Rapid Prototyping of Business Rules to Demonstrate Inconsistencies Using Semantic Web Software

Wondaal, L.

Rapid Prototyping of Business Rules to Demonstrate Inconsistencies Using Semantic Web Software

Degree programme:	Open University of the Netherlands, Faculty Science
	Master of Science Business Process Management & IT
Course:	IM0602 BPMIT Graduation Assignment Preparation IM9806 Business Process Management and IT Graduation Assignment
Student:	Lieuwe Wondaal
Identification number:	
Date:	23-01-2022
Thesis supervisor	Dr. Lloyd Rutledge
Second reader	Dr. Ir. Ella Roubtsova
Version number:	1.0
Status:	Final

Abstract

This paper presents a rapid throwaway prototyping technique using Semantic Web technologies that product owners can use to gain valuable insight by quickly experiencing demonstrations of inconsistencies so that the functions present in the resulting end system can be tested. The product owner or client can then determine whether this behaviour matches their expectations. An inconsistency is triggered by a collection of data that violate a rule defined by the developer, indicating that some of the data must be removed. The technique quickly demonstrates how the application of a given logical inconsistency determines system behaviour. Developers can use this to demonstrate the effect of inconsistencies, allowing them to determine whether the inconsistency triggers an interaction with the end user so that the user can be informed. This rapid prototyping technique contributes to the existing body of literature on the Semantic Web. It can also be implemented together with existing software development methodologies to produce systems that better match the expectations of stakeholders. The rapid prototyping technique is applied in use cases to evaluate its effectives in demonstrating inconsistencies.

Key terms

Inconsistencies; business rule system; explanations; Semantic Web.

Summary

This research presents and evaluates a technique for demonstrating inconsistencies by creating rapid throwaway prototypes. The goal of the technique is to present inconsistencies to an end user or stakeholder quickly by creating rapid throwaway prototypes. The stakeholder can use this information to determine whether the software is working as intended and provide feedback in an early stage of the development process. The demonstration is performed by presenting a prototype in two states: one with data that will cause an inconsistency, the other with data that conform to the system's parameters.

The technique consists of several components. The prototypes are developed in the Semantic Web context and are created by importing computer code in Protégé, thus producing a system. To make the process faster, the code is created with the help of the conversion tool s2o. This is done by first creating business rules based on natural language which can be imported into the conversion tool. Throwaway prototypes are chosen so that a stakeholder can provide feedback quickly on existing functionality. This type of prototype works well for this function as it is fast and cheap to produce, thus decreasing the amount of time needed to determine whether the stakeholder needs to make changes regarding, for example, design or scope.

This research does not present new logic but a technique that explains a system to a stakeholder or end user by rapidly demonstrating inconsistencies. The goals of this research are the following:

1. **Present the technique:** Demonstrate that the rapid prototyping technique that we developed can be used to build working prototypes that present inconsistencies to a user, thus helping the user better understand the system.
2. **Demonstrate rapidity:** Show that by applying the rapid prototyping technique, users can gain insight into inconsistencies more quickly than with a classic approach to software development.

The main research question is as follows:

How and to what extent can rapid prototyping using Semantic Web software quickly demonstrate the application of inconsistencies to a stakeholder in a business rule system?

The cause for an inconsistency can be categorized into one of three groups: instance-level, class-level or class-assertion. These groups are used to determine the top-level structure for demonstrating the technique. Three scenarios are the result of this, all three based on a use case to test the technique against different causes of inconsistencies. The following three inconsistency causes are used with the corresponding use cases attached to them:

1. **Cardinality constraint**: Introduce a new EU-Rent customer.
2. **Unsatisfiable class**: Buy a car.
3. **Disjoint class**: Introduce a new EU-Rent customer.

The conclusion is that the technique can be used to demonstrate inconsistencies rapidly. This is mainly due the efficiency of the code-conversion tool as it greatly reduces the amount of manual input needed to create a system. Furthermore, each scenario is easier to develop than the last due to the ability to reuse components of the business rules from previous scenarios. It can also be noted that a person without the technical knowledge of computer code can indirectly make changes to the system by changing the business rules using this technique. These benefits are found across all inconsistency causes.

Contents

1. Introduction

1.1. Business rule systems

A business rule is a statement that influences or guides information and behaviour in an organization. Business rules bridge the gap between business and data. When an organization understands its business rules and applies them appropriately, knowledge can be gained for making key business decisions.

Controlled natural language (CNL) is one method of expressing business rules. CNLs are subsets of natural languages (NLs) engineered to reduce the ambiguity and complexity of full NLs such as German or English. This is done by restricting vocabularies and grammar (Feuto Njonko, Cardey, Greenfield, & El Abed, 2014). The semantics of business vocabulary and business rules (SBVR) provide a way to present concepts and statements in CNLs. The intended use of SVBR is to express business rules and vocabulary in natural language. The framework was developed by the Object Management Group (OMG), a large software consortium with a focus on developing standards for the software engineering industry (OMG, 2017) .

CNL and SVBR are used in this paper to define business rules. SBVR specifically is an important component because there are various tools which allow for business rules that conform to SVBR to be imported and converted to other types of code.

Business rules can be inserted in a business rule management system (BRMS) to use as a basis for an application. Using a BRMS introduces flexibility to the IT architecture and enables developers to easily and quickly change the behaviour of the decisions an application produces. One approach to inserting an organization's business rules in a BRMS is the agile business rule development (ABRD) methodology (Boyer j., 2011), which describes a five-cycle approach as follows:

1. *Harvesting:* The first goal is to understand business entities and document enough rules to begin implementation. A project team must perform business-modelling activities aiming to describe the business processes and decisions applied within the scope of the envisioned business application.
2. *Prototyping:* The second cycle involves the preparation of the structure of the project and outlining how the rules will be organized in a rule set. Once a certain level of discovery is reached, the team can implement the structure of the rule set and begin authoring the rules while analysis and discovery continue.
3. *Building:* The goal of the building phase is to implement test cases with realistic data.
4. *Integrating:* An execution server is deployed to test the process in an end-to-end scenario.
5. *Enhancing:* The rule set is maintained.

Specifically, harvesting and prototyping are important steps in this paper; they serve as a basis in the rapid prototyping technique to demonstrate inconsistencies quickly.

1.2. Data inconsistencies

A data inconsistency is a single instance of conflicting information in a system or a violation of a predefined constraint. Manually identifying and resolving inconsistencies is time consuming and error prone. Across domains, different approaches are used to explain inconsistencies.

One such framework proposes a method called inconsistency repair in which an inconsistency is first located and then resolved by removing or changing the data that caused the inconsistency. Locating an inconsistency starts with finding its source, which is often the most recent change to the system. To understand why it is causing an inconsistency requires contextual information. This is why a common approach is to present the change that caused the inconsistency to the user and let the user choose the solution to solve the conflict (Haase, 2005).

Bahe categorizes system responses to inconsistencies as errors and calls to action. The former constitute a blocking of input when a user enters data that violates system parameters. This gives the user a chance to change the input, for example when only numbers are expected but the user inputs text. A call to action is a batch fix. This occurs when a large amount of data is entered by a user, and the system searches to find inconsistencies. The user is then presented with a list of inconsistencies which can be debugged one at a time (Bahe, 2021).

1.3.Rapid prototyping

One technique for developing business roles is prototyping. This paper presents a technique for rapid prototyping to demonstrate the effect of business rules so that developers and product owners can determine whether this expected result meets their expectations. Rapid prototyping is based on the concept of rapid-application development (Bradley Camburn & Daniel Jensen), which is a term for adaptive software development approaches. In general, RAD puts more focus on an adaptive process than planning. Prototyping is the activity of developing software applications that are incomplete versions of the final envisioned product. It simulates one or a few aspects and may differ completely from the final product. The benefit of prototyping lies in obtaining feedback early on in a project, thus providing time to change requirements during the development process to increase the chances of delivering a successful product.

Agile methods are often used for RAD development, and these in turn use prototyping. This is considered a key tool to develop systems efficiently as it has several advantages compared to traditional methodologies: better quality and risk control and projects are completed on time and on budget (Leo R. Vijayasarathy, 2008).

1.4. The Semantic Web

Software and frameworks from the Semantic Web are used in this paper to demonstrate inconsistencies with a rapid prototyping technique. The Semantic Web is a set of standards set by the World Wide Web Consortium (W3C) as an extension of the World Wide Web. The purpose of the Semantic Web is to make data on the internet readable by machines (Berners-Lee, 2001). To enable this, semantic metadata is added to applications through data-modelling techniques such as Web Ontology Language (OWL) and resource description framework (RDF). The concept of the Semantic Web originates from the earliest days of the World Wide Web and has since evolved with more

features and technologies. One important step for the realisation of the Semantic Web is the standardisation of languages for web knowledge. Currently, these are RDF, RDFS, OWL and SPARQL. RDF is the standard format for interchange in the Semantic Web and is used as a simple yet powerful annotation language for Web resources. This is accomplished by defining triplets: subject, predicate and object (Klyne & Carroll, 2006). RDF is used in this paper.
RDFS and OWL are used to develop vocabularies for metadata with the assumption that different applications are interoperable with each other (McGuinness & van Harmelen, 2004). They are also used to redefine Web resources so they can be exploited by reasoners. In this way, models can be validated, and implicit knowledge can be automatically generated. This is accomplished by defining classes and properties.

