Validation and Recommendation Engine from Service Architecture and Ontology

Daniel Mercier[1][a] and Anthony Ruto[2]
[1]*Autodesk Research, 661 University Ave, Toronto, ON, Canada*
[2]*Autodesk Research, 17 Broadwick St., Soho, London, U.K.*

Keywords: Validation, Recommendation, Service Architecture, Ontology, Object-oriented, Domain Engineering.

Abstract: The Cloud has emerged as a common platform for data convergence. Structured data, unstructured, serialized, or even chunks of data in various formats are now being transferred, processed, and exchanged by a multitude of services. New applications, service-oriented, rely heavily on the managing of these flows of data. The situation is such, that the perspective is changing, placing data at the center. In this context, end-users must rely on new derivative services to validate these flows of information; and expect from these services accurate feedback and some degree of intelligent recommendations. In this article, we introduce a new validation and recommendation engine encapsulated in a service, backed by ontology and a knowledge structure based on reusable components for fast integration and increased resilience.

1 INTRODUCTION

The transition from desktop applications to cloud-based applications is changing the way we approach software engineering. Applications which were historically designed to contain all the necessary functional requirements, from display to data handling and processing, now deployed on the cloud are being transformed into assemblies of functions encapsulated into easily deployable services sometimes called micro-services due to their small size. With the reduced load on the applications, access moved to lightweight or web-based front-end applications connected to a processing back-end. This transition from a monolithic architecture to a cloud back-end of interconnected services placed an emphasis on communications and drove the industry to transform its approach to data. We are reaching a point where data is now driving executions. This data integration (Hohpe and Woolf, 2004) requires strong data consistency and good control over flows between services (Kim et al., 2010). Naturally, validation is central to maintaining consistency. Validation is applied for security reasons such as in firewalls when the data crosses trust boundaries (Aljawarneh et al., 2010); or simply to check whether the data fulfills the requirements placed by a target application. In this ever-changing and highly distributed environment, applications built from service-oriented architecture (SOA) relies on new derivative services or agents to account for all non-core functions. Data validation naturally falls under this category. During our study of the research field, we identified that while in the last few years, there are consistent efforts towards validating service architecture and service workflow (Faiez et al., 2017), we also found a somewhat limited presence in the segment of generic content validation. This observation became more noticeable when looking for combination of semantic web technologies to the validation of data.

1.1 Validation

This work focuses on the architecture of a service for the validation of data. Validation for statistical purposes as stated by the European Union (ESS handbook, 2018) differentiates validation methods by the chosen principals that are used to separate validation rules. It introduces validation levels as derived from a business perspective pertaining to a domain where each validation level is verified by a set of rules applied to the given data. The present work is structured to embody these considerations.

[a] https://orcid.org/0000-0003-1239-9728

Mercier, D. and Ruto, A.
Validation and Recommendation Engine from Service Architecture and Ontology.
DOI: 10.5220/0008070602660273
In *Proceedings of the 11th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management (IC3K 2019)*, pages 266-273
ISBN: 978-989-758-382-7

1.2 Environment

The proposed validation service is typically located between a front-end application and a computing Cloud back-end composed from an aggregation of independent and potentially interconnected services. The primary purpose of the service is to verify the data sent by the front-end to the back-end to prevent failed back-end executions that could be costly to the end-user. For example, simulation solvers and graphic rendering engines require complex back-end processing that can run for hours, days or even months. An objective of the service is also to provide a fluid experience. The service does so by running validation and verification over the received data while it is being generated. For this reason, validation as expressed in this work embodies the two concepts of validation and verification. The service response is a validation report and a set of recommendations to improve the quality of the data towards more stable and optimized back-end executions. The service core knowledge structure is a combination of domain engineering and Semantic Web technologies.

