Design Patterns for Description-Logic Programs

L. Cruz-Filipe[1,3,4], G. Gaspar[2,4], and I. Nunes[2,4]

[1] Escola Superior Náutica Infante D. Henrique, Paço d'Arcos, Portugal
[2] Faculdade de Ciências da Universidade de Lisboa, Portugal
[3] Centro de Matemática e Aplicações Fundamentais, Lisboa, Portugal ‡
[4] LabMag, Lisboa, Portugal ‡
lcfilipe@gmail.com, {gg,in}@di.fc.ul.pt

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 present a set of design patterns for Mdl-programs, a hybrid formalism combining several description logic knowledge bases via a logic program. 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 also discuss some limitations of this formalism, examining some usual patterns in other programming paradigms that have no parallel in Mdl-programs.

1 Introduction

In the mid-nineties, the Gang of Four's work on software design patterns [13] 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. [1, 11, 12, 19–21, 23], establishing a kind of common language between development teams, which substantially enriches their communication, and hence the whole design process.

Despite their widespread usage in the object-oriented paradigm, on which a lot of the work has been focused, effort has also been made in adapting these best practices to other paradigms – service-oriented [11], functional [2, 15, 22], logic [24] and others – and in finding new paradigm-specific patterns. As several of these authors observed, studying design patterns in different programming

‡ Work partially supported by Fundação para a Ciência e Tecnologia under contracts PEst-OE/MAT/UI0209/2011 and PEst-OE/EEI/UI0434/2011.

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.

In this spirit, we carried the task of identifying several basic and other, more complex, patterns in the paradigm of Mdl-programs [5] – 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 that generalizes the original dl-programs [8]. The goal of this paper is to extend the original presentation in [6] with a more detailed analysis of the limitations that arise in this framework.

This work should be seen as quite distinct from that on ontology design patterns [14]. In the setting of Mdl-programs, ontologies are seen as immutable, being used and not changed, under the coordination of a set of rules. Our patterns focus therefore almost exclusively on these rules; thus, ontology design patterns and design patterns for Mdl-programs should in general be seen as two complementary techniques, and not as alternatives.

1.1 Motivation

The usefulness of combining description logics with rule-based reasoning systems led to the introduction of dl-programs [7, 8], which couple a description logic knowledge base with a generalized logic program, interacting by means of special atoms, the *dl-atoms*. These programs were later generalized to include several knowledge bases, yielding Mdl-programs [5].

Looking at Mdl-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 5).

Mdl-programs, combining description logic knowledge bases and a logic-based rule language, provide the adequate setting for the study of design patterns for the Semantic Web. Indeed, 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 [18]. The components of an Mdl-program are kept independent, giving them nice modularity properties; furthermore, Mdl-programs keep ontologies separate, which is much more convenient than e.g. merging them: not only is it simpler to have independent knowledge bases (which might even be physically separated, or independently managed), but merging ontologies is in itself a mighty task with its own specific problems [4, 16].

On the other hand, 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 (such as HEX-programs [9] or multi-context systems [3]), but expressive enough for their use within the Semantic Web.

The remainder of the paper is structured as follows. Section 2 explains Mdl-programs in detail. Section 3 presents seven different design patterns, and Section 4 illustrates their combined use by means of a larger example. Section 5 explores limitations and future directions of research, and Section 6 summarizes the contributions presented earlier.

2 Mdl-programs

Multi-description logic programs, the framework in which we will introduce our design patterns, generalize the original definition of dl-programs in [8] to accommodate for several description logic knowledge bases. This construction, introduced in [6] and detailed in [5], is in line with [25], although it sticks 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

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

often abbreviated to $DL_i[\chi; Q](\bar{t})$, 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) $Q(\bar{t})$ 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.

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

[2] The precise semantics can be found in [5]; the third operator in [8] is not included, as it can be defined in terms of $\cup$, and this option simplifies the semantics [25].

The semantics of Mdl-programs [5] is a straightforward generalization of the semantics of dl-programs [8] and will not be discussed here, since it will not be needed explicitly.

