Scenario Testing using Formal Ontologies

Hendrina F Harmse, Katarina Britz, Aurona Gerber, and Deshendran Moodley

Centre for Artificial Intelligence Research, CSIR Meraka and University of KwaZulu-Natal, Pretoria, South Africa

Abstract. One of the challenges in the Software Development Life Cycle (SDLC) is to ensure that the requirements that drive the development of a software system are correct. However, establishing unambiguous and error-free requirements is not a trivial problem. As part of the requirements phase of the SDLC, a conceptual model can be created which describes the objects, relationships and operations that are of importance to business. Such a conceptual model is often expressed as a UML class diagram. Recent research concerned with the formal validation of such UML class diagrams has focused on transforming UML class diagrams to various formalisms such as description logics. Description logics are desirable since they have reasoning support which can be used to show that a UML class diagram is consistent/inconsistent. Yet, even when a UML class diagram is consistent, it still does not address the problem of ensuring that a UML class diagram represents business requirements accurately. To validate such diagrams business analysts use a technique called scenario testing. In this paper we present an approach for the formal validation of UML class diagrams based on scenario testing. We additionally provide preliminary feedback on the experiences gained from using our scenario testing approach on a real-world software project.

1 Introduction

One of the most difficult challenges of the Software Development Life Cycle (SDLC) is to ensure that the business requirements are correct [1]. The later errors are detected during the SDLC, the more costly they are to correct [2]. In this paper we illustrate how the formalization of scenario testing can be used to create UML class diagrams that accurately represent the business requirements.

The definition of a scenario test is based on the definitions of a use case and a scenario. A *use case* is a set of actions performed by a software system to produce an observable result. Typically, a use case consists of a number of *scenarios* [3][4][5]. According to Gomaa "[a] scenario is one specific path through a use case". The main scenario of a use case describes the most common path through a use case and alternative scenarios describe the less-frequent paths through a use case [4]. A test based on a scenario is called a *scenario test*.

Depending on the complexity of the requirements, testing even a single scenario or group of scenarios can still be error prone. Being able to apply formal reasoning procedures to a conceptual model provides the benefit of having

both consistencies and inconsistencies illuminated. As mentioned, UML class diagrams are often used to represent the conceptual model, and a formal means to describe such models is therefore desirable. Indeed, several means exist for describing UML class diagrams formally. In this paper we focus our attention on formal ontologies based on OWL 2. OWL 2 has a model theoretic semantics that corresponds to the description logic (DL) $\mathcal{SROIQ}(\mathcal{D})$ [6].

Description logics (DLs) are decidable and complete fragments of first-order logic that are specifically designed for the conceptual representation of an application domain in terms of classes and relationships between classes [7, 8]. A DL knowledge base generally consists of the TBox and the ABox. A TBox is used to define concepts and relationships between concepts and an ABox is used to assert knowledge regarding the domain of interest, i.e. that an individual is a member of a concept [9]. Cali, et al. [7] and Berardi, et al. [8] laid the foundation for describing UML class diagrams in DLs. Cali, et al. described UML class diagrams in the DL $\mathcal{DLR}$ [7]. Berardi, et al. extended this work by describing UML class diagrams in the DLs $\mathcal{DLR}_{ifd}$ and $\mathcal{ALCQI}$ [8]. Recent research described UML class diagrams in OWL 2 [10].

Research to date has shown how to transcribe UML class diagrams into formal ontologies, which enables modelers to check their models for consistency [7, 8, 10]. Yet, even when a UML class diagram is consistent, it may not represent the business requirements accurately. Our contribution is to present a scenario testing approach using formal ontologies to validate that a UML class diagram represents the business requirements accurately. Our scenario testing approach is valuable because it adds formal reasoning procedures to the well-established industry practice of scenario testing [3, 4].

In addition to considering UML class diagrams, our scenario testing approach specifically includes UML object diagrams for representing different scenarios. The UML specification is not overly prescriptive and allows objects to be present in UML class diagrams and class definitions to be present in UML object diagrams [11]. In this paper we make the assumption that UML class diagrams only contain classes while UML object diagrams only contain objects. With this assumption in place, we note that, loosely speaking (due to the UML specification not being restrictive), a TBox corresponds with a UML class diagram and an ABox corresponds with a UML object diagram. Prior research on reasoning on UML class diagrams using DLs has focused on reasoning on UML class diagrams (resp. TBoxes) alone whilst our approach includes reasoning on UML object diagrams (resp. ABoxes) [7, 8, 10].

