programs) limit heterogeneity to two different frameworks: description logics for the knowledge bases part and logic programming for the rule part; the latter somehow represents the "conductor" that "coordinates" the other parts. However, they fully support non-monotonicity (even at the level of the description logic knowledge bases as will be seen later by application of a specific basic pattern). Mdl-programs are therefore a simpler framework than other, more powerful, alternatives – which is an advantage for our purpose; still, description logics *are* at the core of the Semantic Web, with a huge effort being currently invested in the interchange between OWL – an extension of the description logic SROIQ and a W3C recommendation – and a diversity of rule languages (Kifer and Boley (eds.), 2010).

Furthermore, as we will discuss, Mdl-programs inherit all the good properties of dl-programs, which made them interesting in the first place, while simultaneously keeping enough key ingredients of multi-context systems and HEX-programs to make the study of design patterns general.

3 Mdl-programs

Multi-description logic programs, the framework we will use to introduce our design patterns in, generalize the definition of dl-programs in (Eiter et al., 2008) to accommodate for several description logic knowledge bases. This is in line with (Wang et al., 2005), although we will stick to the original operators $\uplus$ and $\cup$ in dl-atoms.

A *dl-atom* relative to a set of knowledge bases $\{\mathcal{L}_1,\ldots,\mathcal{L}_n\}$[1] is

$$DL_i\left[S_1\, op_1\, p_1,\ldots,S_m\, op_m\, p_m; Q\right](t),$$

often abbreviated to $DL_i[\chi;Q](t)$, where: (1) $1 \leq i \leq n$; (2) each S_k, with $1 \leq k \leq m$, is either a concept or a role from $\mathcal{L}_i$ or a special symbol in $\{=,\neq\}$; (3) $op_k \in \{\uplus,\cup\}$; (4) p_k are the *input predicate symbols*, which are unary or binary predicate symbols depending on the corresponding S_k being a concept or a role; and (5) $Q(t)$ is a *dl-query* in the language of $\mathcal{L}_i$, that is, it is either a concept inclusion axiom F or its negation $\neg F$, or of the form $C(t_1)$, $\neg C(t_1)$, $R(t_1,t_2)$, $\neg R(t_1,t_2)$, $=(t_1,t_2)$, $\neq(t_1,t_2)$, where C is a concept, R is a role, t, t_1 and t_2 are terms (variables or constants).

The operators $\uplus$ and $\cup$ are used to extend the knowledge base $\mathcal{L}_i$ locally, with $S_k \uplus p_k$ (resp., $S_k \cup p_k$) increasing S_k (resp., $\neg S_k$) by the extension of p_k. Intuitively, the dl-atom above adds this information to $\mathcal{L}_i$ and then asks this knowledge base for the set of terms satisfying $Q(t)$.[2]

A *Multi Description Logic program* (Mdl-program) is a pair $\langle\{\mathcal{L}_1,\ldots,\mathcal{L}_n\},\mathcal{P}\rangle$ where: (1) each $\mathcal{L}_i$ is a description logic knowledge base; (2) $\mathcal{P}$ is a set of (normal) *Mdl-rules*, i.e. rules of the form

$$a \leftarrow b_1,\ldots,b_k, not\ b_{k+1},\ldots,not\ b_p$$

where a is a logic program atom and each b_j, for $1 \leq j \leq p$, is either a logic program atom or a dl-atom relative to $\{\mathcal{L}_1,\ldots,\mathcal{L}_n\}$. Note that $\mathcal{P}$ is a generalized logic program, so negation is the usual, closed-world, negation-as-failure. This is in contrast with the $\mathcal{L}_i$, which (being description logic knowledge bases) come with an open-world semantics.

The semantics of Mdl-programs is a straightforward generalization of the semantics for dl-programs (Eiter et al., 2008) and will not be discussed here, since it will not be needed explicitly. In particular, dl-programs can be seen as Mdl-programs with only one knowledge base.

On top of Mdl-programs, we define a useful syntactic construction. An *Mdl-program with observers* is a tuple $\langle\{\mathcal{L}_1,\ldots,\mathcal{L}_n\},\mathcal{P},\{\Lambda_1,\ldots,\Lambda_n\},\{\Psi_1,\ldots,\Psi_n\}\rangle$ where: (1) $\langle\{\mathcal{L}_1,\ldots,\mathcal{L}_n\},\mathcal{P}\rangle$ is an Mdl-program; (2) for $1 \leq i \leq n$, Λ_i is a finite set of pairs $\langle S,p\rangle$ where S is a concept, a role, or a negation of either, from $\mathcal{L}_i$ and p is a predicate from $\mathcal{P}$; (3) for $1 \leq i \leq n$, Ψ_i is a finite set of pairs $\langle p,S\rangle$ where p is a predicate from $\mathcal{P}$ and S is a concept, a role, or a negation of either, from $\mathcal{L}_i$. For each pair in Ψ_i or Λ_i, the arities of S and p must coincide. The sets $\Lambda_1,\ldots,\Lambda_n,\Psi_1,\ldots,\Psi_n$ will occasionally be referred to as the *observers* of

[1] The description logics underlying the $\mathcal{L}_i$s need not be the same.

[2] The precise semantics is a straightforward adaptation of (Eiter et al., 2008); the third operator therein introduced can be defined in terms of $\cup$, but not including it simplifies the semantics of Mdl-programs, as discussed in (Wang et al., 2005).

$\langle\{\mathcal{L}_1,\ldots,\mathcal{L}_n\},\mathcal{P}\rangle$. Intuitively, Λ_i contains concepts and roles in $\mathcal{L}_i$ that $\mathcal{P}$ needs to observe, in the sense that $\mathcal{P}$ should be able to detect whenever new facts about them are derived, whereas Ψ_i contains the predicates in $\mathcal{P}$ that $\mathcal{L}_i$ wants to observe. For simplicity, when we consider Mdl-programs with observers that only have one knowledge base, we will omit the braces and refer to them as dl-programs with observers.

