Modeling Design Patterns with Description Logics: A Case Study

Yudistira Asnar, Elda Paja, and John Mylopoulos

Department of Information Engineering and Computer Science
University of Trento, Italy
{yudis.asnar,paja,jm}@disi.unitn.it

Abstract. Design Patterns constitute an effective way to model design knowledge for future reuse. There has been much research on topics such as object-oriented patterns, architectural styles, requirements patterns, security patterns, and more. Typically, such patterns are specified informally in natural language, and it is up to designers to determine if a pattern is applicable to a problem-at-hand, and what solution that pattern offers. Of course, this activity does not scale well, either with respect to a growing pattern library or a growing problem. In this work, we propose to formalize such patterns in a formal modeling language, thereby automating pattern matching for a given problem. The patterns and the problem are formalized in a description logic. Our proposed framework is evaluated with a case study involving Security & Dependability patterns specified in Tropos SI*. The paper presents the formalization of all concepts in SI* and the modeling of problems using OWL-$\mathcal{DL}$ and SWRL. We then encode patterns as SPARQL and SQWRL queries. To evaluate the scalability of our approach, we present experimental results using models inspired by an industrial case study.

Keywords: design patterns, description logics, pattern matching.

1 Introduction

Design patterns represent recurring design problems and how to solve them. Design patterns gained prominence initially in Architecture [1], and within Computer Science with the widely-known and used design patterns for object-oriented design [2]. Today, there are dozens of proposals for design patterns covering a range of design domains, such as: requirements, software architectures, business processes, workflows etc.

Generally speaking, a design pattern consists of a triple *(context, problem-to-solve, solution-pattern)* [1]. To use a pattern one needs to first match the design problem-at-hand to the context, (if successful), match the problem-at-hand to the pattern context (thereby creating mapping for pattern variables), and revise the problem at-hand by using the solution-pattern. However, design patterns often are represented and documented informally in natural language (for an example [2]). This means that it is up to users of a pattern library to determine which patterns are relevant, and also exactly how they apply to the design problem-at-hand. Unfortunately, this approach does not scale with respect to the size of the pattern library, the problem-at-hand, or the expertise of the designer. Indeed, there is plenty of evidence that pattern libraries have low acceptability rates, especially so among non-expert users by now [3]. Some of the prominent

H. Mouratidis and C. Rolland (Eds.): CAiSE 2011, LNCS 6741, pp. 169–183, 2011.

reasons are (i) finding patterns relevant to the problem-at-hand is hard [3], often because pattern libraries are unstructured; (ii) understanding patterns requires some expertise [4] that users often do not have.

The main objective of this paper is to address this situation by formalizing design patterns in a language that has a built-in matching operation, thereby offering automated support for the identification of applicable patterns for a given design problem, also for formulating a design solution. In this work, we formalize the pattern library as well as the design problem as a knowledge base using description logics [5], and represent the context of the patterns as a query to the knowledge base (SPARQL and SQWRL). To evaluate the proposed framework, we use a case study coming from SERENITY Security and Dependability (S&D) patterns [6]. Our evaluation is conducted along two fronts. Firstly we want to see to what extent a description logic can accommodate patterns expressed in an ontologically rich pattern language with built-in concepts such as goal, agent, strategic dependencies among agents, and more. After all, description logics are ontologically simple in the sense that they usually come only with two primitives: concepts and roles (binary relationships). Secondly, we want to know if our proposed solution scales as the size of the design problem to be matched against patterns in the pattern library grow.

The rest of the paper is structured as follows. We outline the research baseline (description logic and OWL-$\mathcal{DL}$) in Section 2. Section 3 presents the case study, while Section 4 details the formalization of patterns. In Section 5, we explain the experimental setup we used and its results. We then offer a discussion of related work in Section 6 and concluding remarks in Section 7.

2 Baseline

Description Logics (DL). [5] are used to formalize design patterns and also the problem-at-hand in terms of concepts, roles, individuals and their relations. In DL there is no explicit use of variables. Instead, operators are exploited to define complex concepts and roles starting from atomic ones. The set of operators is restricted to ensure tractability for reasoning, such as deciding if a concept is a generalization (subsumes) another concept. A DL knowledge base is composed of two components: 1) a terminological box (TBox) consisting of definitions for concepts and roles, and 2) an assertional box (ABox) consisting of individuals and true facts about them. For instance, if we consider a knowledge base about persons, the TBox would contain concepts such as Person, Male, Female, Parent, Child etc., and roles such as hasChild, and axioms of the form:

$$\begin{aligned} Person &\sqsubseteq Male \sqcup Female \\ Man &\equiv Person \sqcap Male \\ Father &\sqsubseteq Man \\ Father &\equiv Man \sqcap \exists hasChild.Person \end{aligned}$$

The ABox, on the other hand, contains assertions regarding individuals, such as axioms of the form: $Father(Tom)$.

OWL-$\mathcal{DL}$ Among DLs, we chose OWL-$\mathcal{DL}$ because it is state-of-the-art as far as DL go, it is part of W3C standard, and is readily available with several possible implementations to choose from. Our use of description logics is as follows: we formalize the modeling language concepts using OWL-$\mathcal{DL}$(creating the TBox), and we use this as a basis for representing formally the context of the patterns. We perform matching at the instance level (ABox), that is why we represent patterns as queries. However, the expressiveness of OWL-$\mathcal{DL}$ alone is weak to represent some constraints, such as the ones related to individuals. We solved the problem by adopting the SWRL rule language [7]. This allows us to enrich the formal pattern description with inferred knowledge, thereby ensuring better pattern matching for the problem-at-hand.

