Designing a knowledge representation interface for cognitive agents

Bagosi, T.; de Greeff, J.; Hindriks, KV; Neerincx, MA

DOI
10.1007/978-3-319-26184-3_3

Publication date
2015

Document Version
Final published version

Published in
Proceedings of the 3rd International Workshop on Engineering Multi-Agent Systems, EMAS 2015

Designing a Knowledge Representation Interface for Cognitive Agents

Timea Bagosi(✉), Joachim de Greeff, Koen V. Hindriks, and Mark A. Neerincx

Delft University of Technology, Delft, The Netherlands
{T.Bagosi,J.deGreeff,K.V.Hindriks,M.A.Neerincx}@tudelft.nl

Abstract. The design of cognitive agents involves a knowledge representation (KR) to formally represent and manipulate information relevant for that agent. In practice, agent programming frameworks are dedicated to a specific KR, limiting the use of other possible ones. In this paper we address the issue of creating a flexible choice for agent programmers regarding the technology they want to use. We propose a generic interface, that provides an easy choice of KR for cognitive agents. Our proposal is governed by a number of design principles, an analysis of functional requirements that cognitive agents pose towards a KR, and the identification of various features provided by KR technologies that the interface should capture. We provide two use-cases of the interface by describing its implementation for Prolog and OWL with rules.

Keywords: Knowledge representation technology · Agent programming framework · Generic interface design

1 Introduction

In cognitive agents, knowledge representation (KR) is used to store, retrieve and update information. In principle, knowledge can be represented in many different ways, but in practice programmers tend to be limited to a specific KR approach that a particular agent programming framework offers. We consider an agent programming framework to be a set of tools for developing or creating cognitive agents. Cognitive agents are entities or pieces of software that percieve and act in an environment, as it is explained more in detail in Sect. 3.2. In many agent frameworks (e.g. Jason [5], 2APL [7] and GOAL [15]), Prolog (or a variant) has become the de-facto standard. There are several reasons why a programmer might prefer to use a different KR from Prolog. A negotiating agent, for example, might need some legislative information, that would need to be encoded when using Prolog. On the other hand, when using OWL, it is possible for the agent to access large amounts of readily available information on the semantic web. However, most agent programming frameworks are committed to a specific KR, and switching to another is not supported.

M. Baldoni et al. (Eds.): EMAS 2015, LNAI 9318, pp. 33–50, 2015.
DOI: 10.1007/978-3-319-26184-3_3

1.1 Motivation

A generic interface for connecting different KRs to cognitive agents is useful for several reasons. Our main motivations are described next.

Knowledge representation languages differ in the *expressivity* that they offer. It is well-known, for example, that negation in logic programming has a semantics based on the Closed World Assumption whereas the family of web ontology languages support the Open World Assumption. Depending on the task, domain, or scenario, one language might be more appropriate than another.

An *agent programmer*, may have a personal preference based on, e.g., ease of use, familiarity, or other factors. An ontology enginner could model a domain of interest easily, but might find other languages difficult.

The Dagstuhl report on "Engineering Multi-Agent Systems" [10] advocates a *component-based* agent design, as this would provide flexibility, reduce overhead, bridge the gap to other architectures and could facilitate more widespread adoption of agent frameworks in real-world applications. A separation of the agent framework and the KR it uses – that is agnostic with respect to the underlying agent programming language – subscribes to this component-based approach, that our interface aims to support.

When using agent programming as part of real-world applications, one commonly has to access existing infrastructure, which typically may include industry-standard approaches for data storage (e.g. Oracle database). Rather than implementing some kind of bridge between these *legacy databases* and the knowledge representation language used in the agent framework on an ad-hoc basis, a much better approach would be the use of a generic interface, so that the agent framework can use the available technology directly. The semantic web offers a wide range of information in RDF standard format, that could be accessed by OWL-knowledgeable agents.

An agent may need to combine knowledge from *multiple sources*, that are either distributed or not. A generic interface supporting a variety of KR languages, allowing the use of several KR sources from several locations is useful in this context. A particular case is when dealing with large multi-agent systems that may include different manifestations of the agents, such as embodied in robots, software agents and modeling users, where they might be of different technologies.

A wide range of agent frameworks could benefit from providing a flexible choice of various logic-based KR formalisms. *Reusability* prevents the need for reinventing the wheel, as the effort to support this interface for a particular agent framework or a particular KR is a one time investment.

1.2 Scope and Methodology

In this paper we propose a design for a *Knowledge Representation Interface* (KRI) that facilitates an easy choice of KR for cognitive agents. Currently, this interface presupposes the adoption of the chosen KR by all agents in an agent programming framework. In principle, it is conceivable that a single agent would

use multiple KRs, or multiple interacting agents would each utilize different KRs. The combination of multiple KRs into a single agent framework poses a number of issues that are investigated by [8]. This work is orthogonal to our work, as our aim is to facilitate the easy integration of an arbitrary *single* KR technology into a cognitive agent. Investigating issues relating to multiple interacting agents that each may use a different KR technology is therefore outside the scope of this work.

Our proposed interface design is applicable to a range of agent frameworks that facilitate agents with mental states, and all classes of KR that comply to the definition of [8], as described in detail in Sect. 3. By supporting the interface, an agent framework facilitates the choice of a technology that provides the required expressivity or other feature, and the choice of a preferred knowledge technology by its user.

Creating a generic KRI poses a number of challenges. For instance, it is important to identify the right abstraction level for the KRI specification. Striking the right balance between a high level description (to be as inclusive as possible) and a low level description that may be close to a particular KR language (to be able to specify the details) is essential for the interface design. Careful consideration is needed when identifying where an agent needs some form of KR, such as to represent the contents of its plans, skills, goals, etc.