The above Mdl-program with observers implicitly defines the Mdl-program $\left\langle\{\mathcal{L}_1,\ldots,\mathcal{L}_n\},\mathcal{P}^{\Psi_1,\ldots,\Psi_n}_{\Lambda_1,\ldots,\Lambda_n}\right\rangle$ where $\mathcal{P}^{\Psi_1,\ldots,\Psi_n}_{\Lambda_1,\ldots,\Lambda_n}$ is obtained from $\mathcal{P}$ by: (1) adding rule $p(X)\leftarrow DL_i[;S](X)$ for each $\langle S,p\rangle\in\Lambda_i$, if S is a concept (and its binary counterpart, if S is a role); and (2) in each dl-atom $DL_i[\chi;Q](t)$ (including those added in the previous step), adding $S\uplus p$ to χ for each $\langle p,S\rangle\in\Psi_i$ and $S\uplus p$ to χ for each $\langle p,\neg S\rangle\in\Psi_i$.

The next sections will discuss examples of Mdl-programs in detail.

We are currently working on implementing Mdl-programs with observers, using the `dlvhex` tool (Eiter et al., 2006b), with the goal of experimenting with the design patterns proposed herein.

We end this section with two remarks. The first one concerns the asymetry in the generalization from dl-programs to Mdl-programs. The reader might be wondering why we do not take this construction a step further, and define a system with several description logic knowledge bases and several logic programs. The answer lies in the asymetry already present in dl-programs themselves: the logic program is not only a component of the system, it is also the component responsible for interaction with the outside world. In particular, the dl-program's view of $\mathcal{L}$ is the view from each dl-atom – all changes made by $\mathcal{P}$ to the knowledge base only affect the logic program's view of it. Therefore, it is natural to consider several knowledge bases with a single logic program as the orchestrator (and where the outside world only "sees" that program's view of each knowledge base) rather than a more democratic system with several logic programs and several knowledge bases freely interacting with each other. This does not exclude that, in the future, it might be worth to study connecting patterns between two or more Mdl-programs (for instance, using a Mdl-program as a module of another Mdl-program, as a way to protect and restrict the access to certain knowledge bases).

A second aspect concerns the similarities and differences between Mdl-programs and multi-context systems. In both cases, we are faced with several independent components connected by a set of rules – a logic program, in the case of Mdl-programs, and the sets of bridge rules, in multi-context systems. However, all components of multi-context systems are observable from the outside world, since models of multi-context systems contain models of each individual component. It is also interesting to observe that the *flavor* of bridge rules is pretty much captured by the generalization to Mdl-programs with observers: the pairs in the observer sets achieve the effect of systematically extending a predicate, concept or role in one component with information from another component, in the same way bridge rules control the information flow within a multi-context system (Cruz-Filipe et al., 2013).

4 DESIGN PATTERNS FOR Mdl-programs

In this section, we present seven design patterns for Mdl-programs. These are divided in two categories: the three elementary design patterns are the building blocks for the four more complex ones. Together, these seven patterns form a powerful set from which quite complex programs can be designed in a more structured way, simplifying the programmer's task while at the same time yielding more flexible programs that are easier to maintain.

The presentation of each design pattern follows a similar scheme: we motivate the pattern by means of a simple scenario leading to the development of a well-designed program; from these we abstract a general design pattern, presented as a pair problem/solution within the context of an Mdl-program with observers $\langle\{\mathcal{L}_1,\ldots,\mathcal{L}_n\},\mathcal{P},\{\Lambda_1,\ldots,\Lambda_n\},\{\Psi_1,\ldots,\Psi_n\}\rangle$. In the next section, we develop a larger application, illustrating how the seven patterns work together, complementing each other.

4.1 Elementary Design Patterns

We now introduce the three basic design patterns for Mdl-programs.

4.1.1 Observing a Knowledge Base

We first consider the case when the logic program component systematically wants to import information from a knowledge base in order to define a predicate, keeping track of changes made to the relevant concept or role in its defining component.

Scenario. The National Zoo keeps an ontology containing information about all the animals currently

living in its premises, as well as several rules pertaining to the characteristics of the different species. This ontology $\mathcal{L}$ is used also by the company that provides food for all the zoo's feline inhabitants, coupled with a rule-based program $\mathcal{P}$ in a dl-program $\langle \mathcal{L}, \mathcal{P} \rangle$. In this program, it is necessary that $\mathcal{P}$ be constantly updated with the information about the current feline population of the zoo, hence the following rule was included.

$$\mathsf{feline}(X) \leftarrow DL[;\mathsf{Feline}](X) \qquad (1)$$

Note that feline in the head of the rule is a predicate from $\mathcal{P}$ whereas the Feline in the body is a concept from $\mathcal{L}$. No confusion can arise from this, since the only place where concepts or roles from $\mathcal{L}$ may occur in $\mathcal{P}$ is within dl-atoms.

The same program can be written as a dl-program with observers, namely $\langle \mathcal{L}, \mathcal{P}', \{\langle \mathsf{Feline}, \mathsf{feline} \rangle\}, \emptyset \rangle$, where $\mathcal{P}'$ is $\mathcal{P}$ without rule (1). In both cases, any changes made to Feline in the zoo's ontology will be automatically reproduced in $\mathcal{P}$ or $\mathcal{P}'$. Although the effect is the same, the latter option makes it very clear that feline in $\mathcal{P}'$ is an observer of its homonym from $\mathcal{L}$, making the global dl-program with observers easier to understand and maintain.

Pattern *Observer Down*.

Problem. A predicate p from $\mathcal{P}$ needs to be updated every time the extent (set of named individuals) of a concept or role S (of the same arity as p) in $\mathcal{L}_i$ is changed.

Solution. Add the pair $\langle S, p \rangle$ to Λ_i.

4.1.2 Dynamically Modifying the View of a Knowledge Base

A second scenario occurs when one of the description logics' functionality relies on the observation of a predicate from $\mathcal{P}$. Consider the following example.

Scenario. A sweets distributor provides the shops selling its products containing general information and reasoning about the different types of sweets on sale. Being a generic knowledge base, this ontology has no concrete facts: in particular, the extent of the predicate Sweet is empty. Each shop interacts with the central ontology by means of a particular dl-program $\mathcal{KB} = \langle \mathcal{L}, \mathcal{P} \rangle$, where $\mathcal{L}$ is the central ontology and $\mathcal{P}$ contains facts detailing the specific brands of sweets being sold in that particular location. In order for the program to function appropriately, Sweet (in $\mathcal{L}$) needs to be automatically updated whenever a new fact is added to the relevant predicate in $\mathcal{P}$, so that this fact can be taken into account.