1.5. Scope

For the scope of this study, the software Protégé is chosen from the Semantic Web domain as the technological form of the proposed solution. We use rapid prototyping in Protégé to demonstrate inconsistencies that arise from the violation of business rules. The application area of this study is business rule systems, which are information systems based on business rules.

When demonstrating inconsistencies, we focus on what responses the system provides. Two types of responses are defined: errors and calls to action. The former is a response when data is entered that violates a rule or a required parameter. The latter is a response when data input is incompatible within the context of the feature; in this case, the system process will stop until the user removes the data that causes the inconsistency. We also narrow down the type of inconsistencies demonstrated to two categories as follows:

1. **Instance-level causes**. Also called individual assertion inconsistencies, they are caused by conflicting assertions regarding an instance.
2. **Class-level causes**. Caused when individuals relate to an unsatisfiable class. This occurs when a class is related to an individual, while the class should not have any instances related to it (Kalyanpur, Parsia, Sirin, & Hendler, 2005).

These concepts are further described in Section 2.3.

To further narrow down the scope of the research, we present the following main research question: How and to what extent can rapid prototyping using Semantic Web software quickly demonstrate the application of inconsistencies to a stakeholder in a business rule system?

Sub-questions for the literature search:
1. Which rapid prototyping technique is most useful for demonstrating inconsistencies?
2. What are inconsistencies in business rule systems?
3. Which type of inconsistencies will be demonstrated?
4. How will rapid prototyping principles be used to demonstrate inconsistencies?

1.6. Main lines of approach

The theoretical framework (Section 2) describes which methods and tools currently exist to support the detection of invalid data entry, what inconsistencies in the Semantic Web are and which rapid prototyping methods are popular.

In the methodology chapter (Section 3) is described how this study is performed and presents the method of how the study is conducted.

In the results chapter (Section 4), a technique that can demonstrate inconsistencies rapidly is presented, and it is evaluated by bringing it into practice based on three use cases.

In the final chapter, a discussion is presented regarding the steps taken during this study and what findings were derived from the implementation of the technique. Furthermore, the research question is discussed, and recommendations for future research are proposed.

2. Theoretical framework

2.1.Research approach

This chapter describes the literature review conducted to gain a base of knowledge regarding existing research related to this study. The goal of this research is to use a rapid prototyping technique to demonstrate inconsistencies. To achieve this, popular rapid prototyping techniques are analysed in Section 2.2 to provide context for the proposed technique as it will be used in conjunction with a development approach. Section 2.3 describes inconsistencies in business rule systems. Section 2.4 explains the EU-Rent case and presents an investigation into how the case can be used to showcase the benefit of the rapid prototyping approach. In the final Section 2.6, the findings of the literature review are summarized. To narrow the scope of the theoretical framework, the following questions are presented:

Main question: How and to what extent can rapid prototyping using Semantic Web software quickly demonstrate the application of inconsistencies to a stakeholder in a business rule system?

Sub-questions for literature search:

1. Which rapid prototyping technique is most useful for demonstrating inconsistencies?
2. What are inconsistencies in business rule systems?
3. Which type of inconsistencies will be demonstrated?
4. How will rapid prototyping principles be used to demonstrate inconsistencies?

The primary tools used for searching literature is the Open University library and Google Scholar. There is no specific filter for journals but we do look for the most recent sources. The search terms used are the subjects of all the sub-questions for literature search listed above.

2.2.Rapid prototyping for inconsistencies

According to (Hallgrimsson, 2012), prototyping is part of an iterative learning process intended to reduce uncertainties in an early phase by involving users in the development process. By developing multiple prototypes and testing them with customers and users, insights are gained into problems and possible solutions.

The topic of rapid prototyping has been sparsely explored within the domain of the Semantic Web, so it is useful to experiment with rapid prototyping principles to determine how beneficial these can be in the Semantic Web. Prototypes are not always in the form of software but can take many forms such as simple paper scribblings, cardboard models or functional design specifications. Scrum is the most widely used approach in software development to deliver prototypes. With Scrum, the goal is to have a working product at any time. The starting point is a simple version of the product with minimal functionality; then, it is improved during each iteration with the implementation of new features. The improvement is measured by a set of criteria defined by an end user. With every iteration, the software is improved (K Beck, M Beedle, A Van Bennekum, A Cockburn, 2001).

(Forward, Badreddin, Lethbridge, & Solano, 2012) consider three kinds of prototypes:

1. *Throwaway prototypes*: These are considered rapid prototypes and are often used for a proof of concept.
2. *Vertical evolutionary prototypes*: System components are developed with full functionality in every release. This is more suitable for a big data system.
3. *Minimum viable product (MVP)*: This is an evolutionary prototype containing only features considered essential to make the product work, thus minimizing the time spent on each iteration and providing the opportunity to gain more feedback from users.

The approach used in this research to demonstrate inconsistencies closely resembles the throwaway prototyping as it is used to prove a hypothesis and is not subject to user testing to develop additional features. However, principles from MVP are applied to deliver a prototype rapidly. The main principles used are definition of features and committing only to the features that the researcher deems essential to prove the concept. The concept of iteration is also used to determine whether a feature is working as intended.

2.3.Inconsistencies in business rule systems

A business system is defined as a system in which transactions result in consistent states, and its current state satisfies predefined integrity constraints (S. Finkelstein, 2009). A business system becomes inconsistent when integrity constraints are violated. This usually occurs when a set of rules is no longer valid. An inconsistency can be defined as a single instance of conflicting information in a system or a violation of a predefined constraint. This is usually triggered when new data is entered in the system by a user (Haase, 2005).

One method of expressing business rules is the SVBR, the method proposed by the (OMG) The method constitutes a sound standard and was created with the intention of holding any kind of knowledge. SVBR provides meaning-centric, semantically rich and unambiguous capability standard for defining meanings (OMG, 2017).

A method for identifying inconsistencies is proposed by (Chittimalli & Anand, 2016), who use the inherent first order logic (FOL) of SBVR representation. This method can be used to decide the effectiveness of FOL theories and identify conflicting and thus inconsistent rules in SVBR.

In the context of the Semantic Web, business rules can be managed in BRMS. Protégé is such a system and is frequently used, mainly for authoring ontologies, but since ontologies rely heavily on business rules, Protégé has many features and plugins to allow for the creation and editing of rules and the handling of inconsistencies (Noy et al., 2001). For these reasons, Protégé is used extensively in this paper to demonstrate inconsistencies

Three reasons are given by Kalyanpur, Parsia, Sirin and Hendler for how OWL inconsistencies are caused (Kalyanpur et al., 2005). These are important as they form the basis for demonstrating the technique in this paper to ensure that the reasons for inconsistencies are valid. We refer to the following categories for causes of inconsistencies:

1. **Instance-level causes**. The first cause is the individual assertion inconsistency and is the result of conflicting assertions regarding an instance.
2. **Class-level causes**. The second cause is when individuals relate to an unsatisfiable class. This occurs when a class is related to an individual when the class should not have any instances related to it.
3. **Class-assertion causes**. The third cause is a class conflict. This occurs when a statement about classes contains assertions of instances. When these classes and instances are merged, an inconsistency results.

2.4. EU-Rent case

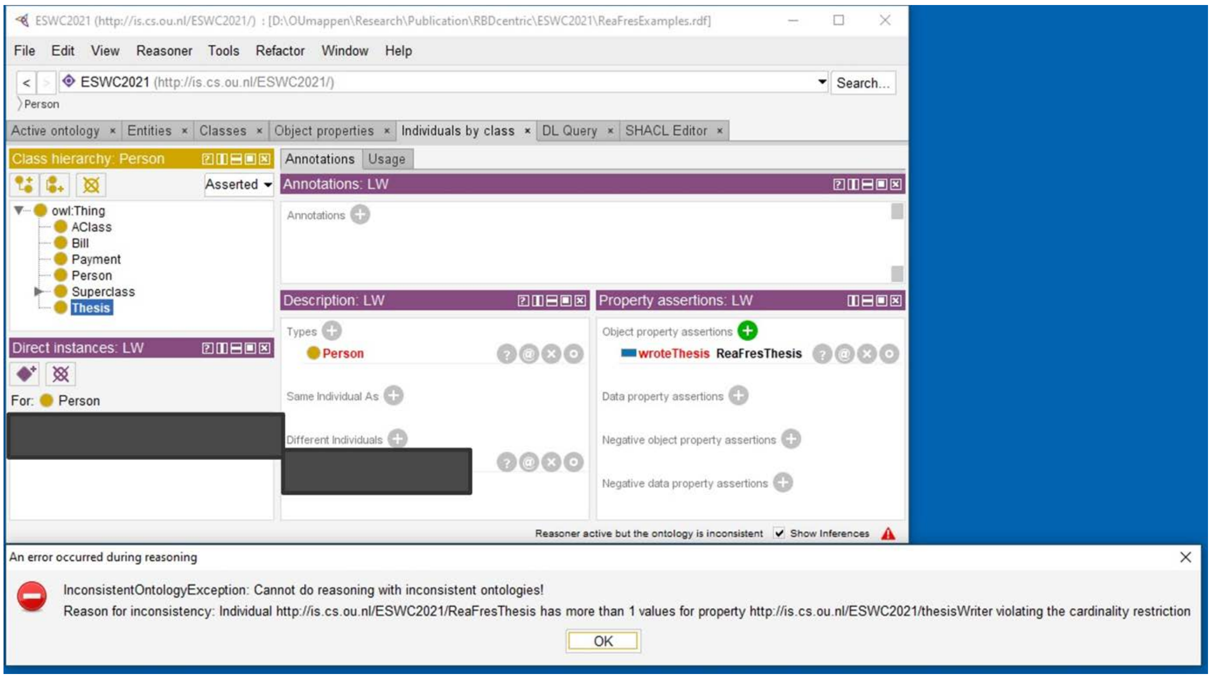
EU-Rent is a fictional international car rental organization which offers a selection of cars to consumers. A consumer can rent a car through a rental agreement. A wide variety of business rules can be derived from EU-Rent, which is why it is often used by researchers to demonstrate models and theories related to the Semantic Web.