We use the following methodology to derive the interface. First, we explore related literature, describing the various approaches of how each agent framework incorporates a specific KR. Usually the choice of representing knowledge through a certain language is implicitly integrated within a given framework, rather than being explicitly considered, let alone providing users with any sort of choice. To the best of our knowledge, no work has yet been done on the design and development of a generic interface that facilitates the use of a range of KRs.

Having identified the need for such a KRI (based on the motivations described above), and given the apparent lack of such a construct in related work, we then present the design of the KRI, governed by the following three aspects: (1) a number of design principles serving as guidelines, (2) the concept of cognitive agents and the functionality requirements they pose towards a KR, and (3) the identification of features provided by various KR technologies that the KRI should be able to provide.

After having presented the KRI, we describe its application with two implementation: in the first implementation the KRI is instantiated with SWI Prolog (representing a logic programming KR language), and in the second it is instantiated with the ontological web language OWL with SWRL rules (a description logics language), with Pellet [31] as the reasoning engine. After that we assess the KRI usability for these two cases, and based on this draw conclusions regarding the interface's effectiveness and limitations.

The remainder of this paper is organized as follows. Section 2 discusses related work on the usage of knowledge representation technologies into agent frameworks with a focus on the agent programming literature. In Sect. 3 we introduce a number of design principles and present a structural analysis of agents and features of KR technologies that guide the design of the proposed interface.

Section 4 presents the design of the KR interface itself and motivates the choices that we have made. In Sect. 5 we discuss two instantiations of the interface (Prolog and OWL). Section 5.3 briefly discusses a preliminary analysis of the interface that was implemented for Prolog, and OWL with rules. Finally, we conclude the paper with future work in Sect. 6.

2 Related Work

In this section we discuss related work with respect to the choice and possible use of KR languages in agent frameworks. It is useful to note here that some agent frameworks such as JACK [33] and Jadex [28] have taken a more pragmatic road, and use object oriented technology in combination with, e.g., XML, to implement the beliefs and goals of an agent, rather than using a knowledge technology in the sense that we use it here (cf. Davis [9]). The focus of our paper is more on generic *logic-based* agent frameworks that use an existing technology for representing an agent's environment.

Most work on logic-based agent programming frameworks has built on top of logic programming or some kind of variant thereof, e.g. 2APL [7], GOAL [15], Jason [5]. Alternatively, several works have described approaches towards the integration of semantic web technologies (such as OWL) into agent-based frameworks. For example, for Jason there exist the JASDL extension [19], which allows for integration with OWL, and as such lets agents incorporate OWL knowledge into their belief base. The Java-based agent framework, JIAC [16], also uses OWL for representing agent knowledge. While comparable in the sense that these systems allows for the use of OWL in the agent framework, the KR interface that we propose here is aimed to provide a practical solution to the more general problem, and to allow a range of KRs to be used in an agent framework.

The work in [22] defines a version of the BDI agent-oriented programming language AgentSpeak based on description logic, rather than one based on predicate logic (e.g. Prolog). The work reported in [12] proposes the use of a semantic web language as a unifying framework for representing and reasoning about agents, their environment, and the organizations they take part in. The work is presented as a first step towards the use of ontologies in the multi-agent framework JaCaMo, but does not discuss the particulars to achieve this.

Probabilistic approaches have also been considered as KRs in conjunction with multi-agent systems. E.g. [32] propose an extension of the 3APL language based on a probabilistic logic programming framework, while [30] discuss the use of Bayesian networks for representing knowledge in agent programs.

Access to external data sources by agents in the IMPACT agent framework [11] is achieved through an abstraction layer, dubbed *body of software code*, that specifies a set of all data-types and functions the underlying data source provides.

DyKnow [13] is a stream-based approach to knowledge processing middleware, supporting knowledge sharing and processing within a single platform. It focuses more on the dynamics of knowledge, and such is orthogonal to our work.

The work described in [8] investigated the issue of integrating multiple KR technologies into a single agent. The paper proposes techniques for combining knowledge represented in different knowledge representation languages. This is orthogonal to our work as our aim is to facilitate the easy use of an arbitrary *single* KR within a cognitive agent framework.

The usefulness of facilitating the use of a particular KR in other frameworks has been recognized in the literature, and has driven several efforts in defining an Application Programming Interface (API) for several technologies. [3,17] for instance, have proposed an API for description logics and OWL respectively, and [6] proposes an API for a Fuzzy Logic inference engine. These APIs are facilitating all aspects of a specific KR. In contrast, in this paper we aim at a generic KRI to connect arbitrary agent frameworks with arbitrary KRs that comply to our minimal assumptions.

Although most work has focused on the integration of logic programming and semantic web technologies and Bayesian networks, we are not aware of any work that has investigated the use of these technologies in agent frameworks in a generic manner.

3 Dimensions of the KRI Design

Our aim is to design a standardized, extensible and easy to use interface that allows for a flexible choice of KR languages in agent frameworks. To this end, we first present our **design approach**. In Sect. 4 we propose an interface specification as a Java-based API. Three design dimensions are taken into account to cover all aspects that can have influence on the design of such an interface. The first dimension concerns the *design principles*, which we discuss in Sect. 3.1. The second dimension concerns the concept of a *cognitive agent* and related assumptions that we make about agent frameworks. In Sect. 3.2 we present a structural and generic analysis of the features and components that are typically required by agent frameworks. The third dimension concerns the *features* that are made available by existing KR technologies that can be supported by the proposed interface. In Sect. 3.3 we analyze and identify these features. Taken together, these three dimensions define the design space of the proposed interface.