A common feature of multi-component systems is the need for entities in one component to "observe" entities in another component. In the setting of Mdl-programs, this is achieved by means of observers. An *Mdl-program with observers* is $\langle\{\mathcal{L}_1,\ldots,\mathcal{L}_n\},\mathcal{P},\{\Lambda_1,\ldots,\Lambda_n\},\{\Psi_1,\ldots,\Psi_n\}\rangle$ 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 $\langle\{\mathcal{L}_1,\ldots,\mathcal{L}_n\},\mathcal{P}\rangle$. 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.

Instead of defining formal semantics for Mdl-programs with observers, we introduced a translation of these into (standard) Mdl-programs that reduces observers to syntactic sugar. The above Mdl-program with observers thus 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 $\langle p,\neg S\rangle \in \Psi_i$.

We now illustrate these concepts by means of a simple example. Consider two knowledge bases $\mathcal{L}_1$, defining travel-related concepts, including that of (tourist) Destination, and $\mathcal{L}_2$, compiling information about wines, including a concept Region identifying some major wine regions throughout the world. We wish to join these ontologies by means of rules to obtain an Mdl-program with observers that reasons about wine-related destinations. This is achieved by taking $\mathcal{P}$ to be

$$\mathsf{wineDest}(\mathsf{Tasmania}) \leftarrow \qquad (r_1)$$
$$\mathsf{wineDest}(\mathsf{TamarValley}) \leftarrow \qquad (r_2)$$
$$\mathsf{wineDest}(\mathsf{Sydney}) \leftarrow \qquad (r_3)$$

$$\mathsf{overnight}(X) \leftarrow DL_1[;\mathsf{hasAccommodation}](X,Y) \qquad (r_4)$$
$$\mathsf{oneDayTrip}(X) \leftarrow DL_1[;\mathsf{Destination}](X),\ \mathit{not}\ \mathsf{overnight}(X) \qquad (r_5)$$

and observers $\Lambda_2 = \{\langle\mathsf{Region},\mathsf{wineDest}\rangle\}$, $\Psi_1 = \{\langle\mathsf{wineDest},\mathsf{Destination}\rangle\}$ and $\Lambda_1 = \Psi_2 = \emptyset$.

This very simple program defines a predicate wineDest with three instances obtained from rules (r_1–r_3), corresponding to three wine regions that are interesting tourist destinations, together with all instances of Region from $\mathcal{L}_2$, which

are obtained via the only element in Λ_2. Unfolding this observer yields the rule

$$\mathsf{wineDest}(X) \leftarrow DL_2[;\mathsf{Region}](X) \qquad (r_0)\,,$$

which corresponds to this intuitive semantics.

The set Ψ_1 causes Destination⊎wineDest to be added to the context of every dl-atom querying $\mathcal{L}_1$, extending $\mathcal{P}$'s view of $\mathcal{L}_1$ – namely in rules (r_4) and (r_5). This causes $\mathcal{L}_1$ to answer taking into account not only those instances of Destination in its knowledge base, but also those instances of wineDest that $\mathcal{P}$ knows about (including the ones derived from $\mathcal{L}_2$).

Rule (r_5) identifies the destinations that are only suitable for one-day trips. The possible destinations are obtained by querying $\mathcal{P}$'s extended view of $\mathcal{L}_1$ for all instances of Destination. The result is then filtered using the auxiliary predicate overnight defined in (r_4) as the set of destinations for which some accommodation is known. This uses the role hasAccommodation of $\mathcal{L}_1$, where hasAccommodation(t_1,t_2) holds whenever t_1 is a Destination and t_2 an accommodation facility located in t_1. The reason for resorting to (r_4) at all is the usual one in logic programming: the operational semantics of negation-as-failure requires all variables in a negated atom to appear in non-negated atoms in the body of the same rule. Note the impact of Ψ_1: if Destination were not being updated with the information from wineDest, the program would not be able to infer e.g. oneDayTrip(Tasmania).