In this paper we do not provide detail introductions on UML class diagrams [3, 12, 11], DLs [9], OWL 2 or the transformation of UML class diagrams to OWL 2 [13], but instead refer the reader to the references mentioned. Later in the paper we will make use of the ontology tool Protégé [14] which we also assume the reader is familiar with.

We provide an example of a subdomain from the hospitality industry in Sect. 2. In Sect. 3 we explain our scenario testing approach, which we illustrate via the example provided in Sect. 2. We give an evaluation of scenario testing

in Sect. 4, with a discussion of the benefits and challenges of scenario testing, and present feedback on a case study wherein we subjected a real-world software project in the hospitality industry to scenario testing. In Sect. 5 we review related research and clarify the gap addressed by our research. Finally, we give a summary of our findings, and make suggestions for further research, in Sect. 6.

2 Example Business Domain

The example business domain we describe here is a much simplified version of the real-world software project that has been done for a South African hotel group where scenario testing has been employed to detect and rectify errors during the requirements phase of the SDLC. In the hospitality industry the business requirements around calculating the rate that needs to be charged when a reservation is made, generally referred to as a room rate, can be extremely complex. Before a room rate can be calculated, the different configurations that are used to determine these rates have to be specified. In this example, we will focus on the model representing rate configurations. In the following we give a brief description of the relevant concepts and business constraints.

The basic rate that is applicable to a room can be determined in three different ways. Firstly a flat rate can be charged across all rooms in the hotel. Secondly the rate can be charged based on the number of guests who form part of the booking. In the hospitality industry this is often referred to as PAX. Lastly the rate can be determined based on the kind of room that is booked, i.e. single room or suite.

The basic rate that applies to a room can be adjusted based on additional criteria that apply to a room. For instance a different rate will apply when two rooms are booked that are connected by an interleading door. When a reservation is made through a partner network (commonly referred to as a channel) a discounted rate may apply. Similarly when a guest makes a block booking, say of more than 10 rooms per night, a different rate may apply. In order to keep our example simple, we will assume that basic rate charges are always adjusted.

Additional business rules are that PAX charges can only be adjusted by channel criterion, while a room type charge can be adjusted by block booking or channel criteria. Hotel charges can be adjusted by any criteria. Failure to adhere to these business rules will give rise to opportunities for fraud. In Fig. 1 we present the UML class diagram that was initially created to represent the rate configuration business requirements.

3 Scenario Testing Approach

In this section we provide an overview of the scenario testing approach we propose (3.1) and we explain the different techniques in which scenario tests can be applied using formal ontologies (3.2). We show how scenario testing can be applied to the example in Sect. 2 to refine the model (3.3) and we provide example OWL 2 translations (3.4).

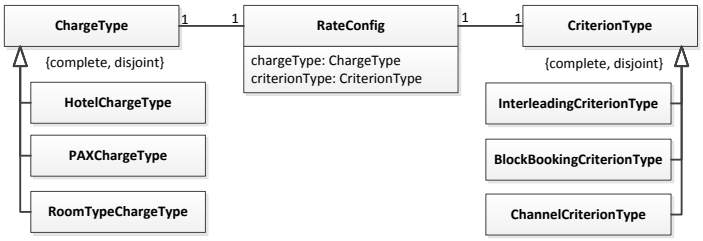

Fig. 1. Initial model of rate configuration

3.1 Overview of Approach

The steps a modeler will take in applying scenario testing are the following:

1. Based on the requirements gathered, the modeler will create a UML class diagram similar to the diagram depicted in Fig. 1.
2. The UML class diagram can be translated to OWL 2 and checked for consistency using Protégé. If any inconsistencies are found, the modeler must go back to step 1 and correct the model. If no inconsistencies are found, we know that the model is consistent and we continue to step 3.
3. We still need to determine whether the model represents the business requirements correctly and hence we apply the scenario testing techniques we introduce in Sect. 3.2. If a scenario test shows that the business requirements are not met, the model needs to be corrected and we therefore go back to step 1.
4. Step 3 is repeated until there are no further scenario tests to apply. This completes the scenario testing process.

Existing research has focused on checking the UML class diagram (resp. in DLs the TBox) for consistency. Scenario testing using formal ontologies extends existing research by checking consistency of UML object diagrams (resp. ABox in DLs) for the UML class diagram that has been created in step 1.