Karpovic presents a converter from SVBR to OWL that also creates OWL inconsistencies, but this is not applied to EU-Rent. However, generating OWL inconsistencies from EU-Rent scenarios written in SVBR can theoretically be accomplished (Karpovic & Nemuraite, 2011).

For demonstrating inconsistencies, we refer to use cases based on the specification provided by Frias et al. (Frias, Calafat, & Ramon, 2003). This sets the demonstration in a real-world context, making it more relatable. By employing use cases from a source in the literature, we expect to erect a strong foundation on which to base the reasons for the inconsistencies.

2.5. Conclusion

Business systems may become inconsistent when a user attempts to enter data into the system that violates a rule or constraint or when a larger collection of data has already been entered in which the system finds inconsistencies. After researching the existing body of literature, it can be concluded that not much research exists regarding demonstrating inconsistencies in business rule systems. However, a fair amount of research exists around prototyping (Hallgrimsson, 2012), and based on this, we deduced that it can play an important role in demonstrating inconsistencies in business rule systems. To determine which tool can best be used to demonstrate inconsistencies, we looked at the literature and found that Protégé is a tool often used in Semantic Web research as it has many features and plugins which allow for the creation and editing of rules and handling of inconsistencies (Noy et al., 2001).


ESWC2021 (http://is.cs.ou.nl/ESWC2021/) : [D:\OUmappen\Research\Publication\RBDcentric\ESWC2021\ReaFresExamples.rdf]
File Edit View Reasoner Tools Refactor Window Help
ESWC2021 (http://is.cs.ou.nl/ESWC2021/)
Search...
Active ontology Entities Classes Object properties Individuals by class DL Query SHACL Editor
Class hierarchy: Person
Asserted
owl:Thing
AClass
Bill
Payment
Person
Superclass
Thesis
Annotations Usage
Annotations: LW
Annotations
Description: LW
Types
Person
Same Individual As
Different Individuals
Property assertions: LW
Object property assertions
wroteThesis ReaFresThesis
Data property assertions
Negative object property assertions
Negative data property assertions
Direct instances: LW
For: Person
Reasoner active but the ontology is inconsistent
Show Inferences

[redacted]

[redacted]

An error occurred during reasoning

InconsistentOntologyException: Cannot do reasoning with inconsistent ontologies!
Reason for inconsistency: Individual http://is.cs.ou.nl/ESWC2021/ReaFresThesis has more than 1 values for property http://is.cs.ou.nl/ESWC2021/thesisWriter violating the cardinality restriction

OK

Figure 1. Inconsistency triggered by violation of a business rule in Protégé

Based on how often Protégé has been used in academic studies and the fact that it is available as open source from a university, we concluded that it is a reliable and robust system. This led to the conclusion that Protégé would lend itself well for this study.

The EU-Rent specification (Frias et al., 2003) lends itself well as a background for demonstrating inconsistencies as it has been a proven source of business rules based on its usage in other research.

3. Methodology

This chapter describes how the main research question is answered based on insights from the theoretical framework. In Section 3.1, the research method is explained. Section 3.2 details the technical design and contains use cases and the process of execution. In Section 3.3, we explain how data is gathered and analysed from the EU-Rent case, and in Section 3.4, we discuss the ethical aspects, validity and reliability of this research.

3.1.Conceptual design: Select the research method(s)

This study is based on research methods from design science. This approach has clear guidelines and definitions for research projects that involve information systems. The aim of this study is to present and evaluate a means for rapid prototyping to demonstrate inconsistencies for end users or product owners so they can confirm that the system is working as intended. Figure 1 in Section 2.5 presents a screenshot of how this information is displayed in Protégé. This is a process that requires designing, building and analysis, which makes design science an excellent framework on which to base this study. Using a process based on design science, a prototype is developed and evaluated to measure its effectiveness for demonstrating inconsistencies using indicators that are further described below.

The design science research process of Hevner (2007) describes three cycles: relevance, design and rigor. In the first cycle, the requirements and relevance of the study are described. The rigor cycle is for grounding the study using scientific methods and theories. These two cycles relate to one another through the design cycle in which an actual artefact or process is produced from which results can be derived. The cycles are presented in this study in the following chapters:

- Relevance: Chapter 1
- Rigor: Chapter 2
- Design: Chapter 4

The figure below illustrates the research model used in this study. It is derived from Hevner's process and is the foundation of this study.

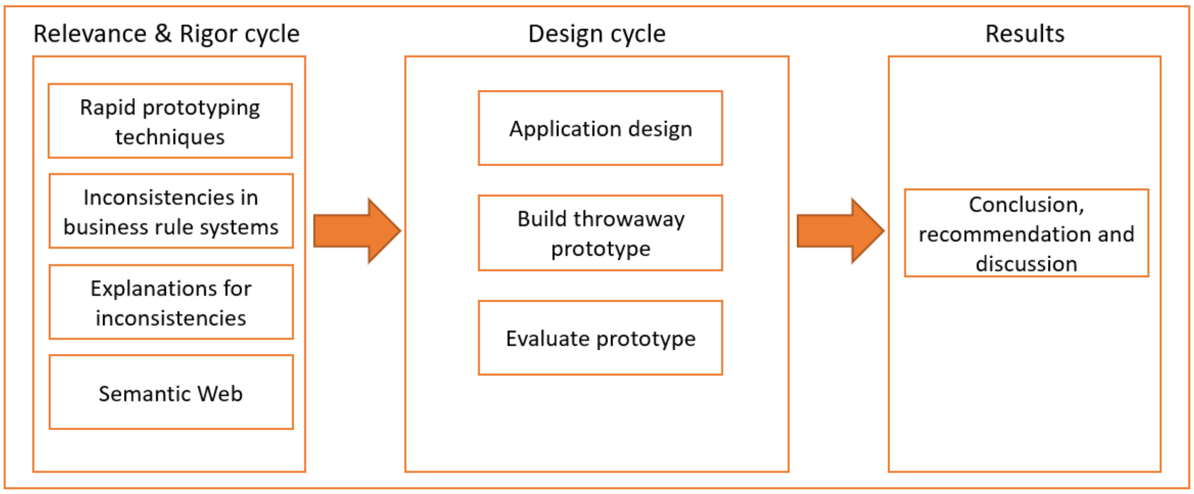

Relevance & Rigor cycle
Rapid prototyping techniques
Inconsistencies in business rule systems
Explanations for inconsistencies
Semantic Web
Design cycle
Application design
Build throwaway prototype
Evaluate prototype
Results
Conclusion, recommendation and discussion

Figure 2. Model of the conceptual design of this study, based on Hevner's design science research process (Hevner, 2007)

The indicators used in the evaluation of the prototype to determine its effectiveness in demonstrating and explaining inconsistencies are as follows:

1. Can the inconsistencies be inferred using logical deduction as described by Herzig, Qamar and Pardis (Sebastian Herzig, 2014)? A logical deduction determines whether a conclusion can be drawn from knowledge of the physical world. For example, an aircraft has 3 and 5 landing gears. This logical contradiction is understandable to an average user. One of those two statements cannot be true. Therefore, an inconsistency exists.
2. Does the inconsistency contain an explanation that hints at the reason it occurred as described by McGuinness (McGuinnes, 2004)? The reason of occurrence is defined as the action that was taken that triggered the inconsistency.

3.2. Technical design: Elaboration of the method

To determine the usefulness of the rapid prototype, several scenarios are defined in which inconsistencies can be demonstrated. The scenarios are based on two options available when facing an inconsistency: an error or a call to action. The former is a response when data is entered that violates a rule or a required parameter, and the latter is a response when data input is incompatible in the context of the feature. In the case of a call to action, the system process stops until the user removes the data causing the inconsistency. The scenarios also take into account different types of user input: manual and automatic. Manual input is when the user submits data that is recorded in the database and triggers a process. Automatic input is a recurring process or a process that is triggered after manual input that changes states, database records and UX documents.

The technical implementation of the scenarios is described in the steps below, which are derived from the design science model of Hevner (Hevner, 2007).