Mdl-programs with observers have been implemented [17] as a plugin for the `dlvhex` tool [10].

3 Design patterns for Mdl-programs

In this section, we present a first set of seven design patterns for Mdl-programs, introduced in [6]. 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: each is 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 present a large-scale example that illustrates how the seven patterns work together, complementing each other.

Elementary design patterns. The three basic design patterns for Mdl-programs deal with three simple tasks: transporting information from the logic program to a knowledge base and reciprocally, and giving closed-world semantics to a concept or role in one of the knowledge bases.

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.

Pattern Observer Down.

Problem. Predicate p from $\mathcal{P}$ needs to be updated every time the extent (set of named individuals) of 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.

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

Pattern Observer Up.

Problem. In $\mathcal{P}$'s view, concept or role S from $\mathcal{L}_i$ needs to be updated every time the extent of predicate p (of the same arity as S) in $\mathcal{P}$ is changed.
Solution. Add the pair $\langle p, S\rangle$ to Ψ_i.

The third building block addresses a very typical situation in ontology usage: a concept or role should be given closed-world semantics.

Pattern Closed-world.

Problem. In $\mathcal{P}$'s view, 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 $s^-(X) \leftarrow \mathit{not}\ s^+(X)$ to $\mathcal{P}$.

Derived design patterns. 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.

A useful 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. This design pattern appears often in combination with Definitions with Holes below.

Pattern Transversal Observer.

Problem. In $\mathcal{P}$'s view, concept (or role) S from $\mathcal{L}_i$ needs to be updated every time the extent of concept (resp. role) R from $\mathcal{L}_j$ is changed ($i \neq j$).
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.

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

Pattern Split Definitions.

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 Split Definitions consists of a combined application of several Observer Down, all with the same observer predicate in $\mathcal{P}$. This pattern deals 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. For this reason, this pattern was originally named Polymorphic Entities [6].

The converse situation yields a different pattern, due to the way Mdl-programs are typically developed: the logic program is written to connect pre-existing knowledge bases. This pattern is particularly useful when in presence of terminological ontologies where some concepts are not defined, and captures a typical way of working with ontologies.

Pattern Definitions with Holes.

Problem. Concept or role S is needed for reasoning in $\mathcal{L}_i$, 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 Split Definitions.

This pattern corresponds to the Template Method pattern of object-oriented programs [13], and to the Programming with Holes technique of [21]. In the next section we will show an example where the holes are filled in by resorting to Split Definitions.

The last design pattern in this section applies when several components of an Mdl-program contribute to the definition of a predicate.

Pattern Combined Definitions.

Problem. There is a predicate being defined in 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 Combined Definitions is essentially different from Observer: in Observer, a predicate is defined in *one* component and used in others; in Combined Definitions, 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 this pattern 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.

This pattern was originally introduced in [6] under the name Lifting.

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.

4 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 WishYouWereThere 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, WishYouWereThere'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 WishYouWereThere'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

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

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 Split Definitions pattern to connect toPay to payFlight and payHotel. The resulting Mdl-program with observers is such that:

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

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 Combined Definitions design pattern applies. Assuming that $\mathcal{L}_F$ uses $\mathsf{TopClient}$ for its special clients, $\mathcal{L}_H$ uses Gold and $\mathcal{P}$ defines $\mathsf{special}$, these three predicates are given the same semantics through Combined Definitions. 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

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

and four similar observers in Λ_H and Ψ_H, with Gold in place of $\mathsf{TopClient}$.