3 Case Study

Several works have been proposed in the literature on S&D patterns (e.g., fault-tolerant patterns [8], security patterns [9], SERENITY patterns [6]). In this work, we use SERENITY patterns, developed within the EU SERENITY project, it is state-of-the-art for its intended application domain and we had expertise on both the patterns and their uses.

SERENITY Patterns. [6] are represented using Alexander's pattern language as triples: $\langle \text{Context}, \text{S\&D Requirements}, \text{S\&D Solution} \rangle$ The *Context* defines the state-of-affairs the problem/situation where the pattern could be applied, which is depicted in terms of the minimum set of actors and relationships, where the S&D Requirements are not fulfilled. *S&D Requirements* specify the required S&D Properties that must be satisfied in the model (representing the problem). *S&D Solution* describes the modifications that need to be performed to the context in order to meet S&D Requirements. The description of SERENITY patterns is enriched with additional description about when, how, and for what the patterns are intended for.

In [6], patterns are identified in scenarios extracted from business cases (e.g., Air Traffic Management, e-Business, Online-Tax, Smart-home) and then described in natural language. Patterns, then, are represented formally; *Context* and *S&D Solution* are represented in terms of SI* models, whereas the *S&D Requirements* in ASP (answer set programming - an extension of DATALOG). The pattern library is composed of 29 SERENITY patterns (4 legal, 3 privacy, 11 security and 11 dependability patterns) [6]. Table 1 presents some of the SERENITY patterns described in natural language. For an illustrative example, we use pattern DP2.1 on *Collaboration in Small Groups for Risky Activities*. In addition, the SERENITY pattern library was used for evaluating the performance of our implementation (Section 5).

Tropos SI*. [10,11] is a modeling language for security requirements. The language offers primitive concepts such as actor, goal, task, as well as various kinds of relationships among actors. This modeling language is used for representing *Context* and *S&D Solution* of SERENITY patterns.

Fig. 1 depicts one of the SERENITY scenarios (e.g., Air Traffic Management - ATM). SI* considers *intentional Actor* as basic concept (e.g., Executive Controller,

Table 1. SERENITY S&D Patterns in Natural Language

Pattern Name	Natural Language Description
SP1. Proof of Fulfillment for Ensuring Non-Repudiation	To prevent repudiation, the executor needs to provide evidence of performing the action to the benefitor, in addition to performing the action.
SP4. Artefact Generation as an Audit Trail	To prevent repudiation of some actions upon a shared resource, a group of agents needs to keep a common audit trail.
DP2.1. Collaboration in Small Groups for Risky Activities	To cope with an activity where a tight coordination among agents is crucial, a failure on a risky sub-activity may compromise the team goal. However, one team member might have a capability to mitigate the risk of the risky sub-activity therefore that team member must mitigate the risk for the team success.
DP6. Reinforcing Overlapping Responsibilities for Robustness	A critical task must be completed successfully most of the time. Therefore, several team members are responsible to perform the task.
PP1. Sign an Agreement to Address Lack of Trust on the Use of Private Data	Sometime a customer does not trust an organization accessing its data. Therefore a representative agent of the organization needs to ask for customer consent before accessing customer's data.

Supervisor, Alice) that wants to achieve goals (ensure traffic safety in its sector, form team sectors). Actors are equipped with certain abilities (e.g., resolve traffic conflict), have beliefs, etc. They are further specialized into *roles* (e.g., Supervisor, Executive Controller, and Team sector) as abstract actors in an organization that are played by *agents* (e.g., Bob, Alice, Dan), which are concrete actors.

Actors (e.g., executive controllers) intend to achieve/satisfy their business goals (manage traffic in the sector) by relying on their capabilities and those of other actors (e.g., resolve traffic conflict). The term Business Object refers to a *goal*, a *task*, or a *resource*. *Goal* represents a state-of-affair that an actor intends to achieve (manage traffic in sector). *Task* is a course of actions performed by an actor to achieve a desired goal (give airway commands). *Resource* refers to physical or informational entities required to achieve goals (flight progress strips) or to perform tasks (air situation display). However, the fulfillment of these business objects is affected by uncertain *events*. Events that can cause a goal failure are risks (overload traffic), while events that can help in the fulfillment of a goal are treated as opportunities (deployment new system for air conflict prediction).

In addition to capturing the strategic rationale of an actor, SI* captures strategic inter-dependencies among actors in an organization. Inter-dependencies can be either *delegation* and *trust* relationships among actors. Delegations also come in two flavours: 1) execution of business objects and 2) permission/entitlement on business objects. *Delegation* refers to the transfer of responsibilities (*Delegation on Execution*) or rights (*Delegation on Permission*). In Fig. 1, team sector delegates the execution of manage inbound traffic to another actor - planning controller. **Trust** refers to the belief and expectation of an actor that another actor (trustee) will fulfill its commitments (will execute all assigned business objects) and will respect its permissions. For an example, Alice trusts Bob to fulfill managing traffic in Sector SU1.