The shop Sweets In Heaven uses a predicate forSale in $\mathcal{P}$ to store the information about the brands available at their shop. In order to use $\mathcal{L}$ to reason about these products, they need to replace all dl-atoms $DL[\chi; Q](t)$ in $\mathcal{P}$ with $DL[\mathsf{Sweet} \uplus \mathsf{forSale}, \chi; Q](t)$ meaning that $\mathcal{L}$ behaves as though its predicate Sweet was actually extended with all sweets for sale at Sweets In Heaven. This is exactly the same as replacing $\mathcal{KB}$ by the dl-program with observers $\langle \mathcal{L}, \mathcal{P}, \emptyset, \{\langle \mathsf{forSale}, \mathsf{Sweet} \rangle\} \rangle$.

Pattern *Observer Up*.

Problem. A concept or role S from $\mathcal{L}_i$ needs to be updated every time the extent of a predicate p (of the same arity as S) in $\mathcal{P}$ is changed.

Solution. Add the pair $\langle p, S \rangle$ to Ψ_i.

4.1.3 Closing the World

The third building block addresses a very typical situation in ontology usage: in order for a concept or role to work as expected, it should be given closed-world semantics.

Scenario. A digital library has an ontology with information about all the titles it contains. One of the concepts it defines is ForSale, applicable to electronic books that are available for download for a price. However, description logics are not able to perform closed-world reasoning, so a query to the ontology about $\neg$ForSale is not sufficient to inform a user that a given title is *not* for sale.

With this in mind, the software developers working for the digital library integrated the ontology $\mathcal{L}$ in the framework of a dl-program $\langle \mathcal{L}, \mathcal{P} \rangle$, where P is a very simple program containing only the following rules.

$$\mathsf{forSale}(X) \leftarrow DL[;\mathsf{ForSale}](X)$$
$$\mathsf{notForSale}(X) \leftarrow \mathit{not}\ \mathsf{forSale}(X)$$

The information about titles that are not for sale is now directly available in the predicate notForSale of $\mathcal{P}$. Furthermore, if the library later wants to extend $\mathcal{P}$ further, then all dl-atoms must be of the form $DL[\mathsf{ForSale} \uplus \mathsf{notForSale}, \chi; Q](t)$ – thus querying $\mathcal{L}$ extended with the desired closed-world interpretation of ForSale. For this reason, it is much safer to write the resulting program as the dl-program with observers $\langle \mathcal{L}, \mathcal{P}, \{\langle \mathsf{ForSale}, \mathsf{forSale} \rangle\}, \{\langle \mathsf{notForSale}, \neg\mathsf{ForSale} \rangle\} \rangle$, where $\mathcal{P}$ contains only the second of the above rules.

Pattern *Closed-world*.

Problem. The concept (or role) S from $\mathcal{L}_i$ should follow closed-world semantics.

Solution.

Choose predicate symbols s^+, s^- not used in $\mathcal{P}$. Add $\langle S, s^+ \rangle$ to Λ_i, $\langle s^-, \neg S \rangle$ to Ψ_i, and $s^-(X) \leftarrow not\ s^+(X)$ to $\mathcal{P}$.

4.2 Derived Design Patterns

The examples presented thus far should suffice at this stage to give the user a feel for dl-programs. We now present a second set of general-purpose design patterns that can be seen as organized combinations of the previous ones, but are also useful as components of more complex patterns. The usage of this design patterns is exemplified in the next section.

4.2.1 Flexible Definitions

The next design pattern allows one to define a predicate in $\mathcal{P}$ abstracting from how it is represented in the knowledge bases.

Scenario. Vineyards are a major tourist atraction of Wineland. Knowing this, the country's Tourist Information Office has decided to combine information from the three major regional wine producers' associations in an Mdl-program $\mathcal{KB} = \langle \{\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3\}, \mathcal{P} \rangle$, where the $\mathcal{L}_i$s are the knowledge bases containing information about the wines produced in each region, by means of which tourists can access the information present in all three.

Since the three knowledge bases were developed independently, the need arose for a predicate unifying the concept of wine, which is distributed among them. Therefore, $\mathcal{P}$ needs to import all the information from the corresponding predicates wine from $\mathcal{L}_1$, wineBrand from $\mathcal{L}_2$ and product from $\mathcal{L}_3$.

To unify these three concepts in a single predicate in $\mathcal{P}$, the Tourist Information Office included the three following rules in its program.

$$\mathsf{wine}(X) \leftarrow DL_1[;\mathsf{wine}](X)$$
$$\mathsf{wine}(X) \leftarrow DL_2[;\mathsf{wineBrand}](X)$$
$$\mathsf{wine}(X) \leftarrow DL_3[;\mathsf{product}](X)$$

This amounts to rewriting the Mdl-program $\mathcal{KB}$ as the Mdl-program with observers $\langle \{\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3\}, \mathcal{P}, \{\Lambda_1, \Lambda_2, \Lambda_3\}, \{\emptyset, \emptyset, \emptyset\} \rangle$ where:

$$\Lambda_1 = \{\langle \mathsf{wine}, \mathsf{wine} \rangle\}$$
$$\Lambda_2 = \{\langle \mathsf{wineBrand}, \mathsf{wine} \rangle\}$$
$$\Lambda_3 = \{\langle \mathsf{product}, \mathsf{wine} \rangle\}$$

Pattern *Polymorphic Entities*.

Problem. In $\mathcal{P}$ there is a predicate p whose instances are inherited from concepts or roles $S_1, \ldots, S_k$ where each S_j comes from the knowledge base $\mathcal{L}_{\varphi(j)}$, for $1 \leq j \leq n$.

Solution. For each $1 \leq j \leq n$, add the pair $\langle S_j, p \rangle$ to $\Lambda_{\varphi(j)}$.

Note that POLYMORPHIC ENTITIES consists of a combined application of several OBSERVER DOWN, all with the same observer predicate in $\mathcal{P}$.

This pattern captures the essences of the Polymorphism and Dynamic Binding patterns from object-oriented programming (Larman, 2004): these patterns deal (at different levels) with definitions that are kept separate from their usage. In POLYMORPHIC ENTITIES, we deal with a predicate that is kept as independent as possible from its definition. Instead of defining clauses, the instances are plugged in through the use of Mdl-programs with observers, thus externalizing the definition of the predicate – in the spirit of Dynamic Binding. The possibility of using different concepts or roles (possibly even from different knowledge bases) captures the essence of Polymorphism.