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 $\mathsf{special}$ is a predicate from $\mathcal{P}$, where the semantics is closed-world: the application of Combined Definitions ensures that Gold and $\mathsf{TopClient}$, 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, one could have

$$\exists \mathsf{flies.10000OrMore} \sqsubseteq \mathsf{TopClient} \qquad \in \mathcal{L}_F$$
$$\mathsf{familyBooking} \sqsubseteq \mathsf{Gold} \qquad \in \mathcal{L}_H$$
$$\mathsf{special}(X) \leftarrow \mathsf{booked}(X, Y), \mathsf{expensive}(Y) \qquad \in \mathcal{P}$$

A subtle issue now appears regarding the consistency problems that may arise from the use of the Combined Definitions 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 Combined Definitions 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

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

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

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

Suppose that airline employee Ann qualifies for WishYouWereThere promotions because she spent three weeks in Jamaica with her husband and their five children, hence $\mathsf{Gold}(\mathsf{Ann})$ 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 $\mathsf{TopClient}(\mathsf{Ann})$ holds that knowledge base will not return $\mathsf{100BonusMilesWinner}(\mathsf{Ann})$. However, she will earn the $20\%\mathsf{Discount}$, 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:

$$\begin{aligned}
\Lambda_F\colon\ & \langle \mathsf{Flyer}, \mathsf{client}\rangle \\
& \langle \mathsf{payFlight}, \mathsf{toPay}\rangle \\
& \langle \mathsf{TopClient}, \mathsf{special}\rangle \\
& \langle \neg\mathsf{TopClient}, \mathsf{notSpecial}\rangle \\
& \langle \mathsf{Blocked}, \mathsf{blockedF}\rangle
\end{aligned}
\qquad
\begin{aligned}
\Lambda_H\colon\ & \langle \mathsf{Guest}, \mathsf{client}\rangle \\
& \langle \mathsf{payHotel}, \mathsf{toPay}\rangle \\
& \langle \mathsf{Gold}, \mathsf{special}\rangle \\
& \langle \neg\mathsf{Gold}, \mathsf{notSpecial}\rangle \\
& \langle \mathsf{Blocked}, \mathsf{blockedH}\rangle
\end{aligned}$$

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.

5 Beyond these patterns

The set of design patterns we presented does not by any means claim to be exhaustive or all-powerful. Design patterns for several programming paradigms have been around for more than two decades, and dozens of different patterns have been proposed and applied, often in very specific contexts. Our goal was to show how some of the most common general-purpose design patterns can be implemented within the framework of Mdl-programs, thereby illustrating the potential of this formalism. Among the more elaborate design patterns, our selection took into account the ones that can be more naturally formalized using Mdl-programs with observers. In this section we explore some limitations and discuss future directions for our work.

In a practical context, it is not uncommon to have a function in $\mathcal{P}$ whose definition is unstable in the sense that it may vary, for example rotating cyclically among several possibilities. Also, there are cases where one foresees possible future variations which are not contemplated in the existing requirements. The following pattern provides a clean way to implement such functions in a way that minimizes undesirable impact on the other elements of the program.

Pattern Indirection.

Problem. There is a predicate p (in $\mathcal{P}$) whose definition may vary, but $\mathcal{P}$ should be protected from these variations, in the sense that it should suffer minimal and easily identifiable modifications.

Solution. Create a stable interface through a dedicated knowledge base $\mathcal{L}_I$ in $\mathcal{KB}$.

Define p with a set of rules, each one protected by a query to $\mathcal{L}_I$ on a concept or role S.

Define S in $\mathcal{L}_I$ such that the satisfiable clauses of p are the ones corresponding to its current definition.

Different applications of this pattern may share the same dedicated knowledge base $\mathcal{L}_I$, since each of them only looks at a particular concept or role.

The INDIRECTION design pattern captures some aspects of the principle of Protected Variations in object-oriented programming [19], which is a root principle motivating most of the mechanisms and patterns in programming and design that provide flexibility and protection from variations. This pattern was not originally introduced in [6].

As a particular case, it may happen that a component of a system is not 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 may also arise 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.

Both of these situations reflect another aspect of the same principle of Protected Variations mentioned above, but now the point of variation in $\mathcal{KB}$ that $\mathcal{P}$ is being protected from lies in one of the knowledge bases and not in $\mathcal{P}$ itself.

Pattern ADAPTER.