3.1 Design Principles

For creating a generic KR interface for agent frameworks, reuse is a key concern. We want the interface to serve all agent frameworks that could benefit from an easy choice of KRs. To this end, we present and briefly discuss various *reuse design principles* that we have taken into account in the design of the interface.

One of the most important reuse principles in the design of a well-defined interface concerns **abstraction**. Abstraction plays a central role in software reuse, and is essential for the reuse of software artifacts [20]. By means of abstraction, important aspects are put in focus while unimportant details are ignored [1,20]. Each KR technology introduces a specific language, and a key issue for

our interface specification is how to abstract from differences in the grammar between KR languages. We want to be largely agnostic about the particular type of agent framework that a knowledge representation is used in. We will only assume, for example, that an agent decides what to do next based on a state representation expressed in some KR language, and will make no stronger assumptions about the particular structure of the mental state of an agent (see for a more detailed discussion Sect. 3.2). Similarly, we want to be largely agnostic about the particular type of KR languages. We assume, for example, that a KR language provides variables, but will not assume that such a language provides rules (which would exclude, e.g., SWRL and PDDL without axioms; see for a more detailed discussion Sect. 3.3). The interface that we propose here provides an abstraction in the sense that it is a *high-level, succinct, natural, and useful specification* that facilitates easy use of KRs in agent frameworks.

Two closely related design principles that are very important when designing for reuse are the principles of **generality** and **genericity** [1]. Generality is achieved by the abstraction of commonalities and ignoring the (detailed) differences that relate to how, when, or where things are done by a technology. Generality is important when looking at different KR technologies, as our aim is to be as general as possible and support any KR class that fits our assumptions. An obvious example is to abstract from the particulars of how a reasoning engine made available by a technology answers a query; an interface should only assume that some engine is made available. Genericity refers to the abstraction of specific parameters of a technology and the introduction of generic parameters that represent generic types. The use of generic parameters is an aid to reusability, because it allows to define generic functionality instead of functionality that is tight to technology specific features.

The principle of **modularity** refers to considerations of size and number of a reusable software components. The general principle dictates to split large software components into smaller subcomponents; the basic idea being that adequate modular design increases reusability. In order to obtain a loosely coupled system, we design a modular interface whose components are determined by the functional requirements it has to fulfill.

3.2 Cognitive Agent Frameworks: Functional Requirements

In this section we examine which features are required for using a KR within an agent programming framework. Importantly, an interface only provides an effective specification if it includes all of the information that is needed to realize its purpose. In other words, the KRI needs to provide support for all of the functions that an agent is supposed to be able to implement. To identify these *functional requirements*, we discuss and make explicit the notion of a cognitive agent that has been used for the interface specification.

Because we do not want to commit to any particular agent concept, we start from the very abstract concept of an agent as an entity that *perceives* and *acts* in its environment of [29]. Starting from this notion of agent, we assume that an agent *maintains a state* in order to represent its environment by means of

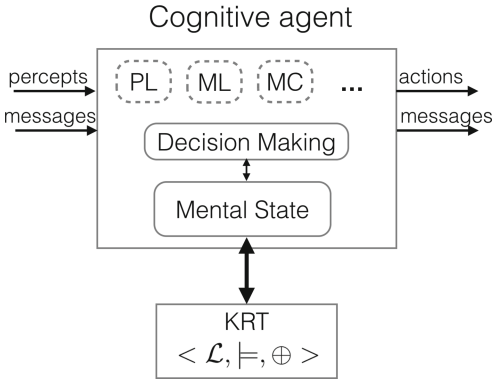

Fig. 1. A cognitive agent architecture, consisting of a mental state and decision making module. Optional components are automated planning (PL), machine learning (ML) model checking (MC), and other modules. Mental states are realized with a KR, accessed through an interface.

a knowledge representation language. As is usual in most agent literature on cognitive agents, we call this agent state a *mental state*, even though we do not make any additional assumptions on the structure of this state. Mental states in agent frameworks differ significantly, and we do not want to commit to any particular framework. A state of a Jason agent, for example, consists of events, beliefs, and plans [4], whereas a state of a GOAL agent consists of knowledge, beliefs, and declarative goals [15].

A cognitive agent (cf. Fig. 1) maintains a mental state in order to be able to evaluate whether certain conditions hold, by *querying* its state. Querying is one of the most important uses of a KR technology, as it provides an essential capability required for effective decision making of an agent, which we identify here as the main functional component of an agent. Another reason for an agent to maintain a mental state is to maintain an accurate and up to date representation of the state of its environment by *updating* its state with information received through percepts or other events. The basic notion of agent of [29] already implies that an agent is connected to an environment. Such an agent needs to be able to align *percepts* it receives from an environment with its mental state. An agent also needs to be able to evaluate when it can perform an action, and represent what the effects of an action are. In other words, an agent needs some kind of *action specification* to be able to interact with its environment. Finally, we also assume that an agent can be part of a multi-agent system, and is able to *exchange messages* with other agents. Figure 1, which represents the basic agent architecture that is used in the design of the interface, illustrates this.

Summarizing, we identify the following list of minimum capabilities that are required for creating a functional cognitive agent in a multi-agent framework:

1. *represent* the contents of a *mental state*
2. *store* the contents of a mental state
3. *query* the contents of a mental state in order to evaluate conditions by means of some form of reasoning

4. *update* the contents of a mental state to reflect changes in an environment
5. *process percepts* received from an environment
6. *process actions* by evaluating *preconditions* and reflecting *postconditions*
7. *process messages* exchanged between agents

Next, we discuss the functional requirements that these items introduce towards the KR language and technology, and its consequences regarding the design of a generic interface.