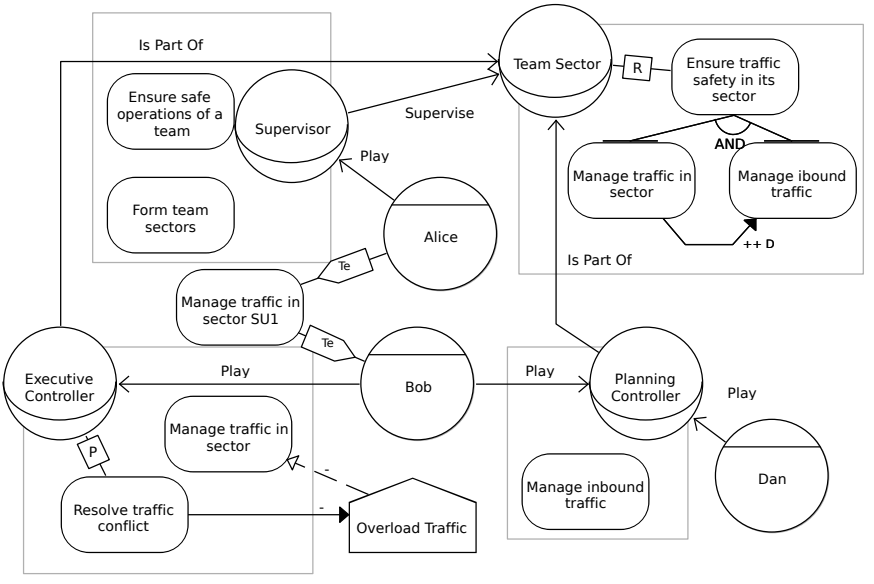


Fig. 1. A SI* Diagram from a fragment of the Air Traffic Management scenario

A SI* model captures relationships between concepts using several basic relations: 1) *AND/OR-decomposition* to refine a goal, 2) *contribution* to capture the effects of a goal to another, 3) *impact* to model the impact of an uncertain event to a business object. Fig. 1 depicts the goal ensure traffic safety in its sector is AND-decomposed into manage traffic in sector and manage inbound traffic, where the achievement of both subgoals are necessary to achieve the up-level goal. Moreover, the achievement of the latter goal contributes positively to the success of the former one. In ATM, we consider the effect of overload traffic event to the goal manage traffic in sector, and it can be mitigated with the capability of an actor in resolve traffic conflict.

4 Formalizing Patterns

Our formalization process includes four steps: (i) Formalize the SI* language by defining non-overlapping OWL-$\mathcal{DL}$ concepts for all SI* primitives and one or more roles for every primitive SI* relationship; (ii) Formalize the context of each pattern using the concepts and roles introduced in step (i); (iii) Enrich the formal pattern descriptions with implicit knowledge [1]; (iv) Represent the problem-at-hand in the ABox by instantiating the concepts and roles of step (i).

4.1 Formalizing SI* Primitives

In general, we represent nodes (e.g., goal, task, resource, event) in a SI* model as concepts and binary relations (e.g., actor's associations, contributions, decompositions, im-

[1] Availability of a domain expert is essential here because this implicit knowledge (constraints, alternatives, and more) is often missing from the informal pattern description.

pacts) as roles. Moreover, inter-actor relations (e.g., delegation, trust, monitoring) are encoded as concepts as well, since they are ternary relations. Later, we discuss some considerations that underlie our formalization.

In the following, we discuss some issues that have arisen while formalizing concepts/relations of the language in terms of DL concepts/roles.

Role versus Subsumption. The relationship between a goal and its subgoals could have been represented as a subsumption relationship or a role, say $hasSubgoal$:

$$Subgoal \sqsubseteq Goal \quad vs. \quad hasSubgoal.Goal$$

But instances of a subgoal do not need to also be instances of the parent goal (for example, consider goal "schedule meeting" and subgoal "collect timetables"). Accordingly, we chose to go with the second option.

Ternary/n-ary relations. Since OWL-$\mathcal{DL}$ does not support N-ary relations, $N \geq 3$, we decided to represent such relationships in terms of a concept and several roles. For example,

$$DelegationOnExecution \equiv DelegationOnExecution \sqcap (hasDelegator = 1) \sqcap (hasDelegatum = 1) \sqcap (hasDelegatee = 1)$$

Consider Fig. 1, where Team Sector delegates execution of manage traffic in the sector to Executive Controller. In the ABox, a delegation on execution in such a setting is represented as follows:

$$DelegationOnExecution(\text{Del-exec1})$$
$$hasDelegator(\text{Del-exec1}, \text{Team Sector})$$
$$hasDelegater(\text{Del-exec1}, \text{Executive Controller})$$
$$hasDelegatum(\text{Del-exec1}, \text{Manage traffic in sector})$$

4.2 Understanding and Formalizing a Pattern as a Query

Once designers specify the pattern language in the DL TBox, the next step is understanding the essence of the pattern description and formalizes the context in terms of OWL queries (i.e., using terms specified in the DL TBox). Designers need to be aware that some patterns are very generic and sometimes vague, (e.g., patterns described in natural language in [8]), and others are rather restrictive because of the limitation of the pattern language (e.g., patterns described using a modeling language, such as [6]). In this phase, we leave it in the hands of the designers to decide how much detail they want to put into the patterns or how generic the pattern should be.