1. **Selecting requirements**: This study develops a technique for rapid prototyping and evaluates it regarding its rapidity and ability to demonstrate inconsistencies.
2. **Existing information:** Section 2 describes the information that was obtained through research as input for the scenarios.
3. **Build and design:** Section 4 details the process and application of the rapid prototyping technique.
4. **Evaluate:** The applicability, quality and efficiency of the artefact must be proven based on good research. The artefact is evaluated with the software Protégé and tested in user scenarios derived from previous studies (Frias et al., 2003).
5. **Demonstration:** The artefact is tested to determine whether it is useful and fit to purpose.
6. **Discussion:** The contribution to the scientific and practical software development communities is reflected upon.

3.3. Data analysis

The data analysed is mostly in the form of business rules created based on the EU-Rent case, which is used in this study. The data comes from fictional user stories in the context of the case; they are analysed with a qualitative method using indicators. The indicators are based on previous research regarding information systems (Sebastian Herzig, 2014) (McGuinnes, 2004). In the final section of the study, the results are summarized and discussed. The advantages and disadvantages are weighed, and a recommendation is provided as to whether the technique is useful and if so, in which cases it can be applied and in which it cannot.

3.4. Validity, reliability and ethical aspects

It is assumed that the internal validity cannot be verified due to a lack of causal relationship in the employed design science research process. However, external validity can be reflected upon due to the generalizability of this study compared to similar studies.

This study can easily be reproduced as the steps taken are thoroughly documented and all applications and methods used are either open source or freely available.

No ethical violations are expected as a result of this study as the material on which the study is based is derived from a fictional company. No real persons or organizations are involved, so the chance of any harmful consequence is minimal.

4. Results

To execute the research described in Section 3, we develop a technique based on the research method described in the technical design section (Section 3.2). Two scenarios are subjected to the approach described in the technical design. The first is a scenario in which an inconsistency is triggered because a user manually submits data that violates a rule. In the second scenario, the user triggers a process that results in invalid data in the database, resulting in inconsistencies. Only two scenarios are chosen to remain within the scope of this research and maintain its feasibility. These scenarios also cover both input methods described in Section 3.2.

The first sub-section covers the technique itself. The other sub-sections each describe a different scenario that uses the same technique. Based on these descriptions, an evaluation is made to determine the effectiveness of the technique across multiple use cases.

4.1. Problem and objective

The aim of this paper is to answer the following research question:

“How and to what extent can rapid prototyping using Semantic Web software quickly demonstrate the application of inconsistencies to a stakeholder in a business rule system?”

To answer the question, we present and evaluate a rapid prototyping technique that involves five steps; it starts by defining the business rule and results in a working throwaway prototype. The intention of the scenarios in this section is to apply the rapid prototyping technique to two categories which represent common causes for inconsistencies (Kalyanpur et al., 2005):

1. Instance-level causes: These are conflicting assertions regarding an instance.
2. Class-level causes: These involve individuals related to a class that should not have any instances.

We initially planned to include a scenario for the third cause category described by Kalyanpur et al.: class-assertion causes. However, this would mean that a class would mistake another class for an individual, and our selected tool Protégé does not allow this at all. Other tools in the Semantic Web could demonstrate this. Instead, we opted for two scenarios in the class-level category.

By applying the technique to both causes, we aim to gather as much information as possible to draw valid conclusions to answer the research question while remaining within the scope and bounds of the research as explained in Section 1.5.

The objectives for these scenarios are as follows:

- **Present the technique:** Demonstrate that the rapid prototyping technique that we developed can be used to build working prototypes that present an inconsistency to a user so that the user can better understand the system.
- **Demonstrate rapidity:** Show that by applying the rapid prototyping technique, users can gain insight into inconsistencies more quickly than with a classic approach to software development.

4.2. Rapid prototyping technique

This section describes the technique used in all three scenarios.

To determine the effectiveness of the method in Section 3.2, a detailed technique is defined and used to perform the demonstrations. The technique is explained in five steps and is developed from a research perspective in which there are only fictional actors and use cases so that the determination can be performed as a desk study. Testing the technique in a real business setting will be interesting when the technique shows desirable outcomes in this study.

Phase 1: Preparing the prototypes

1. **Preparing the CNL input code:** Business rules and vocabulary are prepared based on a use case that will be used for conversion to the technical code. In this case, we prepare SVBR rules, but similar frameworks could also work.
2. **Conversion from CNL to technical code:** By using a conversion tool, the business rules and vocabulary are converted to technical code which can be used to create an application. Multiple conversion tools exist, but we use Karpovic's converter tool (Karpovic, Krisciuniene, Ablonskis, & Nemuraite, 2014) to generate OWL code.
3. **Importing the prototype:** Using a business rule system, we import the code from the converter to generate an application. This concept would work with other business rule systems. We use Protégé because a data model and rules can be implemented rapidly, and we found it easy to use.

Phase 2: Presenting the prototype

4. **Applying the scenario:** The part of the code that causes the inconsistency is held back in the previous step and added in this step to demonstrate the transition from a working system to an inconsistent one.
5. **Validate inconsistency:** We evaluate the inconsistency against the reasons described by Kalyanpur et al. (Kalyanpur et al., 2005).

Discussion

Discussing the results: Chapter 5 summarizes the results and provides arguments for using the technique to demonstrate inconsistencies in business rule systems.

4.3. Scenario 1: Instance level - cardinality constraint

4.3.1. Preparing the CNL input code

For this scenario the use case is based on “Introduce a new EU-Rent customer/driver” (Frias et al., 2003). In this instance, it is obligatory for a driver to have a valid driver’s licence. The following rule is thus declared:

R1: Each driver of a rental must have exactly one valid driver’s license

The chosen business rule is described in the SBVR specifications shown in the tables below; this is based on the specifications of the OMG (OMG, 2019). The colour coding is also based on the specification and has the following meaning:

- Green: The object(s)
- Blue: The relationship
- Orange: Cardinality and miscellaneous

Concept Type:	Object type
Definition	Driver who is responsible for a rental
Necessity:	The driver who *is responsible for* one rental
Interpretation:	One or more drivers are responsible for one rental.

Table 1: Object specification of driver

Concept Type:	Object type
Definition	Rental car which can be linked to drivers with a rental agreement
Necessity:	Each rental *is the responsibility of* one or more drivers.
Interpretation:	

Table 2: Object specification of rental

To implement the business rule, a user story is defined to simulate the actions of an employee. The following user story is created based on Frias’s use case (Frias et al., 2003):

Driver registration user story*: An employee is using the system to generate a rental contract. The employee must enter the driver’s details on a form as a step in this process. The employee tries to submit the form, and an error is returned stating that a driver’s license already exists for the driver.*

In this user story, the employee uploads the driver’s licence, but one already exists. This causes an inconsistency in the system, and the system should provide the employee with an error message that explains the reason for the inconsistency.

The model displayed below is taken from Frias’s specifications (Frias et al., 2003). We determine whether the system that will be the outcome of this rule is true to the data model to ensure the validity of the demonstration. An important distinction in the model is that the EU-Rent customer has exactly one driver’s license.

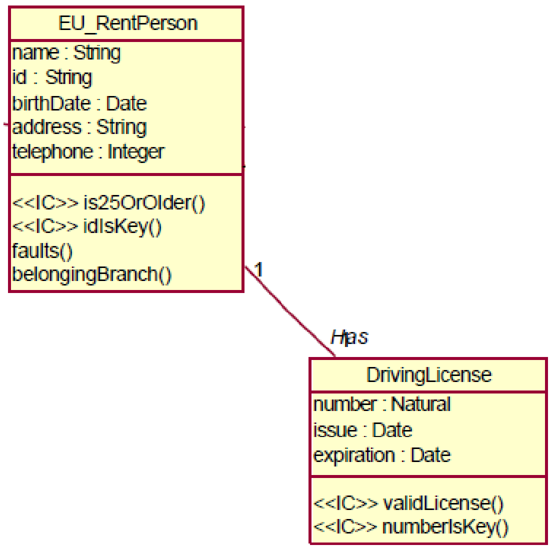


Figure 3. Data model for the "Driver's License" user story (Frias et al., 2003)

The model is populated with the data listed in the table below. The names are fictional and randomly chosen for this paper.

Customer	**Driver's license number**
John	DV_247612
Mary	DV_151231
William	DV_161323

Table 3. Customers in the database

4.3.2. Conversion from CNL to technical code

We convert the following SVBR rule into OWL to make the rule machine processable:

R1: Each driver of a rental must have exactly one valid driver's license.

To facilitate the conversion, we make use of the tool s2o (Karpovic et al., 2014). To make the tool work, we must alter the rule to follow the SBVR dialect used by Karpovic. To enforce that the system views the customers as unique, we add two other rules:

Input rules

It is necessary that drivinglicense is_attached_to exactly 1 driver; It is impossible that John is Mary;

Table 1. Rules for the s2o conversion tool

To use the generated code, we also need to add a business vocabulary. This facilitates the creation of classes, individuals and their associations. We created the following vocabulary:

Input business vocabulary

```
drivinglicense
driver
drivinglicense is_attached_to driver
John
        General_concept: driver
William
        General_concept: driver
Mary
        General_concept: driver
DV_247612
        General_concept: drivinglicense
DV_151231
        General_concept: drivinglicense
DV_161323
        General_concept: drivinglicense
DV_247612 is_attached_to John
DV_161323 is_attached_to William
DV_151231 is_attached_to Mary
```

Table 2. Business vocabulary for the s2o tool

The output of the conversion is OWL code in which “driver” and “drivinglicense” are sub-classes of “things” the sub-class “driver” possesses. The OWL code corresponds to the logic defined in the business rule by using restrictions and cardinality constructs. We split the conversion’s output code into several sections as presented in the tables below.