4.2.2 Interconnecting Description Logics

Another variant of the Observer design pattern occurs when a description logic's functionality relies on the observation of a predicate in a *different* description logic; this can be achieved by combining both the OBSERVER UP and OBSERVER DOWN patterns, thus making the logic program $\mathcal{P}$ a mediator. A particular case arises when an ontology designed primarily for reasoning interacts with a knowledge base that is mostly about particular instances. In practice, this design pattern appears mostly in combination with DEFINITIONS WITH HOLES, so we will postpone the presentation of a concrete scenario to the next step.

Pattern *Transversal Observer*.

Problem. A concept (or role) S from $\mathcal{L}_i$ needs to be updated every time the extent of a concept (resp. role) R from $\mathcal{L}_j$ is changed (with $i \neq j$).

Solution. Choose a predicate symbol p not used in $\mathcal{P}$. Add $\langle R, p \rangle$ to Λ_j and $\langle p, S \rangle$ to Ψ_i.

Note that TRANSVERSAL OBSERVER consists of a

combined application of OBSERVER UP together with OBSERVER DOWN, relative to two different description logics.

4.2.3 Underspecification

In a modular system, it makes perfect sense to use concepts whose definition is left to other components. This is the motivation for the next design pattern.

Scenario. A company is designing an ontology to help with school management. One of the rules included in the ontology is

$$\mathsf{canAskDiplomma} \sqsubseteq \mathsf{graduated} \sqcap \neg \mathsf{onHold},$$

stating that a student can ask for his diplomma after graduating if he has no pending issues with the school. Although graduated is a predicate defined in this ontology, the company has no way of knowing *a priori* which criteria to use to decide whether a student still has pending issues with the school, as this varies from school to school.

Therefore, the ontology will not have any information about instances of onHold – the definition of this concept will be left to other components that will be integrated with the ontology through an Mdl-program via OBSERVER or POLYMORPHIC ENTITIES.

Pattern *Definitions with Holes*.

Problem. In $\mathcal{L}_i$ there is a concept or role S needed for reasoning but its definition will be in $\mathcal{L}_j$ (with $i \neq j$) or $\mathcal{P}$.

Solution.

Use S in $\mathcal{L}_i$ without defining it (so the extent of S is empty).

Later, connect S to its definition using OBSERVER UP, OBSERVER DOWN or TRANSVERSAL OBSERVER, possibly coupled with POLYMORPHIC ENTITIES.

In our example, suppose the company's ontology $\mathcal{L}_1$ is integrated in an Mdl-program $\langle\{\mathcal{L}_1, \mathcal{L}_2\}, \mathcal{P}\rangle$, where $\mathcal{L}_2$ is the school's own knowledge base (essentially a database containing information about the school's students) and $\mathcal{P}$ contains rules connecting both and performing other reasoning tasks. In that school, a student is *onHold* if he is owing money to the school (stored in $\mathcal{L}_2$ by means of the concept inDebt); hence, one can apply TRANSVERSAL OBSERVER to define onHold as an observer of inDebt. In a different, elite, school, the same concept is implemented differently, since the school forces its students to get extra credit by attending at least five seminars; so here onHold is an OBSERVER of $(\leq_4 \mathsf{attendedSeminar}) \sqcup \mathsf{inDebt}$. Note that, in this situation, the school's knowledge base would need to provide an auxiliary concept equivalent to this condition due to the restrictions on the syntax of dl-queries.

This pattern corresponds to the Template Method pattern of object-oriented programs (Gamma et al., 1995), and to the Programming with Holes technique of (Meyer, 1997).

In the next section we will show an example where the holes are filled in by resorting to POLYMORPHIC ENTITIES.

4.2.4 Distributed Definitions

The last design pattern in this section applies when a predicate's definition is split between different components of the Mdl-program.

Scenario. A company sells cars in several dealers across town. Each dealer has access to the company's car knowledge base $\mathcal{L}$, which specifies several properties of the available models and the dealers which sell them. One of the dealers with the company has his own information stored as a rule-based program $\mathcal{P}$. To integrate both systems, he uses a dl-program; however, he still needs to share some concepts between both components. For example, the concept car, whose instances include all cars centrally available for sale, is defined in $\mathcal{L}$, but $\mathcal{P}$ has information about second-hand cars that occasionally go through the store, stored in a predicate secondHandCar and which the dealer wants to reason about in his view of $\mathcal{L}$; $\mathcal{P}$ also has information about motor bikes, that also occasionally go through the store, stored in a predicate motorBike, and which should be considered distinct from cars, when reasoning in $\mathcal{P}$'s view of $\mathcal{L}$. In order to achieve full integration between the two components, he added two new predicates car^+ and car^- to $\mathcal{P}$, connecting them to his predicates by means of the two rules

$$\mathsf{car}^+(X) \leftarrow \mathsf{secondHandCar}(X)$$
$$\mathsf{car}^-(X) \leftarrow \mathsf{motorBike}(X)$$

and with car from $\mathcal{L}$ via the two rules

$$\mathsf{car}^+(X) \leftarrow DL[;\mathsf{car}](X)$$
$$\mathsf{car}^-(X) \leftarrow DL[;\neg\mathsf{car}](X).$$

Furthermore, every dl-atom $DL[\chi;Q](t)$ needs to be replaced by $DL[\mathsf{car} \uplus \mathsf{car}^+, \mathsf{car} \uplus \mathsf{car}^-, \chi; Q](t)$. In this way, one can use positive and negative facts about cars both in $\mathcal{L}$ and in $\mathcal{P}$.

Using Mdl-programs with observers, the same construction can be designed in a simpler way: the

only rules one needs to include in $\mathcal{P}$ are the two first ones (relating car^+ with secondHandCar and car^- with motorBike), while the rest is achieved by taking $\Lambda = \{\langle \mathsf{car}, \mathsf{car}^+\rangle, \langle \neg\mathsf{car}, \mathsf{car}^-\rangle\}$ and $\Psi = \{\langle \mathsf{car}^+, \mathsf{car}\rangle, \langle \mathsf{car}^-, \neg\mathsf{car}\rangle\}$.