In [6], the context of the patterns is modeled in terms of an "abstract" SI* model that includes variables (denoted by identifiers with capitalized letters). For instance in

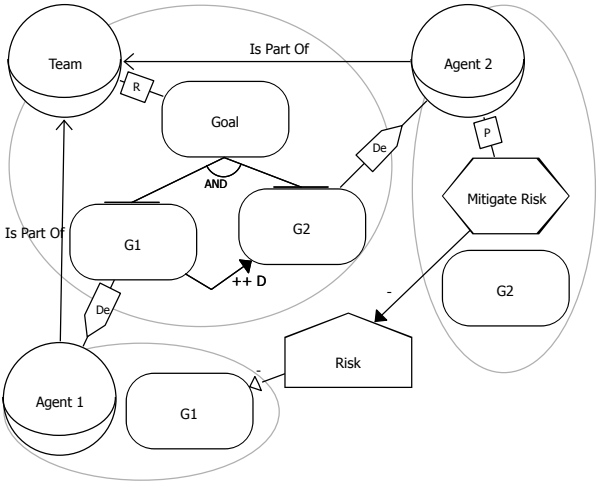


Fig. 2. The "Collaboration in Small Groups for Risky Activities" pattern

Fig. 2, the pattern indicates TEAM delegates execution of G1 to AGENT1 will match elements that involve two roles, where the first role delegates execution of a goal to the second role. Moreover, in some patterns (e.g., GoF [2]) pattern contexts are left implicit and designers need to fill the details. A pattern is applicable to the problem-at-hand if all constructs in the pattern match corresponding elements of the problem. When such a match is found, the reasoner returns not only true, but also mappings for pattern variables. Patterns are formalized in two parts: (i) the *mandatory part* must be matched in the problem-at-hand for the pattern match to succeed; (ii) the *optional part* can bring about useful mappings for variables, but do not affect the outcome of a pattern match. In our approach, matching is performed at the individual level (ABox) of the knowledge base. Therefore, the reasoner checks for individuals present in the problem and the relationships among them, and lists all individuals that match the pattern context. Note that reasoners can return more than one resultset, since it is likely there are several parts of the model that match a given pattern context.

In the context of security and dependability, the *optional part* is only used to define a new actor (including its capabilities & responsibilities) that is not necessarily present in the problem. For instance, A client needs to buy a house from Company A, but he does not trust Company A. Based on [6], to ensure security, the designers can "patch" the trust issue by having a contract arranged by a lawyer. However, in most cases the lawyer does not exist in the "current" statement of the problem, hence the need for optional elements during a pattern match. Application of the pattern basically introduces, a new role - lawyer (i.e., a trusted $3^{rd}party$).

OWL-$\mathcal{DL}$ models can be queried using two languages: SPARQL [12], and SQWRL [13]. The SPARQL is a W3C Recommendation [12] for querying RDF. RDF essentially offers directed labeled graph data format, built out of triples. Thus, SPARQL queries are expressed in terms of triple patterns, consisting of a subject, predicate, and object. The

```
prefix tropos: <http://www.owl-ontologies.com/Tropos_Ontology.owl#>
SELECT   ?team ?goal ?agent1 ?agent2 ?subgoal1 ?subgoal2
WHERE {
        ?team tropos:request ?goal.
        ?goal tropos:isAndDecompositionOf ?g1.
        ?goal tropos:isAndDecompositionOf ?g2.
   FILTER (?subgoal1 != ?subgoal2).
        ?agent1 tropos:isPartOf ?team.
        ?agent2 tropos:isPartOf ?team.
   FILTER (?agent1 != ?agent2).
        ?team tropos:hasDelegation ?d1.
        ?d1 tropos:hasDelegatee ?agent1.
        ?d1 tropos:hasDelegatum ?g1.
        ?team tropos:hasDelegation ?d2.
        ?d2 tropos:hasDelegatee ?agent2.
        ?d2 tropos:hasDelegatum ?g2.
        ?agent2 tropos:provides ?mitigateRisk.
        ?g1 tropos:hasNegDContribution ?g2.
        ?task tropos:hasNegContribution ?risk.
        ?risk tropos:hasNegImpact ?g1.
   OPTIONAL{?agent2 tropos:requests ?mitigateRisk.}
}
```

Fig. 3. SPARQL representation of DP 2.1

Turtle data format [2] is used to represent triple patterns. The query attempts to match the triples on the graph pattern against the model [14]. SPARQL just queries the model and does not support inference [12], nor does it modify the RDF dataset. However, some frameworks (e.g., JENA) and rule engines [15], have the capacity to perform inference and update the dataset by performing OWL reasoning.

Alternatively, a more expressive query language that is founded on DL semantics and supports comprehensive querying of OWL is SQWRL [13]. SQWRL is a SWRL-based query language [7]. SQWRL provides SQL-like operations to retrieve knowledge from an OWL ontology. Similarly to SPARQL, in SQWRL we try to capture all concepts and relationships present in a pattern. Since SQWRL understands the semantics of OWL and SWRL rules, it understands not only the explicit, but also the inferred knowledge. For example, the DP2.1 of SERENITY pattern (Fig. 2 described in Table 1), can be translated into a SPARQL Query (Fig. 3). Each node of the pattern context is a variable in the query and each edge is an RDF triplet. For a SQWRL Query, the DP2.1 translational is shown in Fig. 4.

4.3 Enriching DL T-Box with Implicit Knowledge

Often details of the patterns are described in natural language, due to the expressivity limitation of the pattern language. This was certainly the case with our case study.