3.2 Scenario Testing Techniques

We define three techniques for doing scenario testing: *consistent scenario tests*, *inconsistent scenario tests* and *classification scenario tests*. Each scenario test corresponds with an UML object diagram (resp. ABox in DLs).

Consistent Scenario Tests A scenario that is allowed in a particular business context can be tested for consistency.

Inconsistent Scenario Tests Certain scenarios may not be allowed in a given business context and therefore we want to test that these scenarios are indeed inconsistent. Inconsistent scenarios are useful to ensure that the business rules are defined sufficiently to disallow such scenarios.

Classification Scenario Tests When different classes in a UML class diagram share the same attributes and associations, it could indicate the presence of redundancies or ambiguities. These concerns can be investigated by applying classification to object instances with the appropriate associated features.

3.3 An Example of applying Scenario Testing

It is trivial to confirm that the model in Fig. 1 is indeed consistent. Even so, the model allows business scenarios that should be disallowed. To make the scenarios of the business requirements explicit, we have compiled them in Table 1. Each row in the table represent a possible rate configuration with the last column in the table indicating whether the rate configuration should be allowed or not.

Based on these scenarios we have redesigned the rate configuration model in Fig. 2 to accurately represent the rates business requirements. The refined model follows intuitively from the scenarios in Table 1 as can be seen for example from the class `InterleadingHotelRateConfig` which corresponds with the allowed rate configuration of row 1. Similarly all allowed rate configurations are made explicit in the model by adding classes that represent the specific rate configurations.

Table 1. Allowed/disallowed rate configuration scenarios

No	Criteria Type				Charge Type		Allowed or Disallowed
	Interleading	Channel	Block booking	Hotel	PAX	Room type	Scenario
1	X			X			Allowed
2		X		X			Allowed
3			X	X			Allowed
4	X				X		Disallowed
5		X			X		Allowed
6			X		X		Disallowed
7	X					X	Disallowed
8		X				X	Allowed
9			X			X	Allowed

3.4 Example OWL 2 Translation

Translating the UML class diagram into OWL 2 is based on the work done by Berardi et. al. [8] and Zedlitz et. al. [10]. The only UML class diagram feature that we are using in Fig. 2 which is not addressed by either Berardi et. al. or Zedlitz et. al. is attribute redefinition. For a detailed discussion on the exact semantics we refer the reader to the paper by Bildhauer [12]. It is important to note that according to the UML specification an attribute can be represented by an association [11], a fact which is indeed confirmed by the UML class diagram

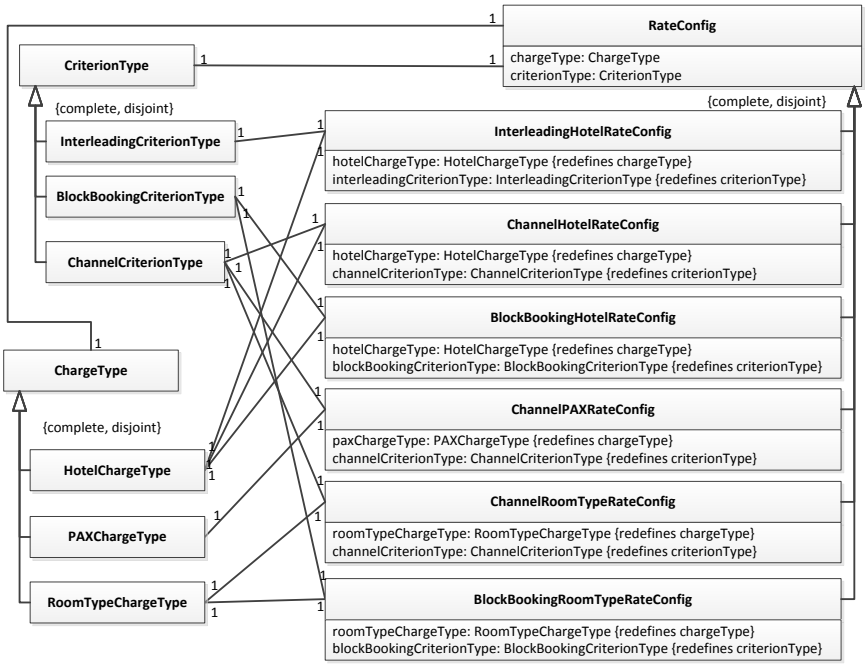

Fig. 2. A model that represents the business requirements of rate configuration accurately

```
ObjectProperty: hotelChargeType
       SubPropertyOf: chargeType
       Domain: BlockBookingHotelRateConfig or ChannelHotelRateConfig
                  or InterleadingHotelRateConfig
       Range: HotelChargeType
```

Fig. 3. The OWL 2 translation of the `hotelChargeType` attribute

translation to DLs [7, 8, 10]. For illustration purposes, we provide the OWL 2 translation of `hotelChargeType` in Fig. 3.