Pattern *(Named) Lifting*.

Problem. The definition of a predicate should be distributed among some of the $\mathcal{L}_i$s (in the form of concepts or roles S_i) and $\mathcal{P}$ (in the form of two predicates p^+ and p^-, corresponding to the predicate and its negation).

Solution.

For each i, add $\langle S_i, p^+\rangle$ and $\langle \neg S_i, p^-\rangle$ to Λ_i.
For each i, add $\langle p^+, S_i\rangle$ and $\langle p^-, \neg S_i\rangle$ to Ψ_i.

Note that LIFTING is essentially different from OBSERVER: in OBSERVER, a predicate is defined in *one* component and used in others; in LIFTING, not only the usage, but also the *definition* of the predicate is split among several components, so that one must look at the whole Mdl-program to understand it. This is also part of the reason to include the negations of the predicates involved in the observers: the distributed predicate must end up with the same semantics, both in $\mathcal{P}$ and in all the involved $\mathcal{L}_i$s – at least regarding named individuals.

It is possible to apply LIFTING when $\mathcal{P}$ does not participate in the predicate's definition. In this case, $\mathcal{P}$ is simply a mediator, and p^+ and p^- can be any fresh predicate names.

Note that the application of each of the patterns proposed in this section yields localized changes to the Mdl-program: they consist of either changing dl-atoms (by means of adding pairs to Ψ_i) or adding rules to $\mathcal{P}$ (either directly, as in the case of CLOSED-WORLD, or by adding pairs to Λ_i). In all cases, these changes are only reflected in $\mathcal{P}$, and they can be divided into two or three distinct types. This is in line with the whole philosophy of dl-programs: there is an asymmetry between their components where the logic program is the orchestrator between all components as well as its façade: it is the only entity interacting with the outside world.

5 A COMPREHENSIVE EXAMPLE

We now illustrate the usage of the different design patterns introduced so far by means of a more complex example.

Scenario. The software developers at WISHYOU-WERETHERE travel agency decided to develop an Mdl-program to manage several of the agency's day-to-day tasks. Currently, WISHYOUWERETHERE has two active partnerships, one with an aviation company, another with a hotel chain. Thus, the Mdl-program to be developed uses three ontologies:

- $\mathcal{L}_A$ is a generic accounting ontology for travel agencies, which is commercially available, and which contains all sorts of rules relating concepts relevant for the business. This ontology is strictly terminological, containing no specific instances of its concepts and roles.
- $\mathcal{L}_F$ is the aviation partner's knowledge base, containing information not only about available flights between different destinations, but also about clients who have already booked flights with that company.
- $\mathcal{L}_H$ is a similar knowledge base pertaining to the hotels owned by the partner hotel chain.

One of the points to take into consideration is that the resulting Mdl-program with observers $\langle\{\mathcal{L}_A, \mathcal{L}_F, \mathcal{L}_H\}, \mathcal{P}, \{\Lambda_A, \Lambda_F, \Lambda_H\}, \{\Psi_A, \Psi_F, \Psi_H\}\rangle$ should be easily extended so that the travel agency can establish new partnerships, in particular with other aviation companies and hotel chains, as long as those provide their own knowledge bases. At the end of this section, we will show how the systematic use of design patterns and observers helps towards achieving this goal.

By establishing partnerships, WISHYOUWERE-THERE's client basis is extended with all the clients who have booked services of its partners. In this way, promotions made available by either partner are automatically offered to every partner's clients, as long as the bookings are made through the travel agency. In return, the partners get publicity and more clients, since a person may be tempted to fly with their company or book their hotel due to these promotions, thereby becoming also their client.

Updating the Client Database. Ensuring that each partner's clients automatically become WISHYOU-WERETHERE's clients can be achieved by noting that this is exactly the problem underlying OBSERVER DOWN. Assuming $\mathcal{L}_F$ and $\mathcal{L}_H$ have concepts Flyer and Guest, respectively, identifying their clients, and that the agency's clients will be stored as a predicate client in $\mathcal{P}$, all that needs to be done is to register client as an observer of Flyer and Guest, which, according to the pattern, is achieved by ensuring that $\langle \mathsf{Flyer}, \mathsf{client}\rangle \in \Lambda_F$ and $\langle \mathsf{Guest}, \mathsf{client}\rangle \in \Lambda_H$.

Identifying Pending Payments. The designers of $\mathcal{L}_A$ resorted intensively to DEFINITIONS WITH HOLES, since many of the concepts they use can only be defined in the presence of a concrete client database. In particular, $\mathcal{L}_A$ contains a role toPay, about which it contains no membership axioms. The information about the specific purchases a client has made and not paid so far must be collected from the partners' knowledge bases, $\mathcal{L}_F$ and $\mathcal{L}_H$.

There are two ways of completing this definition. The more direct one stems from noting that toPay should be an observer of adequate roles in $\mathcal{L}_F$ and $\mathcal{L}_H$. We will assume that these roles are payFlight and payHotel. Applying twice TRANSVERSAL OBSERVER (which is the adequate pattern), one needs to ensure that

$$\langle \mathsf{payFlight}, \mathsf{toPayF} \rangle \in \Lambda_F \quad \langle \mathsf{toPayF}, \mathsf{toPay} \rangle \in \Psi_A$$
$$\langle \mathsf{payHotel}, \mathsf{toPayH} \rangle \in \Lambda_H \quad \langle \mathsf{toPayH}, \mathsf{toPay} \rangle \in \Psi_A .$$

The major drawback of this solution is that it requires adding two dummy predicates to $\mathcal{P}$ whose only purpose is to serve as go-between from both knowledge bases to $\mathcal{L}_A$. An alternative solution is to create a single auxiliary predicate toPay in $\mathcal{P}$ and make toPay from $\mathcal{L}_A$ an observer of this predicate applying OBSERVER UP. In turn, we use the POLYMORPHIC ENTITY pattern to connect toPay to payFlight and payHotel. The resulting Mdl-program with observers is such that:

$$\langle \mathsf{payFlight}, \mathsf{toPay} \rangle \in \Lambda_F$$
$$\langle \mathsf{payHotel}, \mathsf{toPay} \rangle \in \Lambda_H \quad \langle \mathsf{toPay}, \mathsf{toPay} \rangle \in \Psi_A .$$

As we will discuss later, this solution will also simplify the process of adding new partners to the agency.