Problem. One wants to work with $\mathcal{L}_k$ independently of its particular syntax.
Solution. Decide the names to use in $\mathcal{P}$ for the concepts and roles involved, and add an empty interface knowledge base $\mathcal{L}_I$ to $\mathcal{KB}$ using these names. Later, 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 of the usual Adapter design pattern: the Mdl-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. The following example illustrates this situation.

Consider an interface $\mathcal{L}_I$ specifying 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 $DL_C[A \uplus \mathsf{thisIsTrue}; B](X)$ returns 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)

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

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

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

– introduced by the concretization of the observer sets – returns 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

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

since the query is directly on the concept whose extent was altered. This is a restriction with respect to the full power of Mdl-programs; but we see it as a *feature* of the Adapter design pattern. Should a context arise where such flexibility is essential, then this is not the right design pattern to apply. In practice, situations where a Adapter is applicable are common enough to make it a useful pattern.

A more problematic situation arises when one wants to control or restrict access to a resource, for example a database containing sensitive information – a problem typically addressed by means of a proxy. 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.

An implementation along the lines we have followed so far would explore the possibility of a proxy knowledge base to serve as a mediator between two components. In the setting of Mdl-programs, however, 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, since the person who develops the logic program can access the implementation of the proxy.

There would be ways to go around these problems, namely by extending Mdl-programs with appropriate syntactic constructions besides observers. Indeed, our motivation for defining Mdl-programs with observers was, primarily, to guarantee 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 it turned out, this construction is powerful enough to allow for elegant implementations of all the design patterns discussed earlier. There is an aspect that cannot be overstressed: the sets Λ_i and Ψ_i are syntactic sugar. As such, they do not add to the expressive power of Mdl-programs, but they substantially increase their legibility and internal structure. By working with an Mdl-program with observers, one can more easily understand the core of the program (which is the logic program $\mathcal{P}$) without being disturbed by the presence of myriads of rules that connect $\mathcal{P}$ with the several knowledge bases. In particular, most of the design patterns we presented can be expressed simply as adding specific pairs to carefully chosen observer sets Λ_i and Ψ_i – yielding a clean program that is also very easy to maintain and extend. At the end of the day, though, the Mdl-program with observers simply translates into an Mdl-program.

To deal with a full Adapter or Proxy, one would have to extend the syntax of Mdl-programs in a non-conservative way; but doing this would remove their

simplicity, which was the main motivation for using them in the first place. Therefore, this section can be summarized as follows: the design patterns here introduced allow one to write Mdl-programs in a clean and elegant way, thereby obtaining programs of a better quality than *ad hoc* designed solutions. If one needs to go beyond the power of Mdl-programs, namely to implement a full-fledged proxy, then one should not use them at all, but rather move to a more powerful formalism.

6 Conclusions

The purpose of the present study, at this stage, is to show that design patterns 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 extends the original presentation in [6] by discussing an initial set of design patterns for Mdl-programs, together with a large-scale example illustrating their application, and showing the inherent limitations of this programming framework.

An aspect that will have to be addressed in the future 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. Also, the use of observers (essential to many of the patterns proposed) introduces higher complexity, especially when non-stratified negation is involved. It is important to understand the compromise between efficiency and quality obtained by a systematic use of design patterns by means of a practical evaluation using the prototype implementation of Mdl-programs within `dlvhex` [17].

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 [5].

References

1. M. Adams, J. Coplien, R. Gamoke, R. Hanmer, F. Keeve, and K. Nicodemus. Fault-tolerant telecommunication system patterns. In *Pattern Languages of Program Design 2*, pages 549–562. Addison–Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.
2. S. Antoy and M. Hanus. Functional logic design patterns. In Z. Hu and M. Rodríguez-Artalejo, editors, *Proceedings of FLOPS 2002*, volume 2441 of *LNCS*, pages 67–87. Springer, 2002.
3. G. Brewka and T. Eiter. Equilibria in heterogeneous nonmonotonic multi-context systems. In *Proceedings of AAAI-07*, pages 385–390. AAAI Press, 2007.
4. J. de Bruijn, M. Ehrig, C. Feier, F. Martíns-Recuerda, F. Scharffe, and M. Weiten. Ontology mediation, merging, and aligning. In J. Davies, R. Studer, and P. Warren, editors, *Semantic Web Technologies: Trends and Research in Ontology-based Systems*. John Wiley & Sons, Ltd, Chichester, UK, 2006.