Scenarios are defined as assertions on individuals. As an example we consider the disallowed scenario where a rate configuration is defined as consisting of an `InterleadingCriterionType` and `PAXChargeType` in Fig. 4.

```
Individual: interleading
       Types: InterleadingCriterionType
Individual: pax
       Types: PAXChargeType
Individual: interleadingPAXRateConfig
       Types: RateConfig
       Facts: interleadingCriterionType interleading, paxChargeType pax
```

Fig. 4. The OWL 2 translation of the `hotelChargeType` attribute

The main concern we have with the model in Fig. 1 is that it will not prevent disallowed scenarios. Fig. 5 shows the result of running the reasoner on the refined model after setting up the disallowed scenario of Fig. 4.

RateConfig **DisjointUnionOf** BlockBookingHotelRateConfig, BlockBookingRoomTypeRateConfig, ChannelHotelRateConfig, ChannelPAXRateConfig, ChannelRoomTypeRateConfig, InterleadingHotelRateConfig

interleadingPAXRateConfig paxChargeType pax

paxChargeType **Domain** ChannelPAXRateConfig

interleadingPAXRateConfig interleadingCriterionType interleading

interleadingCriterionType **Domain** InterleadingHotelRateConfig

Fig. 5. Protégé explanations for disallowed scenario of Fig. 4

4 Evaluation

Here we discuss the benefits and limitations of scenario testing (4.1) as well as the benefits the use of formal ontologies brings to scenario testing (4.2). We had the opportunity to use scenario testing on a project in the hospitality industry on which we provide feedback (4.3).

4.1 Benefits and Limitations of Scenario Testing

Using scenarios to test the consistency/inconsistency and completeness of a domain of the business has a number of advantages.

Capturing requirements via scenarios, and validating the captured requirements through scenario testing, are practices that are well-established within the software industry [3]. Applying formal reasoning procedures to scenario testing reduces the margin of error while being closely aligned with how practitioners work. This, in turn, reduces the learning curve of adopting our approach.

Limiting the checking of UML class diagrams to a scenario test, or group of scenario tests, allows for a better understanding of the nature of inconsistencies: Explaining why a specific scenario test fails is more readily understood by stakeholders than explaining mathematical logic concepts such as consistency.

A limitation of a scenario testing is that the resulting model will only be as good as the scenarios that have been considered. If, for example due to time constraints, only a subset of scenarios are considered, the possibility exists that some disallowed scenarios, redundancies or ambiguities may not be discovered.

4.2 Benefits of using Formal Ontologies for Scenario Testing

There are a number of benefits gained from the use of formal ontologies for scenario testing. Firstly, scenario testing relies on traditional testing approaches which have no formal reasoning support. Thus, traditional testing approaches can at most identify problems, but they cannot show their absence. Formal ontologies are equipped with formal reasoning procedures, which can prove consistency.

Secondly, based on the reasoning procedures used, it is possible to provide proofs of why a scenario test is consistent or inconsistent. The availability of such proofs enables modelers to know that their models are consistent for the correct reasons rather than merely knowing that their model is consistent.

Thirdly, logical entailment can be used to gain deeper insight into the implicit consequences of the model. As an example, when a scenario that should be allowed is inconsistent, logical entailment can shed light on the cause of the inconsistency. In the absence of logical entailment, stakeholders can only guess at the reasons for the inconsistency.

Fourthly, due to the mathematical basis of formal ontologies, they do not suffer from the ambiguities that plague the UML specification [5]. Indeed, the formal ontology translation of UML class diagrams can be used to make the intended meaning of a UML class diagram explicit.

Fifthly, formal ontologies make it possible to validate the business specification before the business specification is implemented in software. Discovering and remedying defects during the requirements phase is more cost-effective than during any subsequent phase [2].