Item 1 above does not introduce any requirements as representing is the main purpose of a knowledge representation language. We do not assume, for example, that an agent's state must be consistent in a specific sense. **Item 2** requires that a KR provides support for the (temporary) storage of the contents of an agent's state. This item does not require such a store to be persistent. **Item 3** requires support from a KR technology to evaluate queries on the mental state of an agent. Without any additional assumptions on the structure of a mental state, this item does not introduce new requirements, as querying is a common feature provided by the KR. **Item 4** requires support from a KR technology to update, i.e., to add and remove, contents of a mental state. This is a basic requirement, that only requires that a KR makes available the capabilities of adding and removing content from a store. **Item 5** requires support in principle for representing any information that an agent receives from its environment, and updating the representation of the environment that the agent maintains, these functionalities being already mentioned in Item 1 and 4. **Item 6** requires that the knowledge representation language can also be used to represent the actions that the agent can perform. We assume an action can be expressed as a list of preconditions and postconditions. It is essential to be able to evaluate whether an action can be performed, processing preconditions being fulfilled by the querying functionality of Item 3. The ability to process the effects of an action, i.e. its postconditions, is fulfilled by item 4 that requires support for updating a mental state. **Item 7** requires support for representing and processing the content of a message that agents exchange. We assume here that communication between agents does not introduce any additional requirements besides those already introduced by previous items 1–4.

Apart from very generic features and components of cognitive agents such as mental state, we also take into account that agent frameworks might support additional optional components that are only available in some frameworks, but not all. The components drawn with dotted lines in Fig. 1 represent these components. For example, an agent framework might support automated planning (PL), model checking (MC), and even learning mechanisms, such as, for example, reinforcement learning (RL). These components do not exhaust the possible optional components as indicated by the three dots. It is likely that such optional components introduce additional demands on the interface, since they provide support to an agent framework through the interface.

3.3 Features of Knowledge Representation Technologies

Figure 1 includes an abstract definition of a knowledge representation technology as a tuple $\langle \mathcal{L}, \models, \oplus \rangle$, where $\mathcal{L}$ is a language, $\models$ is an inference relation, and $\oplus$ is

an update operator (definition taken from [8] and based on [9]). The inference relation evaluates a subset $L_q \subseteq L$ of expressions of the language called *queries* on a store or set of language elements. We consider our interface to be applicable to the classes of KR that comply to this definition.

This notion of a KR technology covers most, but not all existing technologies, including, for example, logic programming (Prolog), database languages (e.g., SQL, Datalog), semantic web languages (e.g., OWL, SWRL), description logic programming (DLP), planning domain definition language (PDDL), and fuzzy logic. Answer set programming (ASP) provides a computational model that we do not support, even though the pure reasoning support of ASP could be integrated using the proposed interface. Using this abstract definition as a starting point, we identify more concrete features and functions that are supported by KR technologies that can be included in an interface specification.

Having described KR technologies in a general sense above, we now define those modules that have an impact on the design of a generic KR interface, either on its structure or its provided functionality.

Language. Although expressivity is a very important aspect of any knowledge representation language, we do not consider it here, as it does not appear to be useful to control expressivity by means of a KR interface. It is essential for a KR to provide a *parser*, necessary to be able to operate with the textual representation of the language, and perform syntax checking. Syntax highlighting is an extra feature that the parser can provide.

Support for *data types* may widely differ between KRs, but it is important for the engineering of practical agent frameworks. Typically, basic data types such as (big) integers, floats, booleans, strings, and lists are distinguished from more complex data structures such as stacks in programming languages.

Storage. The main purpose of a storage is to store knowledge. As a basic feature of any KRT is a knowledge base, *creating a store* is an important requirement towards a generic abstraction. In addition, *modifying a store* poses the requirement to be able to insert into and delete from a knowledge store.

Even though we did not identify a functional requirement for stores to be *persistent* in Sect. 3.2, still, a knowledge technology may provide support for persistence, and a KR interface may make this capability available to an agent. An example for such a knowledge technology is persistent triple stores for ontologies. This feature should be included in order to create a knowledge base that needs to be preserved for a later use.

Integrating knowledge from other sources can be realized in many forms, such as accessing existing (legacy) databases, or accessing information on the web. One example is the linked open data repositories of the Semantic Web. This feature, however favorable, cannot be considered as a general requirement.

Reasoning. *Querying* is the basic operation to retrieve information from a knowledge base. We can assume the basic form of querying is to retrieve ground data that matches a query pattern with free variables. Without querying there can be no interaction with a knowledge base, hence it is a main requirements towards a KR interface.

Parallel querying is to be able to ask multiple queries simultaneously. This feature is available for some technologies only (like triplestores), but not for others (such as Prolog), where one needs to first exhaust all solutions of a query at a time, hence it is considered an extra feature, and not a basic requirement.

We assume that a substitution based *parameter instantiation mechanism* is supported, as is usual for logic-based languages for all practical purposes. Note that this does not mean that we make any strong assumptions about the domains of computation. Query results are in the form of bindings between variables and some arbitrary terms. A substitution to represent a variable to term binding therefore is the basic form of expressing a query result.

Other. *Error handling* provides support for errors that might occur during parsing, knowledge base creation, modification, or other language-related operations. Some form of error handling is indispensable from an interface.

