Towards a Knowledge Interface for Java Applications

Mario Scrocca[1] and Riccardo Tommasini[2]

[1] Cefriel – Politecnico di Milano, Italy
mario.scrocca@cefriel.com
[2] University of Tartu, Data System Group, Estonia
riccardo.tommasini@ut.ee

Abstract. We are witnessing the spread of data-driven organizations. In particular, enterprise software is migrating from legacy monolithic data systems to *microservices*, which embrace the distributed nature of data. Despite offering many benefits, microservices pose several challenges in the development and maintenance of information systems. Techniques like Domain-Driven Design and Data Mesh emerged to ease data and system integration by infusing domain knowledge within the developing process. In this scenario, knowledge representation and reasoning (KRR) can come to the rescue. Thus, in this paper, we present the Java Knowledge Interface (JKI), whose goal is allowing Java programmers to semantically lift the applications' data model (compile time) and state (runtime) with respect to an OWL2 ontology.

Keywords: Poster · Knowledge Representation · Semantic Debugging · Semantic Programming · Semantic Microservices

1 Introduction

We are witnessing the spread of data-driven organizations. In particular, enterprise software is migrating from legacy monolithic data systems to *microservices*, which embrace the distributed nature of data. Despite offering many benefits, microservices pose several challenges in the development and maintenance of information systems. Microservices have to speed up the *velocity* of the software lifecycle, supported by the adoption of novel techniques for continuous delivery and integration. Moreover, different independent teams often develop microservices, which do not necessarily share the exact requirements. In practice, the adoption of microservices may lead to small incremental changes that can integrate different data products [6]. Thus, techniques like Domain-Driven Design [1], Event Sourcing, and Data Mesh[3] evolved to ease data and system integration by infusing domain knowledge within the developing process.

Copyright © 2021 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

[3] https://martinfowler.com/articles/data-mesh-principles.html

CEUR-WS.org/Vol-2980/paper327.pdf

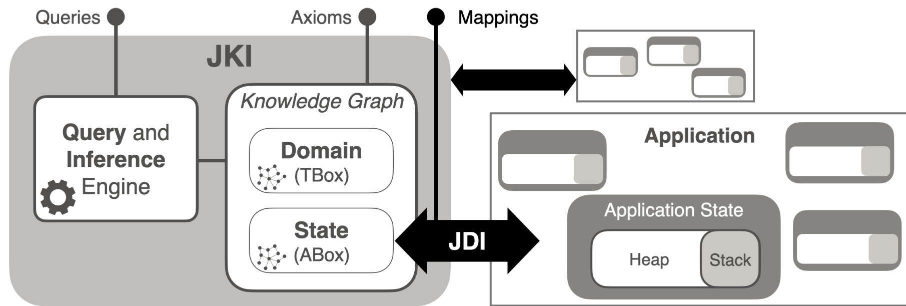

Fig. 1: The Java Knowledge Interface

In this scenario, knowledge representation and reasoning (KRR) can come to the rescue. Indeed, ontologies can model the application domains to ensure semantic interoperability among different data systems. Intuitively, the combination of KRR and software engineering (SE) have been studied already [4,5,7][4]. Leinberger et al. [5] studied the usage of SHACL for type checking code that queries RDF graphs. More recently, Kamburjan et al. [4] discussed the advantages of semantically lifting the state of a software program through ontologies.

In this paper, we present the Java Knowledge Interface (JKI), which falls into *Ontology-driven development* [2]. The goal of JKI is to allow Java programmers to semantically lift the applications' data model and state (runtime) with respect to an OWL2 ontology. Moreover, JKI allows querying and reasoning for the current application state as a knowledge graph (KG). We demonstrated the feasibility and relevance of JKI developing an initial proof-of-concept that exploits a Java Debug Interface (JDI) and OWL API.

Outline. Section 2 defines the proposed JKI. Section 3 describes the implemented proof of concept. Section 4 draws the conclusions.

2 Java Knowledge Interface

In this section, we present the Java Knowledge Interface (JKI). Figure 1 shows an overview of JKI. As mentioned before, JKI aims at running a multitude of reasoning tasks over Java applications. In particular, JKI must enable two inference scenarios: (1) **Static**, i.e., applying a given reasoning task to the application data model at compile time. Java applications rely on a specific data model describing the domain addressed by the software program. JKI requires to map relevant Java classes (i.e., the application data model) to corresponding OWL classes in the given TBox. Notably, not all classes of the application model need to be mapped to the ontology. (2) **Dynamic**, i.e., executing a given reasoning task over the application state at runtime. Given a generic running Java application, its runtime state is represented by instances of classes of the model that are handled and interconnected in-memory (*heap* and *stack*). JKI should be

[4] we invite the reader to consult [2] for a survey of different approaches

responsible for connecting to the running Java application, lifting the state integrating it in a KG and execute different *reasoning* tasks on it. The JKI should implement *Semantic State Lifting* by mapping the active class instances to the corresponding OWL individuals. In practice, JKI should allow checking that no inconsistencies are generated at runtime considering one or more snapshots of the application state, potentially enriched with external domain knowledge.