Lastly, in our discussion we represented business rules in a UML class diagram which we translated to OWL 2 on which we applied our scenario tests. The value of UML is that it provides a graphical representation that is easier to comprehend than a textual description. However, it is completely viable to apply scenario tests directly on an ontology without using UML at all.

4.3 Adoption and Preliminary Feedback

We have been able to apply our scenario testing approach on a real world software project in the hospitality industry. A business analyst (BA) was given instruction in translating UML class diagrams to Protégé. The BA and client started with an initial UML class diagram which was translated into OWL 2. Using Protégé they set up a number of scenario tests which quickly showed that the initial UML class diagram did not represent the required business rules adequately. Interestingly, at this point they abandoned the UML class diagram in favour of Protégé. Occasionally they drew portions of the UML class diagram to help

guide their thinking. In this manner they incrementally created the conceptual model and validated it with a number of scenario tests. Once they completed the conceptual model in Protégé, the BA translated the classes and relationships to a UML class diagram.

At the time of writing the resulting conceptual model consisted of 62 classes which have been validated by 100 scenario tests. The scenario tests consisted of 43 valid, 47 invalid and 10 classification scenario tests. The BA used the resulting UML class diagram to explain the business logic to the development team. Both developers and testers frequently referred to this UML class diagram throughout the development process.

This experience alerted us to opportunities for improvement. Firstly, the availability of tools to convert between UML and OWL 2 will reduce the effort of translation. Zedlitz, et al. [10] have suggested such a tool, but it will need to be extended to fit the needs of scenario testing. Secondly, scenario testing has been instrumental in providing a clear understanding of the business requirement on this project, which benefited both developers and testers. However, the model relied on multiple inheritance which cannot be represented directly in many programming languages [5] and it is not clear how to translate notions like {`disjoint, complete`} into code. Guidelines in this regard will be valuable.

5 Related Research

Various approaches exist for validating UML class diagrams based on generated instances [15–17]. Cabot, et al. [16] encodes UML class diagrams as constraint satisfaction problems (CSP) and then generates instances of the model using their UMLtoCSP tool which passes the instances to a constraint solver for verification. UMLtoCSP generates a UML object diagram of the object instances that satisfies the UML class diagram. Soeken, et al. [17] follow a similar approach using C++ code to generate instances and a SAT solver to do validation. For both approaches decidability is achieved by definition of a finite solution space and therefore both approaches are decidable, but incomplete. That is, results are only conclusive when a solution is found. When a solution is not found, a solution may still exist in some other finite solution space [16, 17].

Braga, et al.[15] applies scenario testing to OntoUML conceptual models which are translated into Alloy, a logic based language. OntoUML is a UML profile which extends the UML class diagram metamodel with Unified Foundation Ontology (UFO) elements. UFO defines the ontological foundations for the most fundamental concepts in structural conceptual modeling [15]. Alloy is defined as "a structural modeling language based on first-order logic, for expressing complex structural constraints and behavior" [18]. In the approach of Braga, et al. the Alloy analyzer is used to automatically generate instances and counterexamples of the model which are presented to the modeler [15]. The Alloy logic is based on first-order logic (and in particular relational calculus) which gives rise to undecidability. Tractability is achieved by specifying a scope, which means a counterexample may possibly be found given a larger scope [18].

Our approach is different in the following ways. Firstly, since OWL 2 is based on the DL $\mathcal{SROIQ}(\mathcal{D})$, reasoning on OWL 2 is decidable and complete [19]. Thus, theoretically it is possible to get an answer on whether a knowledge base is consistent, but due to tractability concerns there is a practical limit to the size of the knowledge base on which reasoning is feasible [6]. Secondly, we rely on the presence of a domain expert for guiding the definition of scenario tests. None of the approaches cater for the explicit definition of scenario tests [15–17] and hence, there is no way to ensure that scenario tests with high business impact are indeed considered. Furthermore, even when a model is consistent, it may not represent the business requirement accurately. The explicit specification of scenario tests can help alert the modeler to this occurrence.

6 Conclusion

In this paper we showed that our formal verification approach based on scenario testing can be used to support and supplement the development of accurate requirements represented in UML class diagrams. The benefit of this is that some ambiguous and incomplete requirements could be detected before system development commences.

Initial feedback based on a real-world software product looks promising; but some challenges remain. Guidelines are needed on how to accurately translate the conceptual model into code. Furthermore, the availability of tools, for doing translations between UML class diagrams and OWL 2, will greatly improve the user experience.