A knowledge technology that supports *modularization* facilitates the structuring of knowledge into different modules. This feature may greatly enhance the simultaneous development of knowledge by a team of developers. A modular architecture might greatly influence our design of interface, as mappings between the modules of the knowledge and the interface might be identified.

Three forms of *logical validation* can be supported by a KR: consistency, satisfiability and validity checking. As these validation forms are either provided by the technology or not, we cannot generalize it into a feature requirement.

Summarizing the above, we identified the following list of basic features and extra features:

Basic Features	***Extra Features***
1. Parsing	1. Persistent storage
2. Data types (including checking)	2. Integrate other knowledge sources
3. Creating a store	3. Parallel querying
4. Modifying a store	4. Modularization
5. Querying	5. Logical validation
6. Parameter instantiation	
7. Error handling	

4 The KR Interface

Next, we describe the KR Interface (KRI) designed, a Java-based API to address the issues of creating a generic, a specific KR-independent abstraction. The link to our repository, where the interface is located, can be found at [14]. Throughout the description of the interface we show how each design choice was based on the generic features of KRTs, described in Sect. 3.3, and how it fulfills the functionality requirements that an agent programming framework poses in Sect. 3.2.

Based on the principle of modularization, we want to ensure a separation of concerns related to *language*, *storage*, *reasoning*, and *others*. We propose a structured interface design, such that it facilitates these sub-interfaces, as described next in detail.

Language. The language module of the interface contains the abstract grammatical constructs of a KRT. This fulfills the requirement of being able to express all items on the list of Sect. 3.2, since the language concepts need to be able to represent the contents of an agent's mental state, queries and updates, percepts of the environment, and agent messages.

Our generic language proposal, shown as a conceptual hierarchy in Fig. 2, abstracts any language construct into the higher level `Expression` concept, corresponding to a well-formed sentence or formula in the knowledge representation language. An expression can be of type: `Term`, `Update`, `Query` and `DatabaseFormula`. A `Term` can be simple: `Var`, and `Constant` or complex: a `Function`.

From a KR language's point of view, differentiation between the concepts of *querying* and *updating* is dictated by the syntax, and hence can differ per language. From an agent programming's perspective such a distinction is necessary to require that performing a query never results in an update. It would be difficult to understand the behavior of a system that can change the state as a side-effect of performing a query.

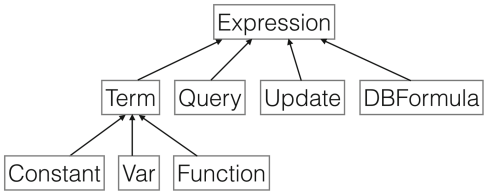

Fig. 2. Language concepts architecture

The `Term` concept represents a language construct of a formula or sentence (ground formula, i.e. without free variables). It can be simple or complex. A variable is a simple term expressed with the concept `Var`. The interface does not enforce variables to be present, however, most languages that support parameter instantiation and querying, need to represent variables. Another simple term is a `Constant`, which is a basic unstructured name that refers to some object or entity, e.g. a number. A `Function` is the representation of a complex term, with a functor and arguments. No restriction on the type or the number of arguments is imposed.

A `Substitution` is a mapping of distinct variables to terms. A substitution binds the term to the variable if it maps the variable to the term. A substitution may be empty. Its functionality includes the usual map operations. It fulfills Item 6 of the language features' list, namely, to have some form of substitution-based parameter mechanism, as we have assumed a set of substitutions to be also the result of a query.

An `Expression` is any grammatically correct string of symbols of a KR language, fulfilling the responsibility of Item 1 of Sect. 3, to be able to represent the contents of an agent's mental state. Every expression has a different signature,

a definition of the form *operatorname/arity*, where the operator name is the functor, and the arity is the number of arguments associated with the operator. In case we need to unify two expressions, the most general unifier method returns a substitution that makes two expressions equal. To apply a substitution to an expression means to substitute variables in the expression that are bound by the substitution with the term bound to the variable, or, only rename it in case the substitution binds a variable to another one.

It is important for an agent to be able to understand which expressions it can use to query, put in a database, and to update a database with. A `DatabaseFormula` stands for an expression that can be inserted into a storage facility. Usually, this is a formula with all ground terms, and no operator that needs more processing, e.g.: conditionals. The `Query` concept is used to query the database, and hence it should contain at least one free variable. An `Update` is semantically equivalent with the combination of a delete and an insert operation. To reflect this, it offers two methods to retrieve the list of database formulas to be added and to be deleted from the knowledge base. For example, in Prolog these classes are different, but may overlap: database formulas are facts (positive literals), a query is an arbitrary conjunction of literals, and an update is a conjunction of basic literals, where basic means the predicate used in the literal is not defined by a rule.

Based on the assumption that every KR should provide its own parsing mechanism identified in Item 1 of the identified KR features' list, the interface should provide a parser for parsing the source (files) represented in the KR language. In case a parser initialization error occurs, proper error handling should be defined and provided.

The `Parser` class fulfills the functionality of a KR to provide its own parser, Item 1 of Sect. 3.3. We abstract a parser to receive an input source file, and return language constructs of our KR interface; database formulas, queries, updates, terms, etc. In case an error occurs during parsing, a method to get the errors returns the source object, which can be inspected for error handling purposes.

Basic *data types*, such as numbers (integers, floats), strings, booleans, are provided together with the functionality of returning the data type of a constant, and data type checking, thus fulfilling the requirement mentioned as Item 2 of Sect. 3.3.

Storage. To create a storage, the main class of the interface provides the way to create a database in the specific KR it hides away. Using the `getDatabase(Collection<DatabaseFormula> content)` method, it creates a new `Database` with the provided content, that is a list of database formulas to be inserted in the database before it being returned. Thus it fulfills the requirement of creating a store by Item 3 of Sect. 3.3.