Offering Promotions. WISHYOUWERETHERE offers a number of promotions to its special clients. For example, in February the agency offers them a 20% discount on all purchases. Because of the partnership, the concept of special client is distributed among all partners: a client is a special client if it fulfills one of the partners' requirements – e.g. having traveled some number of miles with the airline partner, or booked a family holiday in one of the partner's hotels, or bought one of the agency's pricey packages. The partnership protocol requires that each knowledge base provide a concept identifying which clients are eligible for promotions, so that the partners can change these criteria without requiring WISHYOUWERETHERE to change its program.

This is a situation where the LIFTING design pattern applies. Assuming that $\mathcal{L}_F$ uses TopClient for its special clients, $\mathcal{L}_H$ uses Gold and $\mathcal{P}$ defines special, these three predicates are given the same semantics through LIFTING. Intuitively, this means that, in the Mdl-program's view, all three concepts equally denote *all* special clients, regardless of where they originate. The application of the pattern translates to

$$\langle \mathsf{TopClient}, \mathsf{special} \rangle \in \Lambda_F$$
$$\langle \mathsf{special}, \mathsf{TopClient} \rangle \in \Psi_F$$
$$\langle \neg\mathsf{TopClient}, \mathsf{notSpecial} \rangle \in \Lambda_F$$
$$\langle \mathsf{notSpecial}, \neg\mathsf{TopClient} \rangle \in \Psi_F$$

and four similar observers in Λ_H and Ψ_H, with Gold in place of TopClient.

Furthermore, in order to determine whether a particular client is entitled to promotions, it is useful to give closed-world semantics to these predicates. Since they are all equivalent, we can do this very simply in $\mathcal{P}$ by adding the rule

$$\mathsf{notSpecial}(X) \leftarrow \mathit{not}\ \mathsf{special}(X) .$$

Note that we did not need to apply the CLOSED-WORLD pattern because special is a predicate from $\mathcal{P}$, where the semantics is closed-world: the application of LIFTING ensures that Gold and TopClient, being equivalent, also have closed-world semantics.

In order for one of the partner companies to make its clients eligible for special promotions, its ontology just needs to contain inclusion axioms partially characterizing special clients. For example, $\mathcal{L}_F$ might contain the rule $\exists \mathsf{flies}.\mathsf{10000OrMore} \sqsubseteq \mathsf{TopClient}$, or $\mathcal{L}_H$ might specify that $\exists \mathsf{hasCompleted}.\mathsf{FamilyBooking} \sqsubseteq \mathsf{Gold}$, or $\mathsf{special}(X) \leftarrow \mathsf{booked}(X,Y), \mathsf{expensive}(Y)$ might be a rule in $\mathcal{P}$.

A subtle issue now appears regarding the consistency problems that may arise from the use of the LIFTING pattern. Since this pattern identifies concepts from different knowledge bases, it does not *a priori* guarantee that the resulting knowledge bases are consistent. In particular, if one of the partners grants special status to a client and another denies this status to the same client, an inconsistency will arise. More sophisticated variations of the LIFTING pattern can be developed to detect and avoid this kind of situation, but such a discussion is beyond the scope of this presentation.

An example of a promotion offered by WISHYOUWERETHERE to special clients would be

$$\mathsf{20\%Discount}(X) \leftarrow \mathsf{special}(X) .$$

All special clients will benefit from this discount, regardless of who (the travel agency, the hotel partner or the aviation company) decided that they should be special clients. However, in some cases partners may want to deny their promotions to particular clients. For example, the aviation company is offering 100

bonus miles to special costumers booking a flight on a Tuesday, but this promotion does not apply to its workers. In order to allow this kind of situation, partners may define a dedicated concept identifying the non-eligible clients. Since all clients external to that partner are automatically eligible, this concept needs to have closed-world semantics so that (in our example) $\mathcal{L}_F$ can include the rules

$$\mathsf{100BonusMilesWinner} \sqsubseteq \mathsf{TopClient} \sqcap \neg\mathsf{Blocked}$$
$$\mathsf{Worker} \sqsubseteq \mathsf{Blocked}$$

still giving the promotion to all clients from the other partners. Although each knowledge base can enforce this semantics in its domain, in order to extend it to other clients the CLOSED-WORLD pattern must be applied, so we will have

$$\langle \mathsf{Blocked}, \mathsf{blockedF} \rangle \in \Lambda_F$$
$$\langle \mathsf{nonBlockedF}, \neg\mathsf{Blocked} \rangle \in \Psi_F$$
$$\mathsf{nonBlockedF}(X) \leftarrow \mathit{not}\ \mathsf{blockedF}(X) \in \mathcal{P}$$

Suppose that airline employee Ann qualifies for WISHYOUWERETHERE promotions because she spent three weeks in Jamaica with her husband and their five children, hence Gold(Ann) holds in $\mathcal{L}_H$ and therefore Ann is a special client. She is therefore eligible for WISHYOUWERETHERE's promotions, but she will still not earn the bonus miles because it is $\mathcal{L}_F$ who decides whether someone gets that particular promotion, and even though TopClient(Ann) holds that knowledge base will not return 100BonusMilesWinner(Ann). However, she will earn the 20%Discount, since it is offered directly by WISHYOUWERETHERE.

Adding New Partnerships. We now discuss briefly how new partners can be easily added to the system later on, as this illustrates quite well the advantages of working both with design patterns and in the context of Mdl-programs with observers.