5. L. Cruz-Filipe, R. Henriques, and I. Nunes. Description logics, rules and multi-context systems. accepted for publication in *Proceedings of LPAR 2013*, 2013.
6. L. Cruz-Filipe, I. Nunes, and G. Gaspar. Patterns for interfacing between logic programs and multiple ontologies. In Joaquim Filipe and Jan Dietz, editors, *KEOD2013*, pages 58–69. INSTICC, 2013.
7. T. Eiter, G. Ianni, T. Lukasiewicz, and R. Schindlauer. Well-founded semantics for description logic programs in the semantic Web. *ACM Transactions on Computational Logic*, 12(2), 2011. Article 11.
8. T. Eiter, G. Ianni, T. Lukasiewicz, R. Schindlauer, and H. Tompits. Combining answer set programming with description logics for the semantic web. *Artificial Intelligence*, 172(12–13):1495–1539, 2008.
9. T. Eiter, G. Ianni, R. Schindlauer, and H. Tompits. Effective integration of declarative rules with external evaluations for semantic-web reasoning. In Y. Sure and J. Domingue, editors, *Proceedings of ESWC 2006*, volume 4011 of *LNCS*, pages 273–287. Springer, 2006.
10. T. Eiter, G. Ianni, R. Schindlauer, and H. Tompits. Towards efficient evaluation of HEX programs. In J. Dix and A. Hunter, editors, *Proceedings of NMR-2006, ASP Track*, pages 40–46. Institut für Informatik, TU Clausthal, Germany, 2006.
11. T. Erl. *SOA Design Patterns*. Prentice Hall, New York, 2009.
12. M. Fowler. *Patterns of Enterprise Application Architecture*. Addison–Wesley, 2002.
13. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison–Wesley, 1995.
14. A. Gangemi and V. Presutti. Ontology design patterns. In S. Staab and R. Studer, editors, *Handbook on Ontologies*, International Handbooks on Information Systems, pages 221–243. Springer, 2009. 2nd edition.
15. J. Gibbons. Design patterns as higher-order datatype-generic programs. In R. Hinze, editor, *Proceedings of WGP 2006*, pages 1–12. ACM, 2006.
16. B. Grau, B. Parsia, and E. Sirin. Combining OWL ontologies using e-connections. *Journal of Web Semantics*, 4(1):40–59, 2005.
17. R. Henriques. Integration of ontologies with programs based on rules. Master's thesis, FCUL, November 2013.
18. M. Kifer and H. Boley (eds.). RIF overview, June 2010. W3C Working Group Note, `http://www.w3.org/TR/2010/NOTE-rif-overview-20100622/`.
19. C. Larman. *Applying UML and Patterns*. Prentice–Hall, 2004. 3rd Edition.
20. T.G. Mattson, B.A. Sanders, and B.L. Massingill. *Patterns for Parallel Programming*. Addison–Wesley, 2005.
21. B. Meyer. *Object-Oriented Software Construction*. Prentice–Hall, 1997. 2nd Ed.
22. P. Norvig. Design patterns in dynamic programming, 1996. Tutorial slides presented at Object World, Boston, MA, May 1996, available at `http://norvig.com/design-patterns/`.
23. D.C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*. John Wiley & Sons, 2000.
24. L. Sterling. Patterns for Prolog programming. In A.C. Kakas and F. Sadri, editors, *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part I*, volume 2407 of *LNCS*, pages 374–401. Springer, 2002.
25. K. Wang, G. Antoniou, R.W. Topor, and A. Sattar. Merging and aligning ontologies in dl-programs. In A. Adi, S. Stoutenburg, and S. Tabet, editors, *Proceedings of RuleML 2005*, volume 3791 of *LNCS*, pages 160–171. Springer, 2005.