Output code by s2o (Karpovic et al., 2014)

The raw output code from the conversion has been manually edited to improve readability. Non-essential code has been removed and annotations placed to explain what is occurring.

```
Class declarations
Declaration( Class( <ns:s2o#drivinglicense> ) )
Declaration( Class( <ns:s2o#driver> ) )

Object property declaration
Declaration( ObjectProperty( <ns:s2o#is_attached_to__driver> ) )
ObjectPropertyDomain( <ns:s2o#is_attached_to__driver> <ns:s2o#drivinglicense> )
ObjectPropertyRange( <ns:s2o#is_attached_to__driver> <ns:s2o#driver> )

Object property assertions
ObjectPropertyAssertion( <ns:s2o#is_attached_to__driver> <ns:s2o#DV_247612>
<ns:s2o#John> )
ObjectPropertyAssertion( <ns:s2o#is_attached_to__driver> <ns:s2o#DV_161323>
<ns:s2o#William> )
ObjectPropertyAssertion( <ns:s2o#is_attached_to__driver> <ns:s2o#DV_151231>
<ns:s2o#Mary> )

Individual declarations
Declaration( NamedIndividual( <ns:s2o#John> ) )
Declaration( NamedIndividual( <ns:s2o#William> ) )
Declaration( NamedIndividual( <ns:s2o#Mary> ) )
Declaration( NamedIndividual( <ns:s2o#DV_247612> ) )
Declaration( NamedIndividual( <ns:s2o#DV_151231> ) )
Declaration( NamedIndividual( <ns:s2o#DV_161323> ) )

Class assertions, linking drivers to driving licenses
ClassAssertion( <ns:s2o#driver> <ns:s2o#John> )
ClassAssertion( <ns:s2o#driver> <ns:s2o#William> )
ClassAssertion( <ns:s2o#driver> <ns:s2o#Mary> )
ClassAssertion( <ns:s2o#drivinglicense> <ns:s2o#DV_247612> )
ClassAssertion( <ns:s2o#drivinglicense> <ns:s2o#DV_151231> )
ClassAssertion( <ns:s2o#drivinglicense> <ns:s2o#DV_161323> )

Cardinality constraint
SubClassOf( <ns:s2o#drivinglicense> ObjectExactCardinality( 1 )

Declaration of differing individuals
DifferentIndividuals( <ns:s2o#John> <ns:s2o#Mary> )
```

Table 3. Output code by s2o

4.3.3. Importing the prototype

We create a new project in Protégé and import the OWL code resulting from the conversion performed in the previous section. The OWL code from Table 3 represents the ontology and the OWL instances or individuals. It also contains the relation between the individuals and their restrictions. After importing the code from Table 3, a reasoner plugin is run. We use the HermiT reasoner plugin as it comes as a default with Protégé and has undergone several improvements on each version. We use version 1.4.3.456. As a result of running the reasoner, no inconsistency was detected. This is because the code from Table 3 specifies that the driver John has one driver's license, which does not violate any rule.

Upon inspecting the build, we can see that "driver" and "drivinglicense" are a sub-class of "things". Each sub-class has three constituents corresponding with the information from Table 3 in Section 4.3.1. One object property exists, "drivinglicense is attached to driver", which relates the two sub-classes to one another. Lastly, we see that the assertion "drivinglicense SubclassOf" 'drivinglicense'" is attached to exactly one driver. This indicates that a driver's license can only be related to a single driver.

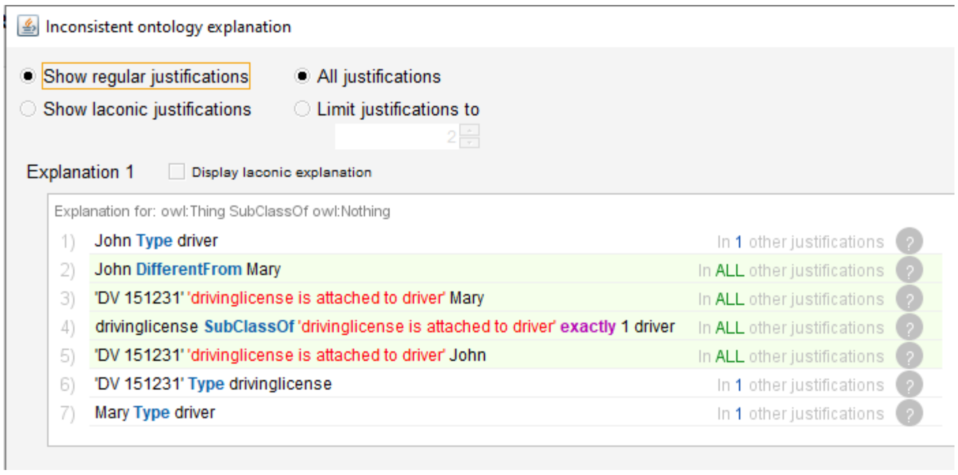
4.3.4. Applying the scenario

The prototype is now ready to be presented to the stakeholder to demonstrate the inconsistency. In its current state, the system is working correctly. Therefore, we add the following code marked with underlining to ensure that the inconsistency occurs.

```
DV_247612 is_attached_to John
DV_151231 is_attached_to John
DV_161323 is_attached_to William
DV_151231 is_attached_to Mary
```

Table 4. Modified input code to be imported to Protege

This code specifies that John has two drivers' licenses, violating the cardinality constraint that a driver can only have one driver's license. After the reasoner has been run, the system displays a pop-up window indicating that it has encountered an inconsistency. The following explanation is provided:


Inconsistent ontology explanation

● Show regular justifications ○ Show laconic justifications
● All justifications ○ Limit justifications to 2

Explanation 1 ☐ Display laconic explanation

Explanation for: owl:Thing SubClassOf owl:Nothing

```
1) John Type driver                                                              In 1 other justifications
2) John DifferentFrom Mary                                                       In ALL other justifications
3) 'DV 151231' 'drivinglicense is attached to driver' Mary                       In ALL other justifications
4) drivinglicense SubClassOf 'drivinglicense is attached to driver' exactly 1 driver  In ALL other justifications
5) 'DV 151231' 'drivinglicense is attached to driver' John                       In ALL other justifications
6) 'DV 151231' Type drivinglicense                                               In 1 other justifications
7) Mary Type driver                                                              In 1 other justifications
```

Figure 4. Explanation box for Scenario 2 presenting the reason for the inconsistency to the product owner

4.3.5. Validate inconsistency

We validate the inconsistency against different types of inconsistencies found in the literature to determine the validity of the outcome. The explanation box in Figure 4 indicates that the reason for the inconsistency is due to a cardinality violation. Kalyanpur et al. (Kalyanpur et al., 2005) describe a phenomena that relates to this: the individual assertion inconsistency. This happens on the instance level due to conflicting assertions about the instance or individual. The assertion that John has two drivers' licenses conflicts with the cardinality constraint because the instance (John) is linked to more than one driver's license. We can conclude that the inconsistency is triggered correctly, and the prototype was successful in demonstrating an inconsistency for a valid reason.

4.4. Scenario 2: Class level - unsatisfiable class

4.4.1. Preparing the CNL input code

The use case for this scenario is based on "Buy a car" (Frias et al., 2003), where a clerk tries to buy a sedan-type car that runs on diesel. However, that combination does not exist. The following rule is declared based on this:

R1: It is impossible that a sedan diesel is a car.

To implement the business rule, a user story is defined. This is used to simulate the actions of an EU-Rent employee. The following user story is created based on Frias's use case (Frias et al., 2003):

> **Driver registration user story**: *A clerk is trying to purchase a car to add to the car rental fleet. The clerk is using the system to determine the features of the car. During the type-selection process, the clerk encounters an error indicating that it is impossible to order a sedan that runs on diesel.*

In this user story, the clerk tries to select a type of car that does not exist. This causes an inconsistency in the system, and the system should provide the employee with an error message that explains the reason for the inconsistency.

The model is populated with the data shown in the table below:

Car fuel type	Car type
Diesel	Sedan
Diesel	Hatchback
Gasoline	Sedan
Gasoline	Hatchback

4.4.2. Conversion from CNL to technical code

We convert the following SVBR rule into OWL to make the rule machine processable:

R1: It is impossible that a sedan diesel is a car.

To facilitate the conversion, we make use of the tool s2o (Karpovic et al., 2014). To make the tool work, we must alter the rule to follow the SBVR dialect used by Karpovic.

Input rules

```
It is impossible that sedan_diesel is car
```

Table 5. Rule for s2o

To use the generated the code, we also need to add a business vocabulary to facilitate the creation of classes, individuals and their associations. Therefore, we created the following vocabulary:

Input business vocabulary

```
car
diesel_car
        General_concept: car
gasoline_car
        General_concept: car
sedan_gasoline
        General_concept: gasoline_car
hatchback_gasoline
        General_concept: gasoline_car
sedan_diesel
        General_concept: diesel_car
hatchback_diesel
        General_concept: diesel_car
```

Table 6. Business vocabulary for the s2o tool

The output of the conversion is OWL code in which "diesel_car" and "gasoline_car" are sub-classes of "car", and "car" is a sub-class of "things". Hatchback and sedan both exist twice as sub-classes below gasoline and diesel. The OWL code corresponds to the logic defined in the business rule by using a restriction construct.