Summing up what we have so far relating to the partnerships, the sets $\Lambda_F, \Lambda_H, \Psi_F$ and Ψ_H are:

Λ_F: $\langle \mathsf{Flyer}, \mathsf{client} \rangle$
$\langle \mathsf{payFlight}, \mathsf{toPay} \rangle$
$\langle \mathsf{TopClient}, \mathsf{special} \rangle$
$\langle \neg\mathsf{TopClient}, \mathsf{notSpecial} \rangle$
$\langle \mathsf{Blocked}, \mathsf{blockedF} \rangle$

Λ_H: $\langle \mathsf{Guest}, \mathsf{client} \rangle$
$\langle \mathsf{payHotel}, \mathsf{toPay} \rangle$
$\langle \mathsf{Gold}, \mathsf{special} \rangle$
$\langle \neg\mathsf{Gold}, \mathsf{notSpecial} \rangle$
$\langle \mathsf{Blocked}, \mathsf{blockedH} \rangle$

Ψ_F: $\langle \mathsf{special}, \mathsf{TopClient} \rangle$
$\langle \mathsf{notSpecial}, \neg\mathsf{TopClient} \rangle$
$\langle \mathsf{nonBlockedF}, \neg\mathsf{Blocked} \rangle$

Ψ_H: $\langle \mathsf{special}, \mathsf{Gold} \rangle$
$\langle \mathsf{notSpecial}, \neg\mathsf{Gold} \rangle$
$\langle \mathsf{nonBlockedH}, \neg\mathsf{Blocked} \rangle$

Also, the application of the design patterns added the following rules to $\mathcal{P}$.

$$\mathsf{nonBlockedF}(X) \leftarrow \mathit{not}\ \mathsf{blockedF}(X)$$
$$\mathsf{nonBlockedH}(X) \leftarrow \mathit{not}\ \mathsf{blockedH}(X)$$
$$\mathsf{notSpecial}(X) \leftarrow \mathit{not}\ \mathsf{special}(X)$$

The similarity between Λ_F and Λ_H, and between Ψ_F and Ψ_H, is a clear illustration of the changes required when future partners of WISHYOUWERETHERE are added to the system. Furthermore, the names they use for each concept or role are not relevant – they just need to indicate how they identify their clients, their clients' debts, their special clients, and the clients they wish to exclude from their promotions.

6 BEYOND THESE PATTERNS: LIMITATIONS AND FUTURE WORK

We have shown so far how the ideas of design patterns can be applied to Mdl-programs, yielding general principles for this programming paradigm. In this section we explore some limitations and discuss future directions for our work.

In many contexts, a component of a system may not be known or available at the time of implementation of others, yet it is necessary to query it. A way to get around this is to use a prototype knowledge base that will later on be connected to the concrete component in a straightforward way. The same problem also arises if one wishes to be able to replace a knowledge base with another with a similar purpose, but whose concept and role names may be different. This can be achieved by means of the following pattern.

Pattern *(Straight) Adapter*.

Problem. One wants to work with $\mathcal{L}_k$ independently of its particular syntax.

Solution. Add an empty interface knowledge base $\mathcal{L}_I$ to $\mathcal{KB}$ using the desired concept and role names. Connect each concept and role in $\mathcal{L}_I$ with its counterpart in $\mathcal{L}_k$ by means of an application of the TRANSVERSAL OBSERVER pattern.

There is one important characteristic of this implementation that distinguishes it from the usual Adapter

design pattern: the dl-program syntax for local extensions to dl-queries only works in the particular case where the query is over a concept or role being directly extended. Because all queries go through the interface knowledge base, where no axioms exist, any other extensions are lost.

Consider a simple example where the interface $\mathcal{L}_I$ specifies two concepts P and Q, which are made concrete in $\mathcal{L}_C$ as A and B. Furthermore, $\mathcal{L}_C$ also contains the inclusion axiom $A \sqsubseteq B$. Finally, $\mathcal{P}$ contains the single fact $\mathsf{thisIsTrue}(\mathsf{ofMe})$. In $\mathcal{P}$, the direct query $DL_C[A \uplus \mathsf{thisIsTrue}; B](X)$ would return the answer $X = \mathsf{ofMe}$, since $\mathcal{L}_C$ is extended with $A(\mathsf{ofMe})$ in the context of this query. However, the corresponding indirect query (i.e. the same query, but passing through the adapter)

$$DL_I[P \uplus p^+, P \uplus p^-, Q \uplus q^+, Q \uplus q^-, P \uplus \mathsf{thisIsTrue}; Q](X)$$

after extending $\mathcal{P}$ with the rules

$$p^+(X) \leftarrow DL_C[;A](X) \qquad q^+(X) \leftarrow DL_C[;B](X)$$
$$p^-(X) \leftarrow DL_C[;\neg A](X) \qquad q^-(X) \leftarrow DL_C[;\neg B](X)$$

as introduced by the concretization of the observer sets would still return no answer, since $\mathcal{L}_I$ only knows the facts about B that are directly given by $\mathcal{L}_C$ through q^+.

Note, however, that the dl-atom $DL_C[A \uplus \mathsf{thisIsTrue}; A](X)$ *is* equivalent to

$$DL_I[P \uplus p^+, P \uplus p^-, Q \uplus q^+, Q \uplus q^-, P \uplus \mathsf{thisIsTrue}; P](X),$$

since the query is directly on the concept whose extent was altered. In practice, this is a common enough situation that this issue is less restrictive than it may seem. This is a restriction with respect to the full power of dl-programs; but the authors see this as a *feature* of the STRAIGHT ADAPTER design pattern. Should a context arise where such flexibility is essential, then this is not the right design pattern to apply.

A more complex problem arises with the PROXY design pattern. This pattern is used when one wants to control or restrict access to a resource, for example a database containing sensitive information. In practice, this is not very different from the ADAPTER design pattern – but ADAPTER is an algorithm-free pattern that just defines interfaces, whereas an entity implementing PROXY is expected to do some processing before passing on the information it receives.

If we attempt an implementation along the lines we have followed so far, it would be natural to explore the possibility of a proxy knowledge base to serve as a mediator between two components. In the setting of dl-programs, this is actually not possible to achieve directly, since all queries must go through the logical program. The only other option is to encode the proxy in the logic program itself, forcing every dl-query to the protected resource to be immediately preceded by some atoms implementing the proxy – which from the PROXY design pattern perspective is not completely satisfactory. There would be ways to go around this problem, namely by defining appropriate syntactic constructions, which we intend to pursue in future extensions of this work.

7 CONCLUSIONS

We proposed Mdl-programs as a generalization of dl-programs to systems that connect several description logics by means of a logic program. On top of these, we defined a syntactic construction (observers) that supports communication between the logic program and the different knowledge bases in a structured way, allowing one to externalize several communication aspects and focus on the development of the algorithmic parts of the program.