The obtained KG offers an abstraction to reason on the domain logic coherence over the implementation details of one or more applications. In particular, we distinguish two scenarios (A) **Intra-Application** Inference, where JKI is executing a given reasoning task over a KG resulting from the state lifting of an application. (B) **Inter-Application** Inference, where JKI is executing a given reasoning task over a KG resulting from the integration of the lifted state of n different applications. Intuitively, all four combinations are possible. Indeed, the JKI users might be interested to verify the compliance of a single application data model to the ontological specification (1A) or check if a static alignment exists across applications (e.g., different microservices in a distributed architecture) (1B). On the other hand, the JKI users might want to reason about a single application state (2A) or the integration of many (2B).

3 Proof of Concepts

To validate JKI we developed a proof-of-concept (POC)[5] that exploits the Java Debug Interface (JDI)[6], and OWL API [3].

The POC consists of two examples, i.e., `app.artmarket` and `app.eshop`, which respectively represents a paint shop and e-commerce. These examples are designed to show the benefits of the lifted states for debugging the application state. The *E-Shop demo* shows how the integration between the application state and external data allows validating (consistency checking) the discounts. For example, including in the ontology, an axiom about discounts available only to particular customers allows evaluating, at runtime, the validity (consistency) of the application state. The *Art Market demo* shows how an ontology allows reasoning over the knowledge that is not explicitly represented in the classes. For example, each individual of the class Artist associated to a Paint can be axiomatically inferenced as an instance of the Painter subclass, thus enabling querying considering external knowledge about Painters.

The applications data models are made available to JKI using a set of maps that are configured to bind IRIs of the ontology to Java classes (to generate individuals) and their fields (to generate data and object properties). Multiple property relations from the same instance are stored in named lists.

The JKI is implemented using the JDI API to connect to the Java application and place breakpoints to collect state snapshots. Indeed, using JDI we do not need to modify the monitored Java application. The only requirement is to run the application in debug mode, where it is possible to receive updates

[5] `https://github.com/marioscrock/java-reasoning`
[6] `https://docs.oracle.com/javase/8/docs/jdk/api/jpda/jdi/`

from the JVM via socket. Once reached a breakpoint event, the JVM execution is suspended and a JKI component (implementing the *InspectToAxiom* interface) incrementally inspects the application state and generates ABox axioms considering a given configuration.

Using OWL API, the POC allows its user to programmatically: (i) add other axioms to the generated KG, and (ii) answer Description Logic queries on the KG. The POC periodically runs a reasoning routine every time the JVM is suspended for a breakpoint. A buffer is kept to ensure that previously added ABox axioms (generated by the last inspection of application instances) are deleted and only the active instances are left in the KG.

4 Conclusion and Future Works

In this paper, we presented a proof-of-concept implementation of a Knowledge Interface for Java programs. JKI supports two kinds of reasoning, i.e., static reasoning about the application data model at compile-time, and dynamic reasoning on the application state at runtime. We showcased the feasibility of JKI with two examples, i.e., an Art Market and an E-Shop. As future work, we plan to (i) design an annotation-based mechanism to map the application data model to the OWL classes, (ii) enable the usage of SHACL shapes to validate the semantically lifted state of the application, (iii) integrate other forms of application data within the KG, i.e., traces, logs, and metrics [8], (iv) integrate RSP4J [9] within JKI for stream reasoning about application state changes. Finally, we will investigate the factors impacting the performances of the JKI.

References

1. Evans, E., Evans, E.J.: Domain-driven design: tackling complexity in the heart of software. Addison-Wesley Professional (2004)
2. Happel, H., Maalej, W., Seedorf, S.: Applications of ontologies in collaborative software development. In: Mistrík, I., van der Hoek, A., Grundy, J., Whitehead, J. (eds.) Collaborative Software Engineering. Springer (2010)
3. Horridge, M., Bechhofer, S.: The OWL API: A java API for OWL ontologies. Semantic Web **2**(1), 11–21 (2011)
4. Kamburjan, E., Klungre, V.N., Schlatte, R., Johnsen, E.B., Giese, M.: Programming and debugging with semantically lifted states. In: European Semantic Web Conference. pp. 126–142. Springer (2021)
5. Leinberger, M., Seifer, P., Schon, C., Lämmel, R., Staab, S.: Type checking program code using SHACL. In: ISWC - New Zealand, Proceedings. LNCS, Springer (2019)
6. Overeem, M., Spoor, M., Jansen, S.: The dark side of event sourcing: Managing data conversion. In: SANER, Klagenfurt, Austria. IEEE Computer Society (2017)
7. Puleston, C., Parsia, B., Cunningham, J.A., Rector, A.L.: Integrating object-oriented and ontological representations: A case study in java and OWL. In: ISWC, Germany, October 26-30, 2008. Proceedings. LNCS, Springer (2008)
8. Scrocca, M., Tommasini, R., Margara, A., Valle, E.D., Sakr, S.: The Kaiju project: enabling event-driven observability. In: DEBS: Montreal, Canada. ACM (2020)
9. Tommasini, R., Bonte, P., Ongenae, F., Valle, E.D.: RSP4J: an API for RDF stream processing. In: ESWC, Virtual Event, Proceedings. LNCS, Springer (2021)