1.3 Domain Engineering

Domain engineering was introduced in software engineering to promote the identification of domains for the reuse of software assets. The necessity for a concept of domains is essential for validation in a service-oriented architecture (Zdun and Dustdar, 2006) where each service has a dedicated purpose and set of content required to operate. As an example, a service may transform computer aided designs into meshes, another may process material data from their raw form to their process specific equivalent. Each of these services acts as an independent and reusable domain with independent validations. Feature modeling, a domain engineering technique (Kim, 2006), adds an additional layer of granularity by introducing the concept of domain feature. When properly implemented, the combination of domain engineering and feature modeling lowers content creation time by leveraging commonalities and increases resilience through the reuse of domain and feature assets. This work uses domain engineering for segmenting and structuring the validation knowledge.

1.4 Ontology and Semantic Web

This work uses ontology for the storage of the validation knowledge. An ontology is a well-structured and rigorous organization of knowledge with tree of classes, class inheritance, and class relationships. It is a natural construct to store domains and domain features. The technology was historically tied to the Semantic Web, an extension of the World Wide Web, and equally designed to allow data to be shared and reused across application, enterprise, and community boundaries. Several approaches proved the value of design patterns to generate reusable domain ontologies (Musen, 2004) and the implementation of these domain ontologies to determine problem-solving strategies (Gil and Melz, 1996). When combined with model-driven design (Strmečki et al., 2016) or feature modeling (Wang et al., 2007), the ontology adds the benefits of formal semantic and reasoning capabilities over its structure. The integration of an ontology in software development is sometimes seen as a challenge (Baset and Stoffel, 2018). Despite the initial low adoption, the technology gained traction in specific industries such as in bio-medical and benefited from the recent interest in artificial intelligence and machine learning. Semantic Web technologies is also often used for the validation of system models (Shanks et al., 2003; Kezadri and Pantel, 2010) and workflow (Khouri and Digiampietri, 2018).

1.5 Objectives

The merit in this work's approach lies in the combining of technologies. The validation knowledge is structured using domain engineering to facilitate the reuse of domain features, while the storage is done through ontology. The proposed validation service offers from the combining of ontology and Cloud computing, a wide range of validation mechanisms.

The service is expected to:

- Validate the data received from the front-end application either in a partial or complete form,
- Generate the necessary information to guide the user to find and correct invalid entries,
- Provide recommendations on how to improve the data,
- Form predetermined flow of validations using dependencies between domain features,

Section 2 covers the composition of the knowledge structure and the validation mechanisms. Section 3 addresses the various aspects of the service workflow. Section 4 covers an overview of related works. Finally, in section 5, we discuss future explorations before the conclusions.

2 ARCHITECTURE

The proposed service is designed as a stateless service for cloud deployment. Its knowledge base is stored remotely and synchronized locally to allow scalability. It has a user interface for knowledge acquisition, user-access control to segment operations, and exposes a REST interface for communications with the front-end application.

2.1 User Roles

The service has four user roles. The first role is the *user*. It refers to the front-end application that sends data within a given scope and receives in return a validation report with recommendations. The second role is the *subject-matter expert* who is in charge of creating the material required to validate the data. The third role is the administrator who controls the deployment and stable operations of the application. The administrator has extended access to the knowledge base and service activities. The administrator manages the user base and the deployment of service plugins. The last role is the *execution-engine expert* who creates for the service, the necessary plugin(s) to compose and execute validation rules from code logic.

2.2 Communications and Data Format

The proposed service has a Representational State Transfer(REST) interface (Fielding, 2000). A typical request to a REST interface is formulated with a payload formatted in either HTML, XML, JSON, or some other format, and elicits a response with a similarly formatted payload. The service implements the JavaScript Object Notation or JSON, a lightweight semi-structured data-interchange format. The format is popular in system communications, in the configuration of systems and applications, as well as for the storage of structured objects. The format offers a conveniently short list of data types for single values with *null*, *boolean*, *number*, and *string*; and two types of complex constructs with *arrays*, and *objects*. The JSON format has an associated schema format to describe and validate the structure of JSON documents. For example, the service uses JSON schema to validate the types of the received data. The service also uses JSON schema during knowledge acquisition to map the expected JSON values to feature attributes.