The fact that scenario testing is based on object instances rather than classes alone, presents new opportunities for representing UML class diagram features in formal ontologies. In related research we are investigating translating identity constraints on UML class diagrams using Easy Keys [20] for use in scenario testing. It will be interesting to explore what other UML class diagram features can benefit similarly.

7 Acknowledgements

This work is based on research supported in part by the National Research Foundation of South Africa (Grant No. 85482).

References

1. F. P. Brooks. No Silver Bullet: Essence and Accidents of Software Engineering. *Computer*, 20(4):10–19, 1987.
2. B. Boehm and V.R. Basili. Software Defect Reduction Top 10 List. *Computer*, 34(1):135–137, January 2001.
3. G. Booch, R. A. Maksimchuk, M. W. Engel, B. J. Young, J. Conallen, and K. A. Houston. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley Professional, 3rd edition, April 2007.

4. H. Gomaa. *Software Modeling and Design: UML, Use Cases, Patterns, and Software Architectures*. Cambridge University Press, March 2011.
5. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley, 2nd edition, 2005.
6. B. C. Grau, I. Horrocks, B. Motik, B. Parsia, P. Patel-Schneider, and U. Sattler. OWL 2: The Next Step for OWL. *Web Semantics: Science, Services and Agents on the World Wide Web*, 6(4):309–322, November 2008.
7. A. Calí, D. Calvanese, G. De Giacomo, and M. Lenzerini. Reasoning on UML Class Diagrams in Description Logics. In *Proceedings of the IJCAR Workshop on Precise Modelling and Deduction for Object-oriented Software Development (PMD)*, 2001.
8. D. Berardi, D. Calvanese, and G. De Giacomo. Reasoning on UML Class Diagrams. *Artificial Intelligence*, 168:70–118, 2005.
9. Franz Baader, Bernhard Ganter, Baris Sertkaya, and Ulrike Sattler. Completing Description Logic Knowledge Bases Using Formal Concept Analysis. In Manuela M. Veloso, editor, *IJCAI*, pages 230–235, 2007.
10. J. Zedlitz, J. Jörke, and N. Luttenberger. From UML to OWL 2. In D. Lukose, A. Ahmad, and A. Suliman, editors, *Knowledge Technology*, volume 295 of *Communications in Computer and Information Science*, pages 154–163. Springer Berlin Heidelberg, 2012.
11. ISO. *Information technology - Object Management Group Unified Modeling Language (OMG UML), Superstructure*, iso/iec 19505-2 edition, 2012.
12. Daniel Bildhauer. On the relationships between Subsetting, Redefinition and Association Specialization. In *Databases and Information Systems : Proceedings of the Ninth International Baltic Conference*, Riga, Latvia, 2010.
13. P. Hitzler, M. Krötzsch, B. Parsia, P. F. Patel-Schneider, and S. Rudolph. OWL 2 Web Ontology Language Primer (Second Edition). `"http=http://www.w3.org/TR/owl2-primer/"`.
14. protégé. url=http="http://protege.stanford.edu/".
15. B.F.B. Braga, J.P.A. Almeida, G. Guizzardi, and A.B. Benevides. Transforming OntoUML into Alloy: Towards Conceptual Model Validation using a Lightweight Formal Method. 6(1-2):55–63, March 2010.
16. J. Cabot, R. Clariso, and D. Riera. Verification of UML OCL Class Diagrams using Constraint Programming. *IEEE ICST Workshop*, pages 73–80, April 2008.
17. M. Soeken, R. Wille, M. Kuhlmann, M. Gogolla, and R. Drechsler. Verifying UML/OCL Models Using Boolean Satisfiability. In *Design, Automation and Test in Europe*, pages 1341–1344. IEEE Computer Society, 2010.
18. D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.
19. I. Horrocks, O. Kutz, and U. Sattler. The Even More Irresistible $\mathcal{SROIQ}$. In P. Doherty, J. Mylopoulos, and C. A. Welty, editors, *Proceedings of the 10th International Conference on Principles of Knowledge Representation and Reasoning*, pages 57–67. AAAI Press, 2006.
20. B. Parsia, U. Sattler, and T. Schneider. Easy Keys for OWL. In C. Dolbear, A. Ruttenberg, and U. Sattler, editors, *OWLED*, volume 432 of *CEUR Workshop Proceedings*, 2008.