Output code by s2o (Karpovic et al., 2014)

The raw output code from the conversion has been manually edited to improve readability. Non-essential code has been removed and annotations placed to explain what is occurring.

```
Class declarations
Declaration( Class( <ns:s2o#car> ) )
Declaration( Class( <ns:s2o#diesel_car> ) )
Declaration( Class( <ns:s2o#gasoline_car> ) )
Declaration( Class( <ns:s2o#sedan_gasoline> ) )
Declaration( Class( <ns:s2o#hatchback_gasoline> ) )
Declaration( Class( <ns:s2o#sedan_diesel> ) )
Declaration( Class( <ns:s2o#hatchback_diesel> ) )

Sub-class declarations
SubClassOf( <ns:s2o#diesel_car> <ns:s2o#car> )
SubClassOf( <ns:s2o#gasoline_car> <ns:s2o#car> )
SubClassOf( <ns:s2o#sedan_gasoline> <ns:s2o#gasoline_car> )
SubClassOf( <ns:s2o#hatchback_gasoline> <ns:s2o#gasoline_car> )
SubClassOf( <ns:s2o#sedan_diesel> <ns:s2o#diesel_car> )
SubClassOf( <ns:s2o#hatchback_diesel> <ns:s2o#diesel_car> )

Declaration of disjoint classes
DisjointClasses( <ns:s2o#sedan_diesel> <ns:s2o#car> )
```

Table 7. Output code by s2o for Scenario 2

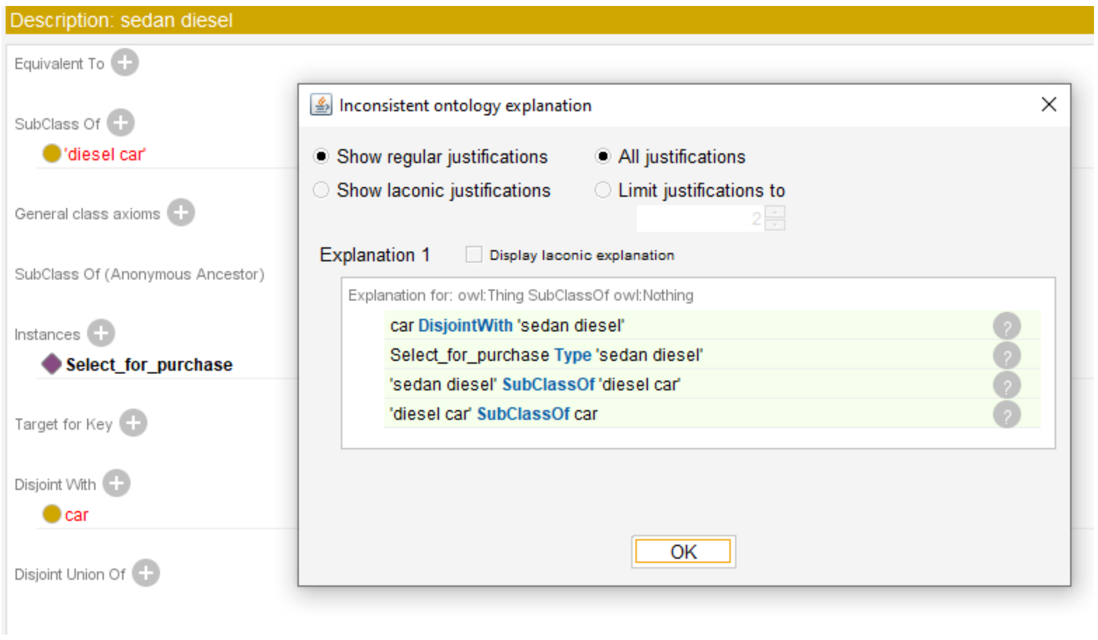
4.4.3. Importing the prototype

We create a new project in Protégé and import the OWL code resulting from the conversion performed in the previous section. The OWL code in Table 7 represents the ontology and the OWL instances or individuals. It also shows the relation between the individuals and their restrictions. After importing the code from Table 3, a reasoner plugin is run. We use the HermiT reasoner plugin as it comes as a default with Protégé and has undergone several improvements on each version. We use version 1.4.3.456. When running the reasoner, no inconsistency was detected. This was because no individuals were yet added to the system.

Upon inspecting the build, we can see that "car" is a sub-class of "things". Diesel_car and gasoline_car are both sub-classes of "car" and sub-classes themselves, representing sedan and hatchback categories. No individuals are added. In the sub-class "sedan_diesel", a "disjoint with" assertion is made with "car". This indicates that a sedan_diesel cannot be a car.

4.4.4. Applying the scenario

The prototype is now ready to be presented to the stakeholder to demonstrate the inconsistency. In its current state, the system is working correctly. Now, we manually enter an individual, which reflects an attempt to select a sedan_diesel car to continue with our purchasing process. The system displays a pop-up window after running the reasoner indicating that it has encountered an inconsistency. The following explanation is provided:


Description: sedan diesel

Equivalent To

SubClass Of
'diesel car'

General class axioms

SubClass Of (Anonymous Ancestor)

Instances
Select_for_purchase

Target for Key

Disjoint With
car

Disjoint Union Of

Inconsistent ontology explanation

● Show regular justifications ● All justifications
○ Show laconic justifications ○ Limit justifications to 2

Explanation 1 ☐ Display laconic explanation

Explanation for: owl:Thing SubClassOf owl:Nothing
car DisjointWith 'sedan diesel'
Select_for_purchase Type 'sedan diesel'
'sedan diesel' SubClassOf 'diesel car'
'diesel car' SubClassOf car

OK

Figure 5. Explanation box for Scenario 2 presenting the reason for the inconsistency to the product owner

4.4.5. Validate inconsistency

We validate the inconsistency against different types of inconsistencies found in the literature to determine the validity of the outcome. The explanation box in Figure 5 indicates “owl: Thing SubClassOf owl:Nothing”. This is an indication that the “sedan diesel” class is unsatisfiable, which is true because the explanation box also indicates that class “car” is disjoint with “sedan diesel”. Disjoint in OWL means that an individual which is a member of one class cannot also be an instance of the other specified class. Since “sedan diesel” is a sub-class of “car”, this means that a sedan diesel car cannot exist.

The reason for this inconsistency is due to an unsatisfiable class. Kalyanpur et al. (Kalyanpur et al., 2005) describe this as a contradiction in the ontology, implying that no instances or individuals can be part of that class but that one has been asserted nonetheless. This is an occurrence that happens on instance.

4.5.Scenario 3: Class level - disjoint class

4.5.1. Preparing the CNL input code

For this scenario, the use case is based on “Introduce a new EU-Rent customer/driver” (Frias et al., 2003), where an employee enters the information about a new customer. The following rule is declared:

R1: A customer cannot be European and non-European at the same time

To implement the business rule, a user story is defined. This is used to simulate the actions of an EU-Rent employee. The following user story is created based on Frias’s use case (Frias et al., 2003):

> **Driver registration user story**: *An employee is using the system to generate a rental contract. He must enter the driver’s details as a step in this process. The driver’s details must be entered on a form. The employee tries to submit the form, and the error is returned that the customer cannot be both from the EU and not from the EU.*

In this user story, the employee enters the customer’s information but accidently enters that the customer is from the EU and not from the EU. This causes an inconsistency in the system, and the system should provide the employee with an error message that explains the reason for the inconsistency.

The model is populated with the data listed in the table below. The names are fictional and randomly chosen for this paper.

Customer	Country	EU
John	Germany	European
Mary	France	European
William	Mexico	Non-European

Table 8. Customers in the database

4.5.2. Conversion from CNL to technical code

We convert the following SVBR rule into OWL to make the rule machine processable:

R1: A customer cannot be European and non-European at the same time

To facilitate the conversion, we make use of the tool s2o (Karpovic et al., 2014). To make the tool work, we must alter the rule to follow the SBVR dialect used by Karpovic.

Input rule

```
It is impossible that europe is non-europe
```

Table 9. Rule for the s2o conversion tool

To use the generated the code, we also need to add a business vocabulary to facilitate the creation of classes, individuals and their associations. Therefore, we created the following vocabulary:

Input business vocabulary

```
customer
country

customer lives_in country
customer isfrom europe
customer isfrom non-europe

europe
        General_concept: country
non-europe
        General_concept: country
france
        General_concept: country
germany
        General_concept: country
mexico
        General_concept: country
John
        General_concept: customer
Mary
        General_concept: customer
William
        General_concept: customer

John lives_in germany
Mary lives_in france
William lives_in mexico
John isfrom europe
Mary isfrom europe
William isfrom non-europe
```
Table 10. Business vocabulary for the s2o tool

The output of the conversion is OWL code in which “driver” and “Drivinglicense ” are sub-classes of “things” the sub-class “Driver” possesses. The OWL code corresponds to the logic defined in the business rule by using restrictions and cardinality constructs. We split the conversion’s output code into several sections as presented in the tables below.