Back to our example in DP2.1, in SI* the notion of "request" means that an actor intends to achieve a particular goal. However, based on DP2.1's description the intent

[2] Turtle: `http://www.w3.org/TeamSubmission/turtle/`

$1: \text{requests}(?team, ?goal) \wedge \text{isPartOf}(?agent1, ?team) \wedge \text{isPartOf}(?agent2, ?team) \wedge$
$2: \text{hasSubgoal}(?goal, ?g1) \wedge \text{hasSubgoal}(?goal, ?g2) \wedge$
$3: \text{hasDelegation}(?team, ?d1) \wedge \text{hasDelegatee}(?d1, ?agent1) \wedge \text{hasDelegatum}(?d1, ?g1) \wedge$
$4: \text{hasDelegation}(?team, ?d2) \wedge \text{hasDelegatee}(?d2, ?agent2) \wedge \text{hasDelegatum}(?d2, ?g2) \wedge$
$5: \text{hasNegDContribution}(?goal1, ?goal2) \wedge \text{provides}(?agent2, ?mitigateRisk) \wedge$
$6: \text{hasNegImpact}(?risk, ?g1) \wedge \text{hasNegContribution}(?mitigateRisk, ?risk) \rightarrow$
$7: sqwrl: select(?team, ?goal, ?agent1, ?g1, ?agent2, ?g2, ?risk, ?mitigateRisk) \wedge$
$8: sqwrl: columnNames(\text{"}team\text{"}, \text{"}goal\text{"}, \text{"}agent1\text{"}, \text{"}g1\text{"}, \text{"}agent2\text{"}, \text{"}g2\text{"}, \text{"}risk\text{"}, \text{"}mitigateRisk\text{"})$

Fig. 4. SQWRL representation of DP 2.1

$$
\begin{aligned}
\text{aim}(?a, ?goal) &\leftarrow \text{requests}(?a, ?goal)\\
\text{aim}(?a2, ?goal) &\leftarrow \text{requests}(?a1, ?goal) \wedge \text{hasDelegation}(?a1, ?d) \wedge\\
&\quad\ \text{hasDelegatee}(?d, ?a2) \wedge \text{hasDelegatum}(?d, ?goal)\\
\text{aim}(?a2, ?goal) &\leftarrow \text{aim}(?a1, ?goal) \wedge \text{hasDelegation}(?a1, ?d) \wedge \text{hasDelegatee}(?d, ?a2) \wedge\\
&\quad\ \text{hasDelegatum}(?d, ?goal)
\end{aligned}
$$

Fig. 5. Relaxing "request" on SI* in SQWRL

of "request" is more relaxed – direct request (i.e., the actor "requests" fulfillment of a goal) or indirect request (i.e., another actor delegates the execution of a goal to him/her, to the actor). Accordingly, we decided to extent the DL TBox and revise the pattern formalization using those new concepts/roles. However, this extension can only be done when we use SQWRL and **not** SPARQL. In Fig. 5, we illustrate an example of the extension of the "request" relation in SI*, namely "aim". To be closer with the DP2.1's description one needs to replace line 3-4 of Fig. 4 with the following:

$$\text{aim}(?agent1, ?g1) \wedge \text{aim}(?agent2, ?g2)$$

4.4 Representing the Problem in the ABox

Finally, designers need to represent the problem-at-hand in terms of instances of concepts and roles in the ABox.

Concept versus Individual. Individuals have a unique identity, and their description can be modified by adding more assertions in the ABox. Conversely, the definition of concepts cannot be modified [16].

The first alternative (i.e., subclasses of the DL TBox) allows us to reason whether a pattern appears anywhere in the problem, but it cannot provide the mapping between construct in the pattern's solution and the problem-at-hand. The second alternative (i.e., individuals in the DL ABox) on the other hand can provide such mappings, but it will not allow us to reason in a situation where the problem contains both abstract and concrete entities in the real world, because both entities will be encoded as individuals and the reasoner will treat them equally. Since we deal with the problem at the design level where mostly models capture the class level instead of the object one, we have chosen the second alternative (i.e., as series of individuals) as the most suitable to our needs.

Moreover, providing mapping between the pattern and the problem is a critical feature to support designers in applying the patterns to resolve their problem.

Here is a fragment of the representation of the problem-at-hand in Fig. 1 in terms of concept and role instances in the ABox:

- Role(Team Sector)
- Role(Executive Controller)
- Role(Planning Controller)
- Agent(Bob)
- Goal(Ensure traffic safety in its sector)
- Goal(Manage traffic in sector)
- Goal(Manage inbound traffic)
- Goal(Resolve traffic conflict)
- play(Bob, Executive Controller)
- play(Bob, Planning Controller)
- isPartOf(Executive Controller, Team Sector)
- isAndDecompositionOf(Ensure traffic safety in its sector, Manage traffic in sector)
- isAndDecompositionOf(Ensure traffic safety in its sector, Manage inbound traffic)
- hasPosContribution(Manage inbound traffic, Manage traffic in sector)
- provide(Executive Controller, Resolve traffic conflict)
- DelegationOnExecution(Del-exec1)
- hasDelegator(Del-exec1,Team Sector)
- hasDelegater(Del-exec1,Executive Controller)
- hasDelegatum(Del-exec1,Manage traffic in sector)

4.5 System Architecture

Fig. 6 depicts the architecture of our implemented system. Though this work supports two types of queries (SPARQL and SQWRL), most system components and artifacts are common for both inputs (normal line). The ones with thick lines refer to parts for SPARQL, while dashed lines to SQWRL. In both cases, the implemented system requires the same input SI* model representing the problem-at-hand and a set of SI* models representing patterns.