The original motivation for defining Mdl-programs with observers was to achieve global changes to the knowledge bases in the logic program's view of them, by guaranteeing that *all* dl-queries were appropriately extended, *even the ones that were written after deciding that a concept or role should be observing a predicate*. As we showed, this construction is powerful enough to allow for elegant and straightforward implementations of several well-known design patterns. By defining other adequate syntactic constructions, other design patterns may become easily accessible. In fact, it has been argued that the study of design patterns can be a key ingredient in deciding what syntactic extensions should be added to specific programming languages (Gibbons, 2006). In this spirit, we believe that we have more than justified the inclusion of observers as a syntactic tool in the context of Mdl-programs.

Another aspect that will have to be discussed relates to the practical issues of the usage of design patterns. *Ad hoc* solutions to specific problems may be more efficient than the application of systematic methods, but they tend to yield less generalizable and less extensible software applications. On the other hand, the use of observers introduces higher complexity, especially when non-stratified negation (which occurs in the examples discussed earlier) is involved. It is important to have a precise understanding of the compromise between efficiency and quality obtained by a systematic use of design patterns. We are working on building a prototype implementation of Mdl-programs within `dlvhex` tool (Eiter et al., 2006b) as a first step towards this study.

The purpose of the present study, at this stage, is to show that design patterns also have a place in the world of the Semantic Web. One can foresee a future where there is a widespread usage of systems combining description logics with rules, and the availability of systematic design methodologies is a key ingredient to making this future a reality. This paper is by necessity limited in its scope, and it is the authors' intent to explore how more sophisticated design patterns could be applied to Mdl-programs.

It would also be interesting to explore how the mechanisms herein discussed can be applied to multi-context systems, in view of the similarities between these and Mdl-programs. A preliminary study of this connection has been undertaken in (Cruz-Filipe et al., 2013).

ACKNOWLEDGEMENTS

This work was partially supported by Fundação para a Ciência e Tecnologia under contract PEst-OE/EEI/UI0434/2011.

REFERENCES

Adams, M., Coplien, J., Gamoke, R., Hanmer, R., Keeve, F., and Nicodemus, K. (1996). Fault-tolerant telecommunication system patterns. In *Pattern Languages of Program Design 2*, pages 549–562. Addison–Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Antoy, S. and Hanus, M. (2002). Functional logic design patterns. In Hu, Z. and Rodríguez-Artalejo, M., editors, *Proceedings of FLOPS 2002*, volume 2441 of *LNCS*, pages 67–87. Springer.

Brewka, G. and Eiter, T. (2007). Equilibria in heterogeneous nonmonotonic multi-context systems. In *Proceedings of AAAI-07*, pages 385–390. AAAI Press.

Bruijn, J. d., Ehrig, M., Feier, C., Martíns-Recuerda, F., Scharffe, F., and Weiten, M. (2006). Ontology mediation, merging, and aligning. In Davies, J., Studer, R., and Warren, P., editors, *Semantic Web Technologies: Trends and Research in Ontology-based Systems*. John Wiley & Sons, Ltd, Chichester, UK.

Cruz-Filipe, L., Henriques, R., and Nunes, I. (2013). Viewing dl-programs as multi-context systems. Technical Report 2013;05, Faculty of Sciences, University of Lisbon. http://hdl.handle.net/10455/6895.

Eiter, T., Ianni, G., Lukasiewicz, T., and Schindlauer, R. (2011). Well-founded semantics for description logic programs in the semantic Web. *ACM Transactions on Computational Logic*, 12(2). Article 11.

Eiter, T., Ianni, G., Lukasiewicz, T., Schindlauer, R., and Tompits, H. (2008). Combining answer set programming with description logics for the semantic web. *Artificial Intelligence*, 172(12–13):1495–1539.

Eiter, T., Ianni, G., Schindlauer, R., and Tompits, H. (2006a). Effective integration of declarative rules with external evaluations for semantic-web reasoning. In Sure, Y. and Domingue, J., editors, *Proceedings of ESWC 2006*, volume 4011 of *LNCS*, pages 273–287. Springer.

Eiter, T., Ianni, G., Schindlauer, R., and Tompits, H. (2006b). Towards efficient evaluation of HEX programs. In Dix, J. and Hunter, A., editors, *Proceedings of NMR-2006, Answer Set Programming Track*, pages 40–46. Institut für Informatik, TU Clausthal, Germany.

Erl, T. (2009). *SOA Design Patterns*. Prentice Hall, New York.

Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. Addison–Wesley.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison–Wesley.

Gangemi, A. and Presutti, V. (2009). Ontology design patterns. In Staab, S. and Studer, R., editors, *Handbook on Ontologies*, International Handbooks on Information Systems, pages 221–243. Springer. 2nd edition.

Gibbons, J. (2006). Design patterns as higher-order datatype-generic programs. In Hinze, R., editor, *Proceedings of WGP 2006*, pages 1–12. ACM.

Grau, B., Parsia, B., and Sirin, E. (2005). Combining OWL ontologies using e-connections. *Journal of Web Semantics*, 4(1):40–59.

Kifer, M. and Boley (eds.), H. (2010). RIF overview. W3C Working Group Note, http://www.w3.org/TR/2010/NOTE-rif-overview-20100622/.

Larman, C. (2004). *Applying UML and Patterns*. Prentice–Hall. 3rd Edition.

Mattson, T., Sanders, B., and Massingill, B. (2005). *Patterns for Parallel Programming*. Addison–Wesley.

Meyer, B. (1997). *Object-Oriented Software Construction*. Prentice–Hall. 2nd Edition.

Norvig, P. (1996). Design patterns in dynamic programming. Tutorial slides presented at Object World, Boston, MA, May 1996, available at http://norvig.com/design-patterns/.

Oliveira, B. and Gibbons, J. (2005). TypeCase: a design pattern for type-indexed functions. In Leijen, D., editor, *Proceedings of Haskell 2005*, pages 98–109. ACM.

Schmidt, D., Stal, M., Rohnert, H., and Buschmann, F. (2000). *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*. John Wiley & Sons.

Sterling, L. (2002). Patterns for Prolog programming. In Kakas, A. and Sadri, F., editors, *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part I*, volume 2407 of *LNCS*, pages 374–401. Springer.

Wang, K., Antoniou, G., Topor, R., and Sattar, A. (2005). Merging and aligning ontologies in dl-programs. In Adi, A., Stoutenburg, S., and Tabet, S., editors, *Proceedings of RuleML 2005*, volume 3791 of *LNCS*, pages 160–171. Springer.