Output code by s2o (Karpovic et al., 2014)

The raw output code from the conversion has been manually edited to improve readability. Non-essential code has been removed and annotations placed to explain what is occurring.

```
Class declarations
Declaration( Class( <ns:s2o#customer> ) )
Declaration( Class( <ns:s2o#country> ) )
Declaration( Class( <ns:s2o#non-europe> ) )
Declaration( Class( <ns:s2o#germany> ) )
Declaration( Class( <ns:s2o#mexico> ) )

Object property declarations
Declaration( ObjectProperty( <ns:s2o#lives_in__country> ) )
ObjectPropertyDomain( <ns:s2o#lives_in__country> <ns:s2o#customer> )
ObjectPropertyRange( <ns:s2o#lives_in__country> <ns:s2o#country> )
Declaration( ObjectProperty( <ns:s2o#isfrom__europe> ) )
ObjectPropertyDomain( <ns:s2o#isfrom__europe> <ns:s2o#customer> )
ObjectPropertyRange( <ns:s2o#isfrom__europe> <ns:s2o#europe> )
```

```
Declaration( ObjectProperty( <ns:s2o#isfrom__non-europe> ) )
ObjectPropertyDomain( <ns:s2o#isfrom__non-europe> <ns:s2o#customer> )
ObjectPropertyRange( <ns:s2o#isfrom__non-europe> <ns:s2o#non-europe> )

Object property assertions, linking customers to countries and continent
ObjectPropertyAssertion( <ns:s2o#lives_in__country> <ns:s2o#John> <ns:s2o#John> )
ObjectPropertyAssertion( <ns:s2o#lives_in__country> <ns:s2o#Mary> <ns:s2o#Mary> )
ObjectPropertyAssertion( <ns:s2o#lives_in__country> <ns:s2o#William> <ns:s2o#William> )
ObjectPropertyAssertion( <ns:s2o#isfrom__europe> <ns:s2o#John> <ns:s2o#John> )
ObjectPropertyAssertion( <ns:s2o#isfrom__europe> <ns:s2o#Mary> <ns:s2o#Mary> )
ObjectPropertyAssertion( <ns:s2o#isfrom__non-europe> <ns:s2o#William>
<ns:s2o#William> )

Subclass declaration
SubClassOf( <ns:s2o#europe> <ns:s2o#country> )
SubClassOf( <ns:s2o#non-europe> <ns:s2o#country> )
SubClassOf( <ns:s2o#france> <ns:s2o#country> )
SubClassOf( <ns:s2o#germany> <ns:s2o#country> )
SubClassOf( <ns:s2o#mexico> <ns:s2o#country> )

Individual declarations
Declaration( NamedIndividual( <ns:s2o#John> ) )
Declaration( NamedIndividual( <ns:s2o#Mary> ) )
Declaration( NamedIndividual( <ns:s2o#William> ) )

Class assertions, linking individuals to class customer
ClassAssertion( <ns:s2o#customer> <ns:s2o#John> )
ClassAssertion( <ns:s2o#customer> <ns:s2o#Mary> )
ClassAssertion( <ns:s2o#customer> <ns:s2o#William> )

Declaration of disjoint classes
DisjointClasses( <ns:s2o#europe> <ns:s2o#non-europe> )
)
```
Table 11. Output code by s2o

4.5.3. Importing the prototype

We create a new project in Protégé and import the OWL code resulting from the conversion performed in the previous section. The OWL code from Table 11 represents the ontology and the OWL instances or individuals. It also contains the relation between the individuals and their restrictions. After importing the code from Table 3, a reasoner plugin is run. We use the HermiT reasoner plugin as it comes as default with Protégé and has undergone several improvements on each version. We use version 1.4.3.456. As a result of running the reasoner, no inconsistency was detected. This is because the code from Table 11 specifies that the customer William is only European, which does not violate any rule.

It can be seen that "country" and "customer" are sub-classes of "things". Country has five sub-classes: Europe, non-Europe, France, Germany and Mexico. Three individuals exist: John, Mary and William. These are instances of customers, their respective countries and whether they are European. This corresponds with the information from Table 8 in Section 4.5.1. Three object properties exist: "customer lives_in country", "customer isfrom europe" and "customer isfrom non-europe”. This relates the sub-classes to one another. Lastly, the assertion “drivinglicense SubclassOf” ‘drivinglicense’” is attached to exactly one driver.

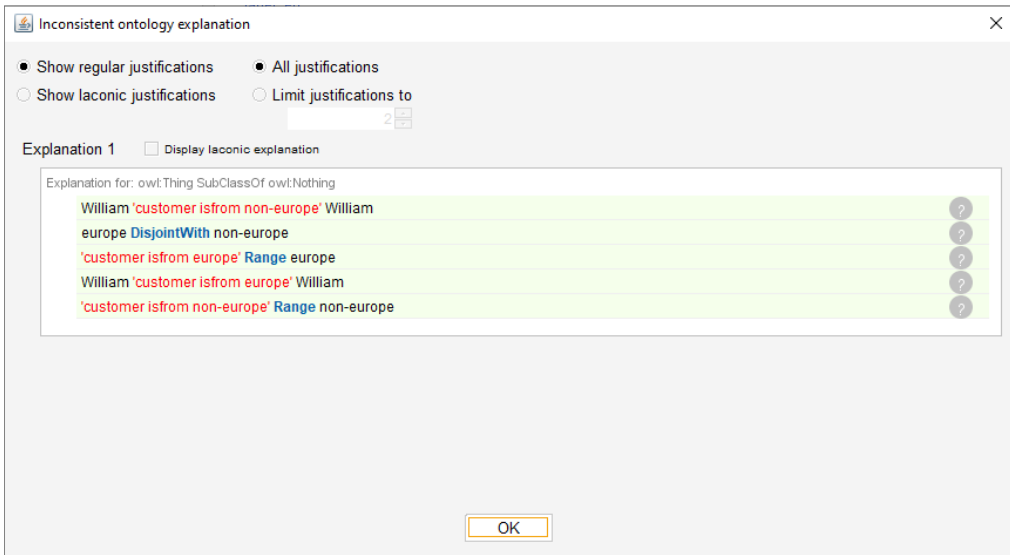
4.5.4. Applying the scenario

The prototype is now ready to be presented to the stakeholder to demonstrate the inconsistency. In its current state, the system is working correctly. Therefore, we add the following code marked with underlining to ensure that the inconsistency occurs.

```
John lives_in germany
Mary lives_in france
William lives_in mexico
John isfrom europe
Mary isfrom europe
William isfrom non-europe
William isfrom europe
```

Table 12. Modified input code to be imported to Protege

This code specifies that William is both from Europe and is not from Europe, violating the rule that a customer cannot belong to both classes. The system displays a pop-up window after running the reasoner indicating that it has encountered an inconsistency. The following explanation is provided:


Inconsistent ontology explanation
● Show regular justifications
○ Show laconic justifications
● All justifications
○ Limit justifications to
2
Explanation 1
Display laconic explanation
OK

Explanation for: owl:Thing SubClassOf owl:Nothing

```
William 'customer isfrom non-europe' William
europe DisjointWith non-europe
'customer isfrom europe' Range europe
William 'customer isfrom europe' William
'customer isfrom non-europe' Range non-europe
```

Figure 6. Explanation box for Scenario 3 presenting the reason for the inconsistency to the product owner

4.5.5. Validate inconsistency

We validate the inconsistency against different types of inconsistencies found in the literature to determine the validity of the outcome. The explanation box in Figure 6 indicates “owl: Thing SubClassOf owl:Nothing”. This is an indicator that something is wrong with the classes. In the statements, William is both a “customer isfrom europe” and “customer isfrom noneurope”, with the statement “europe DisjointWith non-europe” in between. Disjoint in OWL means that an individual that is a member of one class cannot also be an instance of the other specified class.

The reason for this inconsistency is due to a disjoint class. Kalyanpur et al. (Kalyanpur et al., 2005) describe this as a contradiction in the ontology, implying that no instances or individuals can be part of that class while also being part of the other class but that one has been asserted nonetheless.

5. Conclusion, discussion and recommendations

The aim of this section is to summarize the results of the demonstrations documented in Chapter 4 and discuss the outcome of applying the rapid prototyping technique for demonstrating inconsistencies. The main research question is answered based on the results of the demonstrations, and considerations for future research are discussed.

5.1. Conclusion

The research question to be answered in this study was, *"How and to what extent can rapid prototyping using Semantic Web software quickly demonstrate the application of inconsistencies to a stakeholder in a business rule system?".* The research began with the forming of a theoretical framework and the identification of what types of prototype and inconsistency causes can be used to evaluate the demonstration. We found three categories for inconsistency causes in the literature (Kalyanpur et al., 2005):

1. Instance-level causes
2. Class-level causes
3. Class-assertion causes

We used only the first two because we could not base a scenario on the third. The limiting factor for this was the tool Protégé, which we used for the scenarios. This tool would not allow data entry which would result in a class being mistaken for an individual.

In the following phase, a method was developed to demonstrate inconsistencies. To evaluate the rapidity of the method in demonstrating inconsistencies, two use cases were described from the fictional company EU-Rent:

1. Introducing a new EU-Rent customer/driver
2. Buying a car

The use cases were derived from the work of Frias et al. (Frias et al., 2003) to create a setting in which a system could be developed to support a business process. We created SVBR code in preparation for using Karpovic's tool s2o (Karpovic et al., 2014) to convert our easy-to-understand code into technical OWL code which could be imported into an application like Protégé to create a system. After importing the OWL code into Protégé, the inconsistency explanation boxes were reviewed and compared with inconsistency causes found in the literature.

The foremost conclusion drawn was that using a code-conversion tool is efficient. We experienced it as being quick, and it can be especially helpful for a team in which not everyone is trained in the OWL syntax: Part of the team can work on defining the business rules and concepts instead of directly writing OWL code. During this research, we spent hours determining the correct syntax to make assertions for our ontology in OWL code. Using the SVBR-to-OWL convertor from Karpovic et al., we created the same ontology in half the time. This truly made the development process faster. Another important aspect contributing to the rapidity of the process is the scope of the use case for which an inconsistency is to be demonstrated. In the beginning, we chose a far larger use case than we eventually settled on. Therefore, we had to build a less complex system that could still demonstrate the inconsistency that we intended to show. Choosing the right scope proved to be an important part of the rapid prototyping technique.

The next conclusion is that Protégé is a good tool when determining how to employ rapid prototyping to demonstrate inconsistencies in the Semantic Web domain. Primarily, the feature for importing OWL code and the user interface for overviewing the ontology make Protégé fit for this purpose. It is possible to configure systems in such ways that all possible inconsistencies and their resulting behaviours can be demonstrated.

As there is no previous research on the combination of rapid prototyping and demonstrating inconsistencies in the Semantic Web domain, there is no existing literature with which comparisons can be made. However, we did show that throwaway rapid prototypes as researched and defined by (Forward et al., 2012) can be used with Semantic Web software to demonstrate inconsistencies quickly. Additionally, the causes for the inconsistencies generated are identified by (Kalyanpur et al., 2005).

5.2. Discussion

In performing this study, we made choices along the way that influenced the outcome. There are several decisions that could have changed the results or provided different interpretations.

EU-Rent

All use cases were derived from the fictional company EU-Rent, and the results were all collected from desk research. Given the nature of the science, it would be interesting to evaluate the rapid prototyping technique in a real business setting. This could provide different results.

Conversion tools

We used the s2o conversion tool by Karpovic et al. in this study. However, this tool is still in the prototype phase and has not undergone rigorous testing or been used in a production environment in an organization.

Even with these limitations, we experienced that the use of this conversion tool had benefits in the following aspects:

1. **Writing code rapidly**. The tool automatically creates annotations when an entity or concept is declared. When editing OWL code using an editor, all these annotations would have to be entered manually.
2. **Business stakeholders can now write code**. With limited knowledge of OWL and the syntax required to make declarations, a user can still create and edit a complex system. This provides opportunities such as allowing a person with knowledge of the process a system must facilitate to directly edit the business rules that generate code.

It is important to note that these benefits were experienced during this study regardless of the state of the conversion tool. This means that more mature conversion tools could yield even better results.

Protégé

This study was mainly in the context of the Semantic Web. There are not many tools available in this domain, so it was quickly decided that Protégé was the best choice for demonstrating inconsistencies. However, if a study were to reach more broadly than the Semantic Web, other tools might also be interesting. Using different software to demonstrate inconsistencies could have changed the outcome of this study.

Inconsistency causes

While there are only a limited number of causes for inconsistencies discussed in the literature (Kalyanpur et al., 2005), it can still be difficult to demonstrate an inconsistency. As a system becomes more complex, it is increasingly difficult to predict how different components will influence an inconsistency. We limited this study to relatively simple systems. When applying the rapid prototyping technique to more complex systems, different outcomes may occur in terms of rapidity and effectiveness in demonstrating the inconsistency.

5.3. Recommendations for practice

The use of a rapid prototyping technique is highly recommended for demonstrating inconsistencies to stakeholders, not merely in the Semantic Web context. For organizations that struggle to deliver IT artefacts on time, rapid prototyping can hold the key for improvement, especially using a conversion tool like s2o from Karpovic et al. (Karpovic et al., 2014) which converts business rules to technical code. This allows the business stakeholders and IT professionals to work more closely together since there is no technical knowledge required to create and edit business rules; this will have a direct impact on the architecture of the resulting system. In this way, business stakeholders can focus on introducing elements that provide the most value for the business, whilst the IT professionals maintain the overall integrity of the system.

5.4. Recommendations for future research

This study had a limited scope. We only set out to demonstrate inconsistencies with our rapid prototyping technique. As we were searching for methods to make our technique even more rapid, we came across many options that we did not have time to explore. Other conversion tools might exist, and a Jenkins automation server can be used to automatically test and record results.

Another topic for further research is to determine how the rapid prototyping technique can be used to demonstrate and evaluate explanations to users after an inconsistency is found. This would build further upon the research conducted here.

5.5. Reflection

Design science (Hevner, 2007) is the research method for this study. In hindsight, this was a good choice as some core design science principles overlap with rapid prototyping. The main aspect of design science that resonated here was the idea of being able to adapt and improve after the first demonstration when conducting multiple demonstrations in research. By setting up criteria and finding existing literature to compare with our findings, we effectively applied design science principles to answer the research question.

References

Bahe, A. A. (2021). *Design patterns for explaining inconsistencies in business systems.* (Business Process Management & IT Master's program). Open University of the Netherlands, Utrecht.

Berners-Lee, H., Lassila. (2001). The Symantic Web. *Scientific American*.

Boyer j., M. H. (2011). Agile Business Rule Development. In *Agile Business Rule Development. In: Agile Business Rule Development*.

Bradley Camburn, V. V., Julie Linsey, David Anderson,, & Daniel Jensen, R. C., Kevin Otto and KristinWood. (2017). Design prototyping methods: state of the art in strategies, techniques, and guidelines. *Cambride University Press, Des. Sci., vol. 3, e13 journals.cambridge.org/dsj DOI: 10.1017/dsj.2017.10*.

Chittimalli, P. K., & Anand, K. (2016). *Domain-independent method of detecting inconsistencies in SBVR-based business rules*. Paper presented at the Proceedings of the International Workshop on Formal Methods for Analysis of Business Systems, Singapore, Singapore. https://doi.org/10.1145/2975941.2975943

Feuto Njonko, P. B., Cardey, S., Greenfield, P., & El Abed, W. (2014, 2014//). *RuleCNL: A Controlled Natural Language for Business Rule Specifications.* Paper presented at the Controlled Natural Language, Cham.

Forward, A., Badreddin, O., Lethbridge, T., & Solano, J. (2012). Model-driven rapid prototyping with Umple. *Software: Practice and Experience, 42*. doi:10.1002/spe.1155

Frias, L., Calafat, A. Q., & Ramon, A. O. (2003). *EU-Rent car rentals specification*.

Haase, V. H., Huang, Stuckenschmidt, Sure. (2005). A framework for handling inconsistency in changing ontologies.

Hallgrimsson, B. (2012). *Prototyping and Modelmaking for Product Design*: Laurence King Publishing.

Hevner, A. R. (2007). A three cycle view of design science research. *Scandinavian journal of information systems*.

Kalyanpur, A., Parsia, B., Sirin, E., & Hendler, J. (2005). Debugging unsatisfiable classes in OWL ontologies. *Journal of Web Semantics, 3*(4), 268-293. doi:https://doi.org/10.1016/j.websem.2005.09.005

Karpovic, J., Krisciuniene, G., Ablonskis, L., & Nemuraite, L. (2014). The Comprehensive Mapping of Semantics of Business Vocabulary and Business Rules (SBVR) to OWL 2 Ontologies. *Inf. Technol. Control., 43*, 289-302.

Karpovic, J., & Nemuraite, L. (2011). Transforming SBVR business semantics into Web ontology language OWL2: main concepts. *Information Technologies*, 27-29.

Klyne, G., & Carroll, J. (2006). Resource Description Framework (RDF): Concepts and Abstract Syntax. *World Wide Web Consortium, 10*.

Leo R. Vijayasarathy, D. T. (2008). AGILE SOFTWARE DEVELOPMENT:A SURVEY OF EARLY ADOPTERS *Journal of Information Technology Management, ISSN #1042-1319*

McGuinnes, D. S. (2004). Explaining answers from the semantic web: The inference web approach. *Journal of Web Semantics, 1(4)*.

McGuinness, D. L., & van Harmelen, F. (2004). Web Ontology Language. In *W3C Recommendation.*

Noy, N. F., Sintek, M., Decker, S., Crubezy, M., Fergerson, R. W., & Musen, M. A. (2001). Creating Semantic Web contents with Protege-2000. *IEEE Intelligent Systems, 16*(2), 60-71. doi:10.1109/5254.920601

OMG. (2017). *Semantics of Business Vocabulary and Business Rules*. Retrieved from http://www.omg.org/spec/SBVR/1.4/PDF

OMG. (2019). *Semantics Of Business Vocabulary And Business RulesSBVR*. Retrieved from https://www.omg.org/spec/SBVR/1.5/PDF

S. Finkelstein, R. B., D. Jacobs. (2009). Principles for Inconsistency. *arXiv:0909.1782*.

Sebastian Herzig, A. Q., Christiaan Paredis. (2014). An approach to identifying inconsistencies in Model-Based Systems Engineering. *Procedia Computer Science, 28*.