The `Database` class fulfills the second item of the functional requirements listed in Sect. 3.2. It holds the content represented in the KR language, viewed as a set of `DatabaseFormula`-s. It provides the functionality to store new information in the database by inserting a formula in it, deleting a formula from it, fulfilling the update operation, listed as Item 4 of Sect. 3.2, and Item 4 of Sect. 3.3. Upon insertion of a formula or an update, the database should entail

the information added. The converse applies to deleting a formula, after removal of the formula, in principle, the database should no longer entail the information removed from the database. Any occurring error during insertion, deletion, or destruction of the database is signaled by throwing a database exception.

Reasoning. In order for an agent to inspect its knowledge base, querying functionality has to be provided by the KR, as we mentioned in our assumptions sections, Item 3, and our KR features section, Item 5. The `query(Query query)` method fulfills that functionality, and returns as a result a set of `Substitutions`. In case of an error, a query failed exception is thrown.

Other. The *KRException* and its more specific classes capture the several different types of exceptions, and take the responsibility of error reporting, Item 7 of KR features support list. Separate error types are differentiated for parsing, database operations, failed query errors. In case of parsing, error handling is capable to refer to the source (file) where the error occurred.

5 KR Interface Implementations

In this section we describe the two use cases we studied in depth, and implemented the interface with: Prolog and OWL with SWRL rules. Implementing the KR interface with a new language puts our design choices to the test. We want to investigate how much the interface fits other, different logic-based languages, and provide a first proof of concept for our proposal.

5.1 Prolog Implementation

Prolog was the default logic used for knowledge representation in the GOAL agent framework, as it is a first natural choice for cognitive agent programming, due to its computational powers and the features of logic programming.

Next we describe how we instantiated the interface with SWI-Prolog using the JPL API. The high-level API's class hierarchy consists of the top-level classes: `Term, Query, JPLException`. The abstract superclass `Term` consists of subclasses for variables, compounds, atoms as a specialization of compounds, integers and floats. A `Query` is a wrapper around a term, but it also has a mechanism to hold the retreived results and much more.

A clear match of terminology could be found between the way the KRI captures language constructs and the hierarchy of the JPL API. An `Expression` is a JPL term representing a Prolog expression, the most general language construct in Prolog. The `Var` is mapped to a JPL variable, `Constant`s to integers, floats, and strings, and a `Function` is matched to a Compound term. A JPL term is the representation of both a `Term`, a `DBFormula`, and a `Query`. We chose not to map the JPL's query class to the KRI's `Query`. The former attaches more functionality of the querying process to the class than what the representation a query formula would necessitate. The solution to use a term as a query conveniently matches the JPL idea. Then, performing the check if a term is valid to

be inserted in a database, or can be used as a query is delegated to the parser for efficiency reasons (to avoid such checks at runtime).

An `Update` is a term that is assumed to be a conjunction that can be split into a list of conjuncts. We needed to separate the literals to be added or deleted, so we distinguished the positive from the negative literals (with a preceding `not` operator) to denote the two lists. A `Substitution` is a mapping of distinct variables to terms. We do not use JPL variables as keys, because it has no implementation for hash code, and therefore putting these in a map will fail. Thus, we were forced to using strings.

The main issue encountered during the implementation was the question of a parser. Existing Prolog implementations do not completely conform to the ISO/IEC 13211-1 International Standard. We created our own lexer and parser, following the standard in most cases. Our reasons for deviating have been pragmatically motivated: we wanted to keep our grammar simple, and we did not want it to support certain options that quickly lead to unreadable code, such as using graphic tokens as predicate names, or redefine operators' precedence.

The module feature of Prolog has been used to implement different types of stores. As a conclusion of this choice, modules cannot be made available to an agent programmer any more, as it would potentially clash with the modules that are introduced automatically by the interface.

SWI-Prolog has one fast database to hold all formulas. To be able to differentiate different `Databases` for various mental state construction, we need to specify for each clause which database it belongs to. Our solution was to prefix each database formula with the database name.

Destroying a database removes all predicates and clauses from the SWI-Prolog database, but this is not fully implementable in SWI-Prolog. The JPL interface does not support removing the dynamic declarations. The suggested practice is to reset a database to free up some memory, but after resetting not to re-use this database, but to make a new one.

SWI-Prolog needs access to various libraries at runtime and to load these dynamically. If many agents try to do this at the same time, this creates access errors. A possible solution is to load these libraries upfront when we need them, that implies a check whether we need a library of course. The benefit is that we only need to synchronize the creation of databases and not all query calls. As a pragmatic choice, we solved this issue by adding synchronized querying.

5.2 Ontological Language Implementation

We implemented the proposed KR interface using the OWL ontological language with DL-safe SWRL rules, such an agent being considered a novelty in the field of agent programming. The web ontology language standard (OWL) is a W3C standard recommendation [21] for formalizing an ontology. It is based on the underlying logic called: Description Logic (DL) [2], which has become one of the main knowledge representation formalism. The Semantic Web Rule Language (SWRL) [18] is an OWL-based rule language, and is an extension to the existing ontology language OWL, to provide more expressivity through rules. In order to

preserve decidability, SWRL rules are restricted to so called DL-safe rules [23], which requires each variable in a rule to occur in a data atom in the rule body. A data atom is one that refers to existing named individuals in the ontological knowledge base.

In order to instantiate the interface, two APIs are available for the ontological language: the OWL API [17], that contains representation for SWRL rules as well, or the SWRL API [25] of Protégé-OWL, which is built on top of the OWL API, but extends it further with a query language and provides a parser.

In the following we describe the identified matching between the KRI constructs and the ontological rule language. The higher level concept `Expression` was mapped to `SWRLRule`, that consists of a head and a body. The `Function` concept was mapped to `SWRLAtom`, since atoms are the building blocks of rules, a `Constant` to a `SWRLArgument`, representing a data object or an individual object. A variable is corresponding to `SWRLVariable`.

In order to create a shared, persistent storage, and to access the Semantic Web, a `Database` is mapped to an RDF repository (or triple store). The Resource Description Framework (RDF) is a serialized representation of an ontology, in triple format [26]. The most performant reasoners are available for triple store technologies, and can be queried using the query language SPARQL [27], the adopted standard by the community.

The choice of query language for OWL and SWRL was not a straightforward decision. Query languages for Semantic Web ontologies are categorized into two: RDF-based and DL-based. The default and mostly used querying mechanism is the RDF-based SPARQL, but since it operates on the RDF serialization of OWL, it has no semantic understanding of the language constructs that those serializations represent. On the other hand, the Semantic Query-enhanced Web Rule Language (SQWRL) [24] is a DL-based query language designed on top of the SWRL rule language, with a working implementation provided by the Protégé-OWL API, which would be a very convenient choice in our case.

Faced with the decision between using two different languages for representing knowledge and querying on one hand, or not benefiting from the available advanced triplestore technologies on the other hand, we decided to try to keep the advantages of both. We created a transformation from SWRL rules into SPARQL queries, by treating them as query bodies, with all free variables being considered as part of the query pattern. Having established a querying mechanism, an `Update` then consists of an addition and a deletion operation, provided by the SPARQL Update syntax's insert and delete.

5.3 Discussion of the KRI Implementation

In this section we reflect on the outcomes of our work: the KRI, and how well it performed when put to the test by implementing it with two different KRTs. We reflect on the implementation process, and complement our discussion with extra features that the KRI makes available for the agents. Revisiting the creation of mental states for agents, GOAL poses a difficult requirement: it should be possible to query the combination of a knowledge and belief base (and knowledge

and goal base), i.e., query the union of two bases. It was possible to do this with the proposed KRI, since most KRTs provide either some mechanism to import knowledge from one base into another (e.g., modules in SWI-Prolog) or allow for multi-base querying (federated SPARQL queries for OWL).

An implementation of a specific KR with the interface was highly dependent on the available Java API for the technology. In case several APIs for a language were available, we assessed which one fits best our needs, and can provide most features. Then, the concept hierarchy had to be matched to the interface's corresponding elements, and the functionality correspondence validated. In general the proposed KRI turns out to be generic enough to be implemented for different KR technologies. Following the design principles described in Sect. 3.1 and incorporating features identified in Sect. 3.3, the KRI satisfies all requirements deemed fundamental to represent mental states for cognitive agents (Sect. 3.2); moreover, different types of states (cf. Jason vs GOAL, Sect. 3.2) can be implemented.

The KRI can make use of the extra features that come along with the two languages, e.g., it allows for ontological language with rules to use triple store technologies existing on the web, accessing the Semantic Web thus becoming implicitly available to agents. Another example is parallel querying, that again, agents are at liberty to perform using OWL and SWRL, which comes from exploiting the benefit of a triple store for an agent's mental database. A third benefit of OWL agents that the interface makes possible, is the creation of a shared database, so multiple agents can operate on the same set of knowledge, incrementing data reuse and sharing. On the other hand, when chosing Prolog as the KR, the agent is powerful in computational tasks, and can work easily with lists. This support that would not have been available when chosing OWL, since lists are not by default present in OWL, and are not supported by reasoners that can handle rules. The major benefits of the two languages could be exploited through the instantiation of the interface, which shows that our proposal does not limit the use of a KR for agents.

6 Conclusions and Future Work

In conclusion, this paper introduced a generic KRI that is reusable across a range of agent frameworks that can benefit from the use of different KR languages. Our contribution is a methodological analysis of the features and requirements between knowledge representation technologies and cognitive agent programming frameworks. We proposed and implemented a generic interface to create an abstraction layer and a modular setup to how agents can use a KR. The need for such a KR interface and the apparent lack of such a construct in related work has motivated the design of the interface, governed by the following three aspects (as described in Sect. 3): (1) a number of design principles serving as guidelines, (2) the concept of cognitive agents and related assumptions that we make about agent frameworks, and (3) the identification of features provided by various KRs that are considered as requirements for a KRI. We put this interface

to the test with two knowledge representations, namely Prolog and OWL with SWRL rules, in the agent programming framework GOAL. Based on these two cases we conclude that the KRI is generic enough to support a variety of KR languages, and could be easily applied in the GOAL agent framework.

In the future we will focus on the improvement points identified during the process, and move to a next step of trying different knowledge representation technologies and other agent programming frameworks, to discover the full extent of applicability of, and any modifications needed to our proposed interface.

References

1. Anguswamy, R., Frakes, W.B.: Reuse design principles (2013)
2. Baader, F.: The Description Logic Handbook: Theory, Implementation, and Applications. Cambridge University Press (2003)
3. Bechhofer, S., Horrocks, I., Patel-Schneider, P.F., Tessaris, S.: A proposal for a description logic interface. In: Proceedings of Description Logics, pp. 33–36 (1999)
4. Bordini, R.H., Hübner, J.F.: Jason-A Java-based interpreter for an extended version of AgentSpeak (2007)
5. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming multi-agent systems in AgentSpeak using Jason, vol. 8. Wiley (2007)
6. Cingolani, P., Alcala-Fdez, J.: jfuzzylogic: a robust and flexible fuzzy-logic inference system language implementation. In: 2012 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE), pp. 1–8, June 2012
7. Dastani, M.: 2APL: a practical agent programming language. Auton. Agent. Multi-Agent Syst. **16**(3), 214–248 (2008)
8. Dastani, M., Hindriks, K.V., Novák, P., Tinnemeier, N.A.M.: Combining multiple knowledge representation technologies into agent programming languages. In: Baldoni, M., Son, T.C., van Riemsdijk, M.B., Winikoff, M. (eds.) DALT 2008. LNCS (LNAI), vol. 5397, pp. 60–74. Springer, Heidelberg (2009)
9. Davis, R., Shrobe, H., Szolovits, P.: What is a knowledge representation? AI Mag. **14**(1), 17 (1993)
10. Dix, J., Hindriks, K.V., Logan, B., Wobcke, W.: Engineering multi-agent systems (dagstuhl seminar 12342) (2012)
11. Dix, J., Zhang, Y.: IMPACT: A multi-agent framework with declarative semantics. In: Multi-Agent Programming, pp. 69–94 (2005)
12. Freitas, A., Schmidt, D., Panisson, A., Meneguzzi, F., Vieira, R., Bordini, R.H.: Integrating multi-agent systems in JaCaMo using a semantic representations. In: Workshop on Collaborative Agents, CARE for Intelligent Mobile Services (2014)
13. Heintz, F.: Dyknow: A stream-based knowledge processing middleware framework (2009)
14. Hindriks, K.V.: The GOAL Agent Programming Language hub. https://github.com/goalhub/krTools/tree/master/krInterface
15. Hindriks, K.V.: Programming rational agents in GOAL. In: El Fallah Seghrouchni, A., Dix, J., Dastani, M., Bordini, R.H. (eds.) Multi-Agent Programming: Languages, Tools and Applications, pp. 119–157. Springer (2009)
16. Hirsch, B., Konnerth, T., Heßler, A.: Merging agents and services the JIAC agent platform. In: Multi-Agent Programming: pp. 159–185. Springer (2009)

17. Horridge, M., Bechhofer, S.: The OWL Api: A Java Api for OWL ontologies. Semant. Web **2**(1), 11–21 (2011)
18. Horrocks, I., Patel-Schneider, P.F., Boley, H., Tabet, S., Grosof, B., Dean, M., et al.: SWRL: A semantic web rule language combining OWL and RuleML. W3C Member Submission **21**, 79 (2004)
19. Klapiscak, T., Bordini, R.H.: JASDL: a practical programming approach combining agent and semantic web technologies. In: Baldoni, M., Son, T.C., van Riemsdijk, M.B., Winikoff, M. (eds.) DALT 2008. LNCS (LNAI), vol. 5397, pp. 91–110. Springer, Heidelberg (2009)
20. Krueger, C.W.: Software reuse. ACM Comput. Surv. **24**(2), 131–183 (1992)
21. McGuinness, D.L., Van Harmelen, F., et al.: OWL web ontology language overview. W3C Recommendation **10**(10), 2004 (2004)
22. Moreira, A.F., Vieira, R., Bordini, R.H., Hübner, J.F.: Agent-oriented programming with underlying ontological reasoning. In: Baldoni, M., Endriss, U., Omicini, A., Torroni, P. (eds.) DALT 2005. LNCS (LNAI), vol. 3904, pp. 155–170. Springer, Heidelberg (2006)
23. Motik, B., Sattler, U., Studer, R.: Query answering for OWL-DL with rules. Web Semant.: Sci., Serv. Agents World Wide Web **3**(1), 41–60 (2005)
24. O'Connor, M.J., Das, A.K.: SQWRL: a query language for OWL. In: OWLED, vol. 529 (2009)
25. O'Connor, M.J., Shankar, R.D., Musen, M.A., Das, A.K., Nyulas, C.: The SWRLAPI: a development environment for working with SWRL rules. In: OWLED (2008)
26. Pan, J.Z.: Resource description framework. In: Handbook on Ontologies, pp. 71–90. Springer (2009)
27. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and Complexity of SPARQL. ACM Trans. Database Syst **34**(3), 16:1–16:45 (2009)
28. Pokahr, A., Braubach, L., Lamersdorf, W.: Jadex: A BDI reasoning engine. In: Multi-agent programming, pp. 149–174. Springer (2005)
29. Russell, S., Jordan, H., O'Hare, G.M.P., Collier, R.W.: Agent factory: a framework for prototyping logic-based AOP languages. In: Klügl, F., Ossowski, S. (eds.) MATES 2011. LNCS, vol. 6973, pp. 125–136. Springer, Heidelberg (2011)
30. Silva, D.G., Gluz, J.C.: AgentSpeak (PL): A new programming language for BDI agents with integrated bayesian network model. In: 2011 International Conference on Information Science and Applications (ICISA), pp. 1–7. IEEE (2011)
31. Sirin, E., Parsia, B., Grau, B.C., Kalyanpur, A., Katz, Y.: Pellet: A practical OWL-DL reasoner. Web Semant.: Sci., Serv. Agents World Wide Web **5**(2), 51–53 (2007)
32. Wang, J., Ju, S.E., Liu, C.N.: Agent-oriented probabilistic logic programming. J. Comput. Sci. Technol. **21**(3), 412–417 (2006)
33. Winikoff, M.: JACK intelligent agents: An industrial strength platform. In: Multi-Agent Programming, pp. 175–193. Springer (2005)