2.3 Knowledge Base

The structure of the knowledge base is a combination of two constructs. One for domains to contain the reusable assets, and one to contain application specifics and connect to the reusable assets.

2.3.1 Domains

Domains are stored as independent ontologies with unique namespaces. The domain features are represented by ontology classes. As an example of domain segmentation, a structural simulation solver requires a geometry, one of its domains. The geometry has a mesh composed of nodes and elements. While the geometry is the domain, the mesh, nodes and elements can be considered as domain features. As feature classes, they each have a series of individual attributes. For example, axial coordinates are attributes of a node. From the feature attributes, the subject-matter expert can create the validation rules. An example of simple validation rule for a node could be the checking of whether its coordinates are beyond a particular location. Another part of knowledge acquisition is the mapping of feature attributes to the specific JSON values expected in the data sent by the front-end application as illustrated by fig 1. This mapping process is manual. Once the knowledge acquisition is complete and during validation, the mapping between schema and attributes is used to inject the received data into instances of feature classes and the validation rules are then executed for each feature class over the instance of that class.

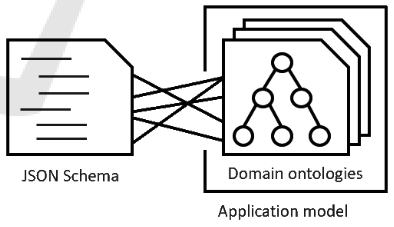

Figure 1: JSON and knowledge structure.

2.3.2 Application Model

The second type of construct is the application model. It is also stored in an independent ontology but contains all the necessary validation material to validate the data coming from a specific application. The application model combines reusable domain assets and application specific features. The application model contains two distinct types of feature classes in additional to the domain features. The first type is the primary application class that serves as a root and connects all other features. The primary application class is automatically created upon the creation of an application model. The second type is the *list* class to represent JSON *arrays*. The presence of arrays inside the JSON content influences the matching between JSON values and feature attributes. If a feature attribute is matched to a value inside an array, another attribute from the same feature class can only be matched to a value inside the same array or a value underneath that array in the schema tree. For this reason, every time an attribute is selected for a feature class from within an array, a *list* class is automatically associated as a parent to the feature class. This presence of *list* class has the advantage of grouping feature classes within the application model.

2.3.3 Feature Dependencies

The knowledge structure allows dependency between feature classes towards a sequential activation. This system of dependencies creates a natural flow of feature validations. Features may be dependent between each other within the same domain such as a mesh is dependent on its nodes and elements, but dependencies may also go beyond domains, in which case the information is stored inside the application model. There are two methods to create dependencies. One through the *isDependentOn* relationship, the other is by using data as a trigger on feature attributes. Because the activation of child classes may be conditional on values generated by parent classes, the service offers the possible enrichment of the original data with newly processed values. By doing so, the available data naturally grows with the flow of validations.

2.4 Validation Mechanisms

An essential part of our approach is the combination of two complementary types of validation mechanisms:

- Descriptive Logic(DL). A DL rule is embedded as class axiom directly inside an ontology and complement the ontology tree of classes and relationships to determine its coherence and consistence using a reasoner.
- Code Logic(CL). A CL rule is a script, or a piece of code written in a procedural language. The language provides the necessary functions to process class attributes and return an outcome to the validation.

Each technique approaches validation differently. While the Descriptive Logic follows an Open-World Assumption (OWA) where lack of knowledge of a fact does not immediately imply knowledge of the negation of a fact, Code Logic follows a Closed-World Assumption (CWA) where every attribute is assumed to be known or the rule cannot be validated. As each validation rule act independently, the combination of the two approaches constitutes a powerful and unique validation ecosystem.

2.4.1 Descriptive Logic

Descriptive logic is a family of knowledge representation languages for authoring ontologies. An efficient and scalable reasoners such as the Pellet or the HermiT (Motik et al., 2009) reasoners for the Ontology Web Language(OWL) (W3C, 2004) are natural complements to this formal framework to infer logical coherence and content consistency. The service offers two layers of descriptive logic. The first layer is at the domain-level and the second at the application-level. If the DL rule is bound to the domain namespace, the rule is embedded within the associated domain. If the DL rule connects multiple namespaces, the rule is embedded instead within the application model. One challenge with Descriptive Logic stands in the richness of the ontology. We observed that given freedom over the creation of relationships. The taxonomy can quickly grow and lead to the emergence of duplicates with distinct semantic names. The presence of these duplicates both dilutes the capability for feature reuse and generates confusion during content creation. For this reason, we decided to restrict the list of common relationships. This approach made rule creation from descriptive logic much simpler.

2.4.2 Code Logic

Validation rules created from Code Logic are small fragments of procedural code that consume feature attributes as inputs and return the validation outcome as output. The implementation of Code Logic inside the service provides a versatile environment to transform attributes. The challenge with Code Logic is to offer a rich language and an extended library of processing functions. This is where the service takes full advantage of cloud computing and service-meshes to execute code. Fig 2 illustrates the final relationships between entities of the knowledge structure after the introduction of functions.

2.5 Execution-engine Plugin

When it comes to executing validation rules from code logic, the service uses a plugin model to expand its capabilities. The plugin(s) controls the edition and execution of validation rules from Code Logic. The plugin implementation is the responsibility of the

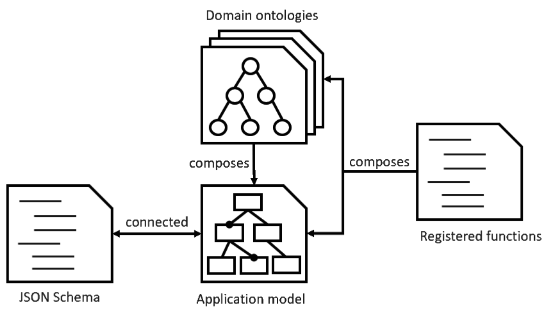

Figure 2: Overall knowledge structure.

execution-engine expert. The objective of the plugin(s) is to provide a controlled environment, flexibility over the procedural language, scope of available functions as well as the execution platform. Due to their extreme versatility, our approach focused on leveraging service-mesh managers.

2.5.1 Service-mesh

The growth of cloud technologies has seen a rise in efficient tools to support service-oriented architectures. Container technologies such as Docker™, or the lightweight headless technology such as AWS Lambda™, introduced a new and effective environment for fast development and encapsulation of small-sized services or micro-services. A whole ecosystem was quickly developed to complement these technologies with complex platforms for application deployment, scaling, and management of fleets of services. The most recent efforts are pushing towards the emergence of *service mesh* managers where a *service mesh* is a predefined workflow of service executions. These *service mesh* managers offer easily configurable, low latency, and high volume layers of network-based inter-processes based on application programming interfaces (APIs). They ensure coordinated, fast, reliable, and secure communications between services. Service-mesh managers consume existing platform technologies for service discovery, load balancing, encryption, observability, traceability, authentication and authorization, and the support of circuit breaker patterns.

2.5.2 Plugin Features

A standard plugin must expose a rich language with a set of logical, conditional, and loop operators along with a list of advanced processing functions. To be effective, the plugin must also be able to assist with the composition of procedural code and validate the generated code. For these reasons, plugin specifications were design to be simple but still allow *rule execution-engine experts* freedom over the plugin composition. A plugin must:

- Run on the same instance as the service,
- Use gRPC or Google Remote Procedural Call to communicate,
- Expose a common set of four gRPC interfaces,
- Implement in its procedural language, three standard functions.

The service consumes four common gRPC interface from the plugin. The first three functions are consumed during knowledge acquisition. The last one is consumed during validation.

- The first interface provides standard information about the plugin such as the name, version, date of creation, and description.
- The second interface returns the registry of advanced functions for data processing. The information serves as a reference on functions and is displayed in a dedicated window inside the user-interface during knowledge acquisition.
- The third interface validates the code during rule edition, acts as an auto-complete feature, and returns schema information describing the enrichment of the original application data.
- The last interface executes the Code Logic for the given attribute values. It executes the rule and returns the outcome. The response contains the outcome of the execution, recommendations, and the values to enrich the original application data.

The plugin specifications also include three function signatures, common to all plugins that should be made available during code edition:

- The first one is the *return* function. This function call marks the ending(s) of a rule execution. The function signature includes the returned state and the message associated with the return state, if any.
- The second one is the *insert* function. The computed values passed by this function will be added to the original application data upon successful validation of the feature class.
- The third one is the *log* function. This function signature has a message and a number to define its importance. The message is considered by the service as a recommendation and returned as such along with the location of generation.

Finally, the plugin specifications emphasize that the service only recognizes JSON types. When the service sends data to the plugin interfaces, the values

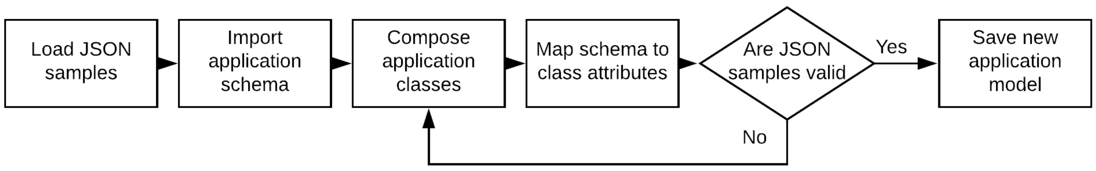
Load JSON samples
Import application schema
Compose application classes
Map schema to class attributes
Are JSON samples valid
Yes
No
Save new application model

Figure 3: Test-driven workflow.

are always sent in a JSON format and as such encapsulated as string. The plugin must manage types and type conversions.

3 SERVICE WORKFLOW

The proposed service validates from the knowledge structure:

- Application model by running a reasoner over its structure and embedded axioms,
- Feature attributes using standard properties (*),
- Feature rules composed from code logic.

(*) During the creation of feature class attributes, the subject-matter expert has the choice to activate standard properties, such as default values and simple checks. These properties are type dependent. For example, minimum and maximum are standard properties for numbers, regular expressions for strings, list of values for enumerations, and average and standard deviation for sets of values.

The recommendations cover:

- Identification of gaps within the list of feature class attributes preventing activation of validation,
- Whenever a default was used to fill a gap,
- Any message sent by rules composed from Code Logic.

3.1 Knowledge Acquisition Workflow

The knowledge acquisition workflow takes its inspiration from the concept of Test-Driven Design (TDD) in software engineering. The method relies on test cases to assist the subject-matter expert with the creation of validation knowledge. The test cases are loaded before knowledge acquisition in the central repository and associated to the target application model. These test cases serve as a baseline over the behavior of the front-end application and are used to validate an application validation knowledge. The TDD workflow as applied to this service is illustrated by fig 3.

3.2 Validation Cycle

The validation cycle starts by a front-end application sending a request to the service. The service expects both a context and the data to validate as payload. Once received, the service loads the necessary application model. Feature class instances are added to the application model from either the received data or by using default values, if any, for the missing attributes. The service then starts an iterative cycle of validations over feature class instances. The reasoner is first run to checks coherence and consistence of the ontology from Descriptive Logic. The service then goes through the list of feature classes and checks the *dependency* rules that control the activation of class validation. If all the dependencies are fulfilled, the service triggers the validation of the feature class. Once all the validation rules have completed or gone beyond a timeout, the service collects and aggregates the results for the feature class. At the end of an iteration, the system checks for generated content, updates the original application data and launches a new iteration. The cycle continues until there is no more class activation; at which points the service generates and returns the final validation report assembled from the report of individual feature classes.

4 RELATED WORKS

There are several interesting works on validation. Their approaches to validation takes many different forms from simple comparison, to domain-independent, to ontology-based validation such as the present work.

One very early work with a lot of similarities, is from Tu et al. (Tu et al., 1995) from the university of Stanford and the original founders of the modern and popular tool called *Protege*. One of their early research projects was the development of an architecture to tackle problem-solving using small-grained reusable components structured by ontology. The architecture was splitting the problem-solving into three layers of ontologies: a domain ontology, an application ontology, and a method ontology. The authors emphasized the importance of representing domain knowledge in formal domain ontologies to make the active content reusable and shareable; and the importance of facilitating the edition and maintenance of knowledge content. To this end, they developed a user interface called MAITRE.

Aljawarneh et al. (Aljawarneh et al., 2010) developed a validation solution specifically designed supplement standard firewall technologies to addresses web vulnerabilities at the application level. The service architecture takes advantages of RDFa annotations embedded in XHTML web pages and extracts the annotations to create an ontology in order to validate the subsequent interactions between client and servers using the created ontology. The service is qualified as survivable as it stands, as the present work, as a middle-ware service.

Da Cruz et al. (da Cruz and Faria, 2008) used an ontology to generate a user interface. With the data collected from the user-interface, the system instantiates classes within the ontology and performs reasoning to validate the data. The approach is aligned to this work as it derives an application model from a general domain ontology.

Tanida et al. (Tanida et al., 2011) built a solution to validate the trace of user interactions with a web-applications built on AJAX, using a temporal logic model.

Finally, the work of Gatti et al. (Gatti et al., 2012) has also many similarities to the present work. It introduces the concept of Data Analysis as a Service (DAaaS) and describes the architecture of a scalable service. The content is received in JSON, converted to XML and processed. The validation involves existing grammar and rule-based schema languages. The set of constraints is domain-independent. The validation process goes through a number of steps such as checking syntax with XSD schema, and validating semantic using the Schematron language. Upon completion, the service reports on the validation by listing failed assertions along with the necessary information to track and correct the failures. The architecture emphasizes the service capabilities for *data assistance* in a somewhat similar fashion as the present work recommendations and providing information on attributes to assist the user-interface.

5 DISCUSSION

A lot of effort has gone into the designing of the user interface for knowledge acquisition. Some of the initial observations were used to segment and simplify the creation of the various entities composing the knowledge structure. One feature that was central to this effort was the auto-complete feature for the edition of validation rules, either from Descriptive Logic or Code Logic.

On performance, the service is naturally dependent on the complexity of the application model. One of the future challenges will to assemble an efficient assessment and prediction engine over response times to promote fluid interactions between the front-end application and the service.

Finally, we identified that the proposed service architecture could be modulated to tackle different situations. The service while focused on data validation is inherently a validated entry-point to Cloud compute back-end services and could directly drive executions. The same service architecture has also the potential to be shrunk and dispersed into a wider, fine grained, scalable framework for data quality assessment.

6 CONCLUSIONS

The present work proposed a service architecture for the purpose of validation and recommendation with a coherent workflow, a well-defined knowledge structure and a comprehensive range of validation mechanisms. The novelty of the approach takes its root in recent advances in Cloud technologies, service-oriented architecture, domain engineering and a knowledge structure backed by ontology. The proposed service is critically positioned within modern cloud infrastructures; and proves to be a practical example of how ontology can benefit service-oriented architecture.

REFERENCES

Aljawarneh, S., Alkhateeb, F., and Al Maghayreh, E. (2010). A semantic data validation service for web applications. *Journal of Theoretical and Applied Electronic Commerce Research*, 5(1):39–55.

Baset, S. and Stoffel, K. (2018). Object-oriented modeling with ontologies around: A survey of existing approaches. *International Journal of Software Engineering and Knowledge Engineering*, 28(12):1775–1795.

da Cruz, A. M. R. and Faria, J. P. (2008). Automatic generation of interactive prototypes for domain model validation. In *Third International Conference on Software and Data Technologies*, volume 2, pages 206–213. INSTICC, SciTePress.

ESS handbook (2018). Methodology for data validation. Technical Report 2.0, European Statistical System.

Faiez, Z., Challita, S., and Merle, P. (2017). A model-driven tool chain for occi. In *35th International Con-*

ference on Cooperative Information Systems, pages 7–26, Rhodes, Greece. Springer.

Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine.

Gatti, M., Herrmann, R., Loewenstern, D., Pinel, F., and Shwartz, L. (2012). Domain-independent data validation and content assistance as a service. In *19th International Conference on Web Services*, pages 407–414. IEEE.

Gil, Y. and Melz, E. (1996). Explicit representations of problem-solving strategies to support knowledge acquisition. In *13th National Conference on Artificial Intelligence*, pages 469–476, Portland, Oregon, USA.

Hohpe, G. and Woolf, B. (2004). *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. The Addison-Wesley Signature Series. Prentice Hall.

Kezadri, M. and Pantel, M. (2010). First steps toward a verification and validation ontology. In *International Conference on Knowledge Engineering and Ontology Development*, volume 1, pages 440–444. INSTICC, SciTePress.

Khouri, A. L. and Digiampietri, L. (2018). Combining artificial intelligence, ontology, and frequency-based approaches to recommend activities in scientific workflows. *Revista de Informática Teórica e Aplicada*, 25(1):39–47.

Kim, C. H. P. (2006). On the relationship between feature models and ontologies. Master's thesis, University of Waterloo, Waterloo, ON, Canada.

Kim, J., Gil, Y., and Spraragen, M. (2010). Principles for interactive acquisition and validation of workflows. *Journal of Experimental & Theoretical Artificial Intelligence*, 22(2):103–134.

Motik, B., Shearer, R., and Horrocks, I. (2009). Hypertableau reasoning for description logics. *Journal of Artificial Intelligence Research*, 36(1):165–228.

Musen, M. A. (2004). Ontology-oriented design and programming. In *Knowledge Engineering and Agent Technology*, pages 3–16. IOS Press.

Shanks, G., Tansley, E., and Weber, R. (2003). Using ontology to validate conceptual models. *Communications of the ACM*, 46(10):85–89.

Strmečki, D., Magdalenić, I., and Kermek, D. (2016). An overview on the use of ontologies in software engineering. *Journal of Computer Science*, 12(12):597–610.

Tanida, H., Fujita, M., Prasad, M., and Rajan, S. P. (2011). Client-tier validation of dynamic web applications. In *6th International Conference on Software and Database Technologies*, volume 2, pages 86–95. INSTICC, SciTePress.

Tu, S. W., Eriksson, H., Gennari, J., Shahar, Y., and Musen, M. A. (1995). Ontology-based configuration of problem-solving methods and generation of knowledge-acquisition tools: Application of protege-ii to protocol-based decision support. *Artificial Intelligence in Medicine*, 7:257–289.

W3C (2004). Owl, web ontology language. www.w3.org/TR/owl-features/. Last access 20th April 2019.

Wang, H. H., Li, Y. F., Sun, J., Zhang, H., and Pan, J. (2007). Verifying feature models using owl. *Journal of Web Semantics*, 5(2):117–129.

Zdun, U. and Dustdar, S. (2006). Model-driven and pattern-based integration of process-driven soa models. *International Journal of Business Process Integration and Management*, 2:109–119.