Since we need some inference capabilities to deduce implicit facts, we use a rule engine (i.e., JESS) that is integrated in Protégé. Essentially, a rule engine takes an input (rules and facts) and produces a model. In SQWRL setting, the input consists of the TBox and ABox defined so far, patterns in SQWRL, and SWRL rules [3]. In this system, the model (produced by the rule engine) contains the resultset of the matching. In SPARQL setting, the input to the rule engine only contains TBox, ABox, and SWRL rules. The rule engine produces a model containing inferred knowledge from available facts and rules. Using the Model-to-OWL library in Protégé, inferred knowledge is added to the knowledge base (TBox and ABox). By means of the OWL-$\mathcal{DL}$ reasoner (e.g., Pellet, JENA), we can query the revised knowledge base to find a match to a pattern. In both settings, if the length of resultset is zero, then there is no match found in

[3] available at:`http://www.w3.org/Submission/SWRL/swrl.owl`

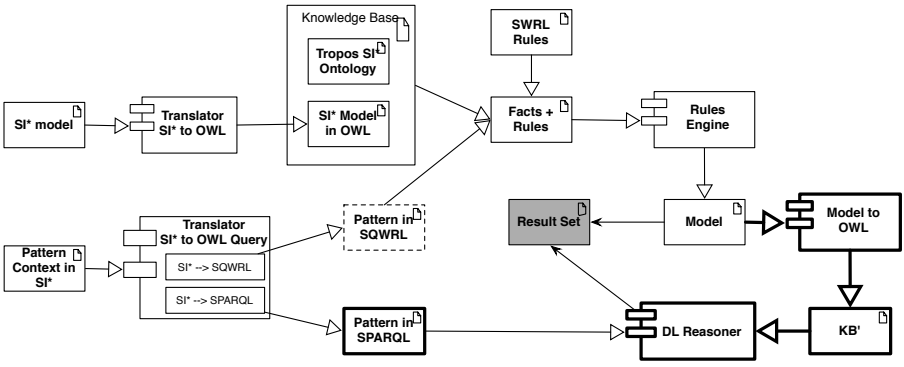


Fig. 6. System Architecture for Pattern Matching

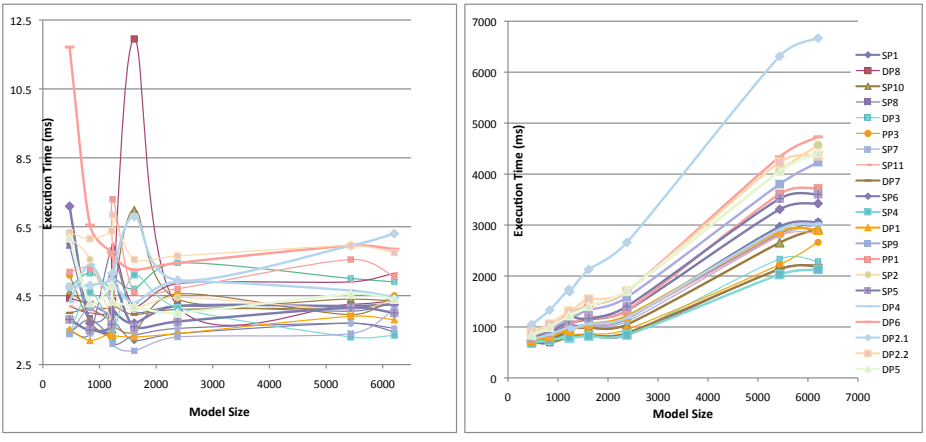
the problem. The resultset will contain several sets when a pattern matches to several parts of the problem. Moreover, each set will provide a mapping from a pattern's constructs to the problem. We have implemented this approach using Java Platform v1.6 along some features from Protégé libraries.

5 Experimental Results

Design of experiments. To evaluate this approach and its implementation, we have conducted experiments using a laptop Intel Core2 Duo T7300 2.0GHz, 2Gb DDR2 667. Through these experiments we intend to assess the performance of our implementation, and investigate how performance (i.e., execution time) is affected by an increase in problem size. To make the experiment realistic, we consider the ATM Scenario [17] as the problem-at-hand. First, the SI* model of the problem is translated to a corresponding model in OWL-$\mathcal{DL}$. Similarly, we translate SERENITY S&D patterns (21 patterns), defined in [6], into OWL-$\mathcal{DL}$ queries in SPARQL and SQWRL. The model is then queried using SPARQL and SQWRL queries to find matches to those patterns. Bigger models are obtained by cloning the OWL-$\mathcal{DL}$ model of ATM scenario facilitated by the "deep copy" feature of Protégé. Originally, the model of ATM scenario has the size of 472 elements composed of 83 nodes and 389 relations [4]. Cloning was performed on the ATM model by cloning nodes and their respective relations. The cloning process was not linear; as we could not control the number of relations a node participates in. Seven models were obtained through this process, starting from a model size of 832 (136 nodes) up to the biggest model with size 6203 (941 nodes).

In the experiment, each pattern is matched against 8 different models. To ensure stability of "execution time", we perform 20 executions for each pair (pattern, model) and used the average of each execution time. Moreover, a manual verification has been performed to validate the correctness of each pattern match.

[4] All datasets can be found at `http://disi.unitn.it/~yudis/lib/exe/fetch.php?media=files:dataset.rar`



(a) Execution time in SPARQL (b) Execution time in SQWRL

Fig. 7. Performance of Pattern Matching

Results. After running the queries (patterns) over the ATM model we found that there are four applicable patterns (e.g., SP1, SP8, DP2.1, DP6). In Fig. 7, we present the performance in milliseconds, of our implementation for both: SPARQL and SQWRL.

In general, there are significant differences between the two query representations. In particular, the worst performance in SPARQL (11.7ms) is much faster then the best performance in SQWRL (713ms). The main reason is that in the case of SPARQL queries, the inferred model is computed only once before the matching starts and used throughout all the queries. Thus, the inference time is not taken into account in the SPARQL execution time. In the SQWRL case, execution time is highly affected by the inference time.

Considering Fig. 7(b), it is an almost linear correlation between the size of the problem model and execution time. However, Fig. 7(a) indicates that the SPARQL performance is constant after a certain model size (model size $\geq$ 2378) [5]. Even though the SPARQL case outperforms the SQWRL case, designers need to be aware on the fact that SPARQL engine does not exploit the semantics of OWL-$\mathcal{DL}$. Moreover, SPARQL is meant to be used for querying RDF and OWL-$\mathcal{DL}$ needs to be serialized before it can be queried. This serialization of OWL-$\mathcal{DL}$ to RDF is vendor specific therefore it could be the case that the same OWL-$\mathcal{DL}$ has several representations in RDF and consequently different SPARQL queries. However, this dilemma does not hold for SQWRL.

6 Related Work

The growing size of pattern libraries has spawned the following challenges: 1) finding a relevant pattern in the pattern library, 2) selecting a pattern that is suited with the problem-at-hand, and 3) applying a pattern. For the first challenge, though there is no central index as mentioned in [3] several initiatives are trying to collect software design

[5] We acknowledge some irregularities on the execution time for the 3 smallest models.

patterns (e.g., Pattern Forge [6], Net Objectives [7], Portland Pattern Repository [8]). In comparison to our approach, such initiatives receive a pattern contribution in natural language without formalizing it. In addition to disadvantages presented in Section 4.2, users might have difficulties in finding relevant patterns in the library because textual matching does poorly without domain knowledge.

To improve the finding and the selection phase, several works use (semi-)formal languages for representing patterns and selection mechanisms of such representation. In [18], Mens et al. use DL to detect inconsistencies between UML models in evolving systems. In that work, the authors take advantage of the underlying DL representation and reason about UML models exploiting the DL reasoning engine (e.g., Racer, Loom). In a nutshell, this work takes a similar approach to ours where the TBox formalizes the UML meta-model and the ABox represents instances in the designers' model. In our work, the query represents the context of a pattern, while in this work the query represents the rules characterizing model constraints. In [19], the authors describe how to use a meta-model to obtain a representation of a pattern at the code level. The meta-model consists of a set of entities and interaction rules between them, and defines pattern semantics. The meta-model is further specialized by adding structural and behavioral constituents, thereby obtaining an abstract representation of patterns. These are gathered in a repository and used to generate code automatically.

Some works facilitate the selection phase by structuring the pattern library in a certain manner. In [20,21], the authors proposed a structure to organize patterns. Moreover, other authors [22,23] provide systematic and automated reasoning to select a pattern. In these works, the authors do not formalize the pattern itself, but rather formalizing the structure and relationships among patterns. Conversely, our approach formalizes a pattern and does not prescribe a particular structure on the pattern library. In our approach, we aim to find an applicable pattern and provide a mapping, while these works intends to limit the solution space so that the pattern users need only evaluate a small number of patterns. In other words, these approaches require less efforts in contributing a new pattern because they only require where a pattern should be categorized and its relationships with other patterns, while in our approach "the formalization" of a pattern defines the performance, in term of correctness, of the system. Note that these approaches does not guarantee the resulted pattern will be applicable to the problem-at-hand, while ours

Some works concentrate on how to apply the patterns in the problem-at-hand. For instance, Eden et al. [24] represent patterns as meta-programs that modify other code (i.e., the problem-at-hand). The authors have implemented a prototype that supports design pattern specification and realization in a given program and this approach allows programmers to edit the source code at any time in the process. In comparison to our approach, this work aims at modifying the problem before implementing a chosen pattern, whereas ours aims to find the pattern(s) that are applicable to solve a given problem.

In the area of Model-Driven Engineering, several frameworks have been proposed to support model transformations [25]. In comparison to ours, their approaches are more expressive in describing how a pattern is to be applied. However, these frameworks

[6] http://www.patternforge.net/
[7] http://www.netobjectives.com/PatternRepository
[8] http://c2.com/ppr/

have some limitations in finding a match because matching is based on graph similarity techniques only, rather than inference in a DL. More generally, our framework can leverage reasoning provided by DL to support pattern matching and pattern application.

7 Final Remarks

We have presented an approach to formalize problems and patterns using Description Logics, so that, given a problem, we can find applicable patterns from a pattern library. Moreover, when a pattern match succeeds, it provides mapping between elements of the problem and variables in the pattern. These mapping are useful in determining how to apply the pattern to a given problem. Our proposal has been evaluated in terms of a case study using the SERENITY pattern library. Our experiences suggest that description logics do constitute a viable solution to formalize patterns, and the problem represented by a rich modeling language such as SI* can be accommodated in a description logic using its concept definition facilities. A corollary of our case study is that there is an important trade-off in formalizing patterns between making them too generic or too specific. Generic patterns match in many contexts but offer vanilla solutions. Conversely, specific ones match few concepts but offer insightful solutions. Pattern designers need to tread carefully as they navigate between these alternatives.

Our future work includes applying our framework to other pattern libraries. In addition, we propose to conduct a controlled experiment to empirically evaluate our approach with pattern designers and pattern users.

Acknowledgments

The research leading to these results has received funding from the EU FP7 under grants no. 216917 MASTER, no. 256980 NESSoS, and no. 257930 ANIKETOS.

References

1. Alexander, C., Ishikawa, S., Silverstein, M.: A pattern language. Oxford Press (1977)
2. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Longman Publishing Co., Inc., Reading (1995)
3. Manolescu, D., Kozaczynski, W., Miller, A., Hogg, J.: The growing divide in the patterns world. IEEE Software 24(4), 61–67 (2007)
4. Sommerville, I.: Software Engineering, 7th edn. Addison Wesley, Reading (May 2004)
5. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F. (eds.): The description logic handbook: theory, implementation, and applications. Cambridge University Press, New York (2003)
6. Asnar, Y., Bryl, V., Dalpiaz, F., El-Khoury, P., Felici, M., Halas, H., Krausová, A., Li, K., Riccucci, C., Saidane, A., Séguran, M., Yautsiukhin, A.: Final set of S&D Patterns at Organizational Level. Project Deliverable A1.D3.3, SERENITY Consortium (January 2009)
7. Horrocks, I., Patel-Schneider, P.F., Boley, H., Tabet, S., Grosof, B., Dean, M.: SWRL: A Semantic Web Rule Language Combining OWL and RuleML (May 2004), `http://www.w3.org/Submission/2004/SUBM-SWRL-20040521/`

8. Hanmer, R.: Patterns for Fault Tolerant Software. Wiley, Chichester (2007)
9. Schumacher, M., Fernandez-Buglioni, E., Hybertson, D., Buschmann, F., Sommerlad, P.: Security Patterns: Integrating Security and Systems Engineering, 1st edn. Wiley, Chichester (2006)
10. Giorgini, P., Massacci, F., Mylopoulos, J., Zannone, N.: Requirements Engineering for Trust Management: Model, Methodology, and Reasoning. IJIS 5(4), 257–274 (2006)
11. Asnar, Y., Giorgini, P., Mylopoulos, J.: Goal-driven risk assessment in requirements engineering. REJ, 1–16 (2010)
12. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF (February 2004), http://www.w3.org/TR/rdf-sparql-query/ (lastchecked: March 7, 2010)
13. O'Connor, M., Das, A.: SQWRL: a query language for OWL. In: Proc. of OWLED 2009 (2009)
14. McCarthy, P.: Search RDF data with SPARQL: SPARQL and the Jena Toolkit open up the semantic Web (May 2005), http://www.ibm.com/developerworks/library/j-sparql/ (lastchecked: March 1, 2010)
15. Friedman-Hill, E.: Jess Rule Engine, http://www.jessrules.com/ (lastchecked: December 2009)
16. Görz, G.: Description Logics, Knowledge Bases, Formal Ontologies and Data Bases: Content. Lecture Notes (2008)
17. Asnar, Y., Giorgini, P., Massacci, F., Saidane, A., Bonato, R., Meduri, V., Riccucci, C.: Secure and Dependable Patterns in Organizations: An Empirical Approach. In: Proc. of RE 2007 (2007)
18. Mens, T., Van Der Straeten, R., Simmonds, J.: Maintaining consistency between UML models with description logic tools. In: Proc. of UML 2003, Workshop on Consistency Problems in UML-based Software Development II (2003)
19. Albin-amiot, H., gaël Guéhéneuc, Y., Kastler, R.A.: Meta-modeling design patterns: Application to pattern detection and code synthesis. In: Proc. of ECOOP 2001 Workshop Automating Object-Oriented Software Development Methods, pp. 1–35 (2001)
20. Manolescu, D., Kozaczynski, W., Miller, A., Hogg, J.: The growing divide in the patterns world. IEEE Software 24(4), 61–67 (2007)
21. Zdun, U.: Systematic pattern selection using pattern language grammars and design space analysis. Software: Practice and Experience 37(9), 983–1016 (2007)
22. Gross, D., Yu, E.: From Non-Functional requirements to design through patterns. Requirements Engineering 6(1), 18–36 (2001)
23. Weiss, M., Mouratidis, H.: Selecting security patterns that fulfill security requirements. In: 16th IEEE International Requirements Engineering, RE 2008, pp. 169–172 (2008)
24. Eden, A.H., Yehudai, A., Gil, J.: Precise specification and automatic application of design patterns. In: Proc. of ASE 1997, pp. 143–152 (1997)
25. Czarnecki, K., Helsen, S.: Classification of model transformation approaches. In: Proc. of OOPSLA 2003 Workshop on Generative Techniques in the Context of the MDA (2003)