Heinrich C. Mayr, Martin Pinzger (Hrsg.): INFORMATIK 2016,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2016 313

# Semantic Object-Oriented Programming (SOOP)

Florian Weber[1] Andreas Bihlmaier[2] Heinz Wörn[3]

**Abstract:** Currently, the interaction between ontologies and general purpose programming languages mostly consists of an indirect mapping between the ontology and the programming language. The language and libraries are often basically used as a programmable ontology editor. In this paper we present a different approach that creates an ontology from regular data objects in a (statically typed) mainstream language (C++). Because in our case the mapping is going from object-oriented programming (OOP) language to the ontology, we can avoid many problems that approaches going into the opposite direction have to face. Furthermore, the interaction between the domains becomes more straight forward and can be mostly kept free of uncommon language constructs. The declarative programming paradigm on the one hand and the (object-oriented) imperative paradigm on the other hand are in this manner combined into the Semantic Object-Oriented Programming (SOOP) paradigm. As a result, SOOP allows programmers to directly use semantic technologies, especially reasoners, for their usual language objects from within C++.

**Keywords:** Ontology, Object-Oriented Programming (OOP), Semantic Reasoner, C++

## 1 Introduction

An ontology is a formal, explicit specification of a shared conceptualization [Gr92][SBF98]. This means that it is a suitable way to represent data for theorem-provers and reasoners, so that they can algorithmically search answers for questions posed in the ontology language. For example, a subject matter, modeled using the ontology, can be checked for consistency. Similarly, the set of individuals satisfying a formula can be extracted.

Since two different domains are brought together here, semantic technologies on the one hand and object-oriented programming (OOP) on the other, we must define some terms (cf. Tab. 1): *Atom* or *individual* refers to a logical individual, i.e. an element of the universe of discourse. An *axiom* is a logical statement that is either true or false. *Predicate* and *quantor* are used in the same manner as in predicate calculus and related logical formalisms. Also *variable* is used in the sense of predicate calculus, more specifically it is synonymous with bound variables.[4] The terms *class* and *object* have the same meaning as in OOP, they refer to the definition of a data type and its instances. *Entity* is used in both domains, it can refer to an individual (semantic domain) or to objects (OOP domain) that represent the individuals within the programming language.

[1] Karlsruhe Institute of Technology (KIT), Institute for Anthropomatics and Robotics (IAR) - Intelligent Process Control and Robotics (IPR), Engler-Bunte-Ring 8, 76131 Karlsruhe, Germany, uagws@student.kit.edu
[2] KIT, IAR-IPR, andreas.bihlmaier@kit.edu
[3] KIT, IAR-IPR, woern@kit.edu
[4] In the context of OOP source code, *C++ variable* is used explicitly to denote program variables.

As will be explained in the following, entity does not only connect the domains terminologically, but is the core concept that facilitates semantic object-oriented programming (SOOP).

Tab. 1: Illustration of how the vocabulary used here is assigned to the OOP and semantic domain.

| OOP | Shared | Ontologies |
|---|---|---|
| Class | Entity | Atom, Individual |
| Object | | Axiom, Variable |
| | | Predicate, Quantor |

The most common method to fill such an ontology with data currently is some kind of ontology editor or ontology API. However, even the APIs that allow to define ontologies from within programming languages, don't allow a direct combination of existing program data structures with the ontology. This results in a strong division between the raw data and the semantic data. The raw data is easily accessible from the programming language (e.g. printing, sorting or direct calculations). The semantic data, presented in the ontology, is easily accessible to reasoners, but falls short with regards to usability in the programming language.

One example for this kind of gap is the OWL-API. While the semantic data is well accessible from Java, the Java objects that represent semantic entities have only a single Java type - and thus have no semantics from the programming language point of view: Entities have types like `OWLLiteral` or `OWLNamedIndividual`; this means that the static typechecker is unable to verify the absence of many obvious errors such as an email address being assigned to a username or a company to vehicle.

Our goal was to find a new approach that brings ontologies together with mainstream programming languages, while avoiding the loss of type information. Thereby, creating a deeper integration compared to existing methods. To avoid the common problems that result from ontologies having greater expressive power compared to object-oriented programming (OOP) languages (as pointed out by [Or07]), especially statically typed ones, we decided to reverse the mapping direction: Instead of creating ontologies with some kind of standalone ontology editor and a subsequent import of the resulting ontology description into the programming language or reasoner, our approach goes directly from OOP to a reasoner. For this purpose, we extended or wrap programming language data structures in such a way that they can be used as semantic entities and combined with existing ontologies. Since ontologies have more expressive power then OOP languages, they can easily represent all the semantic connections that exist in OOP. We call this novel deep integration between semantic technologies and OOP languages *Semantic Object-Oriented Programming (SOOP)*.

The remainder of this paper is structured as follows: In the following section an overview of existing integration approaches between ontologies and OOP is provided. Section 3 details the Semantic Object-Oriented Programming (SOOP) concept and its implementation in C++. Afterwards, the benefits of our integration approach for semantic technologies with OOP are illustrated in an exemplary use case (section 4). The impact of SOOP for this

use case and a general discussion follows in section 5. In the final section, open questions and directions of further research are addressed.

## 2 State of the Art

Nowadays ontologies are primarily created with specific ontology-editors. Particularly well known in this context is Protégé [Kn04]. With regards to the use from within programming languages, there exists a relatively wide range of approaches, some of which we will present in more detail.

However, first, we will describe those approaches in which entities of an ontology are represented in an OOP by a set of generic classes. The aim here is not to create exact representations of the ontology within the OOP-language, but to provide a programmable tool for editing them. These approaches are also called "indirect approaches" [Pu08]. The most important software libraries to mention here are the OWL API [HB11] and Jena [Je13] from the Apache project.

The OWL API is a Java library, which allows access to OWL triples with a comparatively low level of abstraction. Furthermore, it provides a general interface for provers, allowing to easily exchange them with each other. The API provides both the ability to ask questions by means of queries and to edit the ontology. Changes made through the library can be written back to permanent storage. The OWL API is used as backend by Protégé. Apache Jena follows a similar approach, but offers a higher level of abstraction. The basic idea of Jena is to represent data and types of ontologies as objects of classes derived from `OntRessource` (for instance `resource`) and to use these through generic methods. The obvious downside of both OWL API and Jena is the lack of typing with regards to the ontology objects in the Java environment and the strong focus on the use of strings instead of native language objects with types.

Moreover, it is also possible to realize simple queries in Jena with native Java (for example to iterate over the classes of an individual). However, the recommended method for complex requests is to use the query language SPARQL (SPARQL Protocol And RDF Query Language). SPARQL is a language that is standardized by the W3C and modeled after SQL (Structured Query Language). It allows to formulate relatively complex semantic requests, which are then answered by a prover.

At the other end of the existing approaches spectrum, there are attempts to translate the data from ontologies into regular OOP languages, which is also known as the "direct approach" [Pu08]. Unfortunately, there are some fundamental problems with this kind of translation, due to the fact that ontologies are much more general and expressive than most OOP languages. Nevertheless a couple of interesting approaches can be found in literature and are discussed in the following.

ActiveRDF [Or07] for example bypasses many of the common limitations by using Ruby as its implementation language instead of the otherwise widespread Java. As a result many of the common problems posed by a static type system disappear, since Ruby is not only

dynamically typed, but also allows to catch any attempt to use non-existing functions. Yet, it is exactly this dynamic type system that implies several downsides for this approach. Most notably is the lack of basic (compile-time) sanity checks that static type systems provide.

The approach “SWCLOS” presented in “OWL vs. Object Oriented Programming” [KAH05] also uses a dynamically typed Language (Lisp + CLOS). Unlike our work, they concentrate on how a program that uses ontologies can be typed to represent semantic structures, whereas we emphasize the semantics of dynamically created objects.

The final related proposal that tries to unite ontologies and object-oriented programming, can be found in “Integrating Object-Oriented and Ontological Representations: A Case study in Java and OWL” [Pu08]. There an attempted is made to combine direct and indirect representations by making the gap between the ontologies and the OOP data “moveable”. Nevertheless, in contrast to SOOP, there are still two “worlds” with some entities only existing as individuals in the semantic domain or as objects in the OOP domain.

## 3 SOOP

The fundamental idea behind SOOP is to create or populate an ontology from within C++ using regular C++ classes and objects.

This means that we perform the data mapping into the opposite direction compared to the direction of the approaches described in section 2. As an advantage of this direction, many of the usual problems that result from the attempt to map data from a more powerful data representation into a less powerful one are no longer an issue. See table 2 for a summary. Furthermore, the SOOP approach allows us not only to preserve static typing, but also to make active use of the type system in a way that profits the ontology. It is possible to use the C++ entities as regular C++ types, also outside of the ontology context.

Existing knowledge in form of an ontology could be imported into SOOP in two ways: One method directly imports the ontology into the reasoner from a textual representation such as SUO-KIF, i.e. entities from the ontology do not exist as OOP classes and objects. The other method generates OOP code from the ontology[5], which in turn uses SOOP as described below.

[5] The constraints described in Tab. 2 apply here.

Tab. 2: Ontologies provide a much more powerful and dynamic representation of data compared to object-oriented programming languages. Therefore it is easy to translate data from OOP languages into an ontology, but basically impossible to do it the other way round (left side according to [Or07]).

| Ontology → OOP | OOP → Ontology |
|---|---|
| Mapping the dynamic relations of entities into a static type system is basically impossible. | Mapping the type relations of a static type system into a dynamic ontology is trivial. |
| In ontologies, arbitrary objects may be linked using arbitrary relations. This can be hardly translated into object attributes. | Attribute connections are relatively simple in OOP languages and easy to map into an ontology. |
| An ontology is a hypergraph that can naturally represent very complex relations. It is not in general possible to map this to the acyclic inheritance graphs of OOP languages, even if these support multiple inheritance. | The directed, acyclic inheritance graphs of OOP languages are always representable, since they represent subgraphs with respect to the much more general hypergraphs of ontologies. |
| A central mechanism of ontologies is the dynamic evolution of its structure. However, this can results in significant problems, if it is to be translated into a static language. | The constant inheritance structures of OOP languages can easily be transformed into dynamic ontological structures. |

### 3.1 General concept

In order to structure the SOOP concept, we divided it into the following submodules:

- A prover module that handles the communicating with a theorem prover (Z3).
- An ontology module that manages the knowledge and creates queries for the prover module.
- A module that implements the base class of all entities that are backed by a C++ object: `entity`.
- A module that implements formulas in a way that preserves their semantics in a usable manner, while providing a simple syntax in C++ that closely resembles predicate calculus.

The `entity` base class stores a pointer to the associated ontology and an integer that serves as unique ID within the ontology. Due to the unclear semantics of copying[6], no copy-constructor is provided. In contrast, move operations are allowed and simply inform the ontology of the new address of the moved entity. Every class that inherits from `entity` can be used as an entity of the ontology context. Thereby, it is fully supported for these classes to have regular data members and methods, whose meaning or even existence is

[6] See the discussion in section 6.

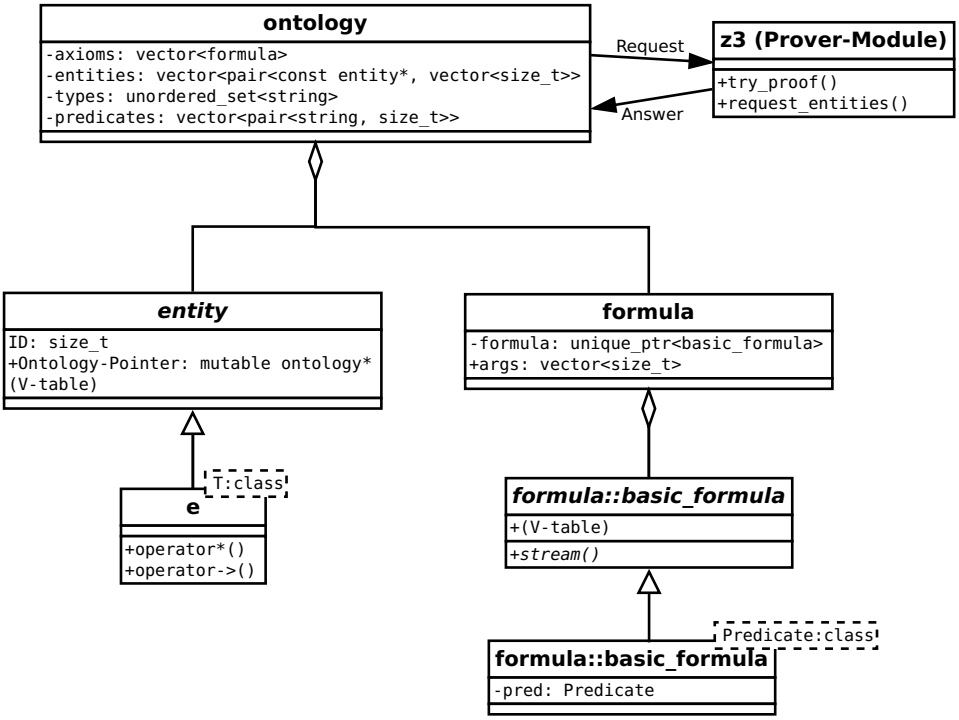


Fig. 1: Overall SOOP architecture in terms of modules and their relations.

not formalized within the ontology. Furthermore, it allows regular usage of the data with normal language facilities, such as sorting or printing functions.

In order to also support the use of already existing types, a class template `e` is provided that wraps the existing types in such a way that the wrapped version (`e<T>`) inherits from `entity` and is therefore usable from within the ontology.

Predicates in SOOP consist of two parts: First, a class template, whose instantiations represent saturated predicates and contain its arguments. Second, a function template that instantiates the class template. By convention the function template is named identically to the predicate (e.g. `f`) and the class template receives `_t` as a suffix (e.g. `f_t`). Through nesting of such predicates, it is possible to create more complicated expressions and types. If `f`, `g` and `h` are for instance predicates and `a` and `b` are entities that are instances of the types `A` and `B`, then `f(g(a), h(a))` will result in the type `f_t<g_t<A>, h_t<B>>` for the entire expression. In order to store different formulas in mixed containers, a wrapper that makes use of type erasure[7] was created.

[7] Type erasure is a technique that allows to treat independent types as if they had a common base class. This is accomplished by a base-class `base` with virtual methods for the desired behavior and a template `derived<T>` that inherits from it and implements them via `T`.

The `ontology` class stores a list of all C++ objects, which are at the same time entities of this ontology. The same class also stores the list of all axioms associated with this ontology. Requests whether a specific set of axioms is satisfiable are processed by creating a full textual representation of the entire ontology, which is then sent to the prover module, whose answer is returned to the caller in C++. Notably, it is not only possible to check for satisfiability of a formula, but also *to request the set of variable assignments in the model and return the actual C++ objects that are associated with them*. Requests for these actual entities are processed similar to the satisfiability check, except that an additional function, which maps all entities to their SOOP ID is defined; the prover is then asked for the value of that function when receiving a suitable entity. These IDs are then used to find the corresponding C++ object, by means of a mapping to its address that is known by the ontology.

In order to get the full power of description logics, quantifiers and variables are needed in addition. The approach that we choose here is to create unique types for all variables by providing a class-template `variable` that receives the identifier as template argument and a class `bound_vars` that receives a variadic number of variables in its constructor and internally safes a textual representation of this list as a string. Thus, it is possible to implement quantifiers in the same manner as predicates with the exception that the former receive an instance of `bound_vars` as their first constructor argument.

A downside of this approach can be seen in the requirement, to define variables before it is possible to use them (at least if a clean syntax is desired). We believe that this is an acceptable trade-off for now, as the `variable` types are empty and can therefore be used for global constants without much overhead.

Combined these features allow the easy creation of complex formulas without requiring a syntax that is unusual for either description logics or C++. Stating for instance that a predicate `f` is transitive can be done with the following valid C++ code:
`forall({x, y, z}, implies(and_(f(x, y), f(y, z)), f(x, z))).`

### 3.2 Implementation

We implemented SOOP in C++14 in “modern C++” style.[8]

The basic idea for a predicate `foo` is to provide a class template `foo_t` and a helper function `foo`:

```
template<typename T>
class foo_t: soop::is_predicate {
public:
    foo_t(const T& arg): arg{arg} {}
    void collect_entities(std::vector<std::size_t>& ids);
    void stream(std::ostream& out,
```

[8] It is available as free software at `https://gitlab.com/FJW/soop`.

```
                const std::vector<std::string>& names) const;
private:
    T arg;
};

template<typename T>
auto foo(const T& arg)
-> foo_t<soop::to_bound_type<T>> {
    return {arg}
}
```

Here `soop::is_predicate` is an empty type that is used by the implementation to dispatch printing and collecting arguments to the correct function templates. The methods `collect_entities` and `stream` are needed to allow simple management of different kinds of formulas and will be explained later in this section. The basic idea behind this model is to allow the user to nest predicates such that each predicate-template would be instated with its arguments types, which in turn might be instantiated predicate-templates.

In order to store different kinds of such formulas in the same container, we created another class `formula` that uses type erasure to store arbitrary formulas while providing uniform access to them. This is where the `collect_entities` and `stream`-methods come into play: In order to keep the information about which entities are used inside the formula easily accessible and editable, their IDs are collected by the former in a depth-first tree traversal and memorized in a regular vector outside the erased type. Printing a formula is accomplished by converting the collected IDs to their textual representations and passing them into the stream method. This in turn also traverses the tree depth-first and uses the IDs in all places where entities were collected earlier.

Since the definition of predicates does require a lot of boilerplate code, SOOP provides four macros that allow the definition of a new predicate in just one short statement:

```
SOOP_MAKE_TYPECHECKED_RENAMED_PREDICATE(Id, Name, Rank, ...)
SOOP_MAKE_RENAMED_PREDICATE(Id, Name, Rank)
SOOP_MAKE_TYPECHECKED_PREDICATE(Id, Rank, ...)
SOOP_MAKE_PREDICATE(Id, Rank)
```

These macros allow to create predicates with arbitrary (even unlimited) rank. Optionally, it is even possible to check the types of the predicate arguments in order to benefit from type safety offered by a typed OOP language. The later three macros are defined in terms of the first universal one, but pick sensible defaults for the other arguments.

### 3.3 A minimalistic SOOP example

In order to illustrate the direct integration between semantic information processing and native OOP in SOOP, a minimal complete SOOP program that will print “Max Mustermann is a parent.” follows:

```
#include <iostream>
#include <string>

#include <soop/onto.hpp>

class person {
public:
    person(std::string name): name{std::move(name)} {}
    std::string name;
};
using person_e = soop::e<person>;

SOOP_MAKE_TYPECHECKED_PREDICATE(parent_of, 2,
                                person_e, person_e)

int main() {
    using namespace preds;
    using namespace soop::preds;
    soop::ontology o{};
    o.add_type<person_e>();
    o.add_predicate<parent_of_t>();
    soop::variable<'p'> p;
    soop::variable<'c'> c;
    // parents and children are persons:
    o.add_axiom(forall({p, c}, implies(parent_of(p, c), and_(
                instance_of(p, soop::type<person_e>),
                instance_of(c, soop::type<person_e>)))));
    // parents are not their childrens' children:
    o.add_axiom(forall({p, c}, implies(parent_of(p, c),
                                       not_(parent_of(c, p)))));

    person_e max{o, "Max Mustermann"};
    person_e erika{o, "Erika Mustermann"};
    o.add_axiom(preds::parent_of(max, erika));

    soop::variable<'s'> s;
    const auto& parent = std::get<0>(
        o.request_entities<person_e>(
            exists({c}, parent_of(s, c)), s));
```

```
    std::cout << parent->name << " is a parent.\n";
}
```

## 4 Exemplary Use Case: Conference Planner

In order to test the usability and practicality of SOOP in a realistic scenario, we created a small conference planner. The motivation for this particular use case is that the problem is easy to understand, realistic and NP-complete.

The basic model that we choose is the following: There are rooms, timeslots, speakers and talks. Every speaker may participate in more than one talk and every talk may have multiple speakers. Obviously, no two talks can be in a single room at the same time and no speaker can hold more than one talk in the same timeslot.

First, we create four C++ classes `talk`, `speaker`, `room` and `slot` that all inherit from `entity`. All of these contain data represented in the usual C++ manner, i.e. without any relation to SOOP. Most of this data does not have a semantic meaning in the use case, but is still important for the final software, such as a description of the talks or a name for the speakers. Only selected fields were given semantics by connecting them to the ontology using SOOP.

This is accomplished by two custom predicates:

```
SOOP_MAKE_TYPECHECKED_PREDICATE(is_speaker_of, 2,
                                speaker, talk);
SOOP_MAKE_TYPECHECKED_PREDICATE(talk_assignment, 4,
                                talk, speaker,room, slot);
```

A `make_ontology` function uses these classes and predicates to create an ontology with all the required types and axioms. In order to define the axioms, one function exists for each axiom, which is returned as a `formula`. One typical example of these functions is shown in the following. It defines that every talk must be held in one room and one slot and that it must be the only talk in that room during that slot:

```
soop::formula uniqueness_of_talks() {
    return forall({t1,t2,s1,s2,r1,r2,sl1,sl2},
        implies(
            and_(
                talk_assignment(t1,s1,r1,sl1),
                talk_assignment(t2,s2,r2,sl2)),
            equal(
                equal(t1,t2),
                and_(equal(r1,r2), equal(sl1,sl2)))));
}
```

After parsing the data about talks, speakers, rooms and slots from a plain text file, the data is stored in containers and axioms are associated to it: Which speakers participate in which talks and all talks are distinct from each other (instead of just different objects referring to the same entity):

```
auto o = make_ontology();
auto data = read_dataset(file, o);

for (const auto& talk : data.talks) {
    for (const auto speaker_id: talk->speaker_ids()) {
        o.add_axiom(preds::is_speaker_of(
                    data.speakers.at(speaker_id), talk));
    }
}

o.add_axiom(soop::preds::distinct_range(
     data.talks.begin(), data.talks.end()));
```

Given this initialisation of the ontology, it is possible to request an assignment for all talks to rooms and time slots. The prover ensures that all constraints - as specified by the axioms - hold. Otherwise the request is not satisfiable and an exception is thrown. It is noteworthy that the returned answers contain references to the actual C++ data with the correct types and can thus be used as all other C++ objects:

```
for (const auto& talk : data.talks) {
    const auto& speaker = data.speakers.at(
            talk->speaker_ids().front());

    auto solution = o.request_entities<room, slot>(
            talk_assignment(talk, speaker, r, sl), r, sl);
    const auto& used_room = std::get<0>(solution);
    const auto& used_slot = std::get<1>(solution);

    std::cout << talk->title()
              << ": in room #" << used_room->number()
              << ", in slot #" << used_slot->time() << '\n';

    o.add_axiom(talk_assignment(talk, speaker,
                used_room, used_slot));
}
```

In the above example code, the assignments are requested sequentially. Thus, it is important to add the selected assignment back to the ontology as axioms. Otherwise, further requests may return assignments that conflict with earlier requested ones.

## 5 Evaluation and Discussion

While some details and most of the axioms where left out, the above provides an impression of the look and feel when using SOOP (as implemented in modern C++). In principal, there are two alternatives to SOOP with very different properties, which is why this evaluation will be split into two parts: The comparison to alternative uses of ontologies and the comparison to approaches that don't use ontologies at all.

With regards to the first point of comparison, SOOP integrates ontologies and regular OOP languages much tighter than existing approaches without requiring idioms that seem out of place for either. The expressive power of the provided description-logic is large enough to describe even complicated problems (such as the NP-complete timetable assignment problem), but still ensures that every statement can be represented in C++. Furthermore, SOOP never requires to compromise type safety or to deal explicitly with translating data from the ontology (usually strings) into actual OOP objects.

Excluding functionality such as parsing the input that are always necessary, creating a working software with SOOP for the use case required less then one hundred lines of code for the actual program logic. From our point of view, the effort is less or equal to the effort necessary with other ontology APIs and it is far less compared to the effort for a solution without using semantic technologies.

This brings us to the comparison with unaugmented OOP languages. The main work when working with SOOP is formalizing the problem, which is a step that is not strictly necessary otherwise. However, this seems a small price to pay, since formalization is always advisable when tackling complex problems, if a correct solution is desired. Using SOOP to formalize the problem has the added advantage that it is easier to spot incomplete formalizations as simple test cases may produce obviously wrong results. Since everything is performed within C++ test cases can be generated in the same manner and with the same tools as for normal unit testing.

With regards to the disadvantages, it is important to note that the theorem prover Z3 currently used in our proof of concept implementation was not designed for use cases such as SOOP. Thus, it can be expected that significant performance improvements are possible in both the prover interface and selection of a different reasoner. Similarly, the scalability of the SOOP approach is largely determined by the employed solver. In the current SOOP implementation, no constrained is imposed on the expressiveness of the axioms in first-order predicate calculus. However, even rather constrained domain logics are able to represent the minimal required expressiveness for OOP relations (cf. Tab. 2).

Another restriction on the usability of SOOP is that formalizing problems can be surprisingly difficult to do correctly, which means that an imperative implementation may be easier for many straightforward problems. However, the later argument applies to ontologies and semantic technologies in general, which often show their full potential only in domains of a certain minimum level of complexity.

One advantage of SOOP certainly lies in the fact that it enables to easily mix imperative solutions with ontological reasoning. Thus, different aspects of the application at hand can be solved in the domain (imperative OOP or declarative semantic reasoning), which is most suitable - or at least feels that way to the application programmer. This also fits nicely with C++ being a multiparadigm language.

Furthermore, libraries using SOOP present a shared conceptualization on two levels: From an OOP perspective, the SOOP classes can be connected to user defined types by inheritance and composition and thus reused just as non-SOOP libraries. At the same time, due to the connection between the OOP types and an ontology in SOOP, different libraries derived from the same base library are also compatible on a semantic level. Thus, a "foundational" SOOP library, directly derived from an upper ontology, has the potential to render, thus far incompatible, OOP libraries at least interoperable.

A much more in depth explanation of the C++ implementation and an extended discussion can be found in [We16].[9]

## 6 Directions of Further Research

During the course of the presented work, we created a working research prototype in C++ and solved several conceptual and implementation issues, but there are still several open topics and a lot of room for further research into SOOP:

The biggest potential for improvement appears to be in the area of static typing: While we implemented a method to state type requirements for predicates, variables are currently still largely untyped. This leads to all the dangers of untyped languages and also means that semantic information available in the OOP type system might not be provided to the reasoner.

Related to the topic of type information is the lack of checks for shadowing variables inside a formula, which results in unexpected answers by the reasoner. All the required information should be available at compile-time and we are convinced that these checks could be performed algorithmically.

Another interesting open question is how to approach copies of entities. In general there are three possible semantic meanings:

- Copy and original are independent of each other and the copy inherits no properties from the original.
- Copy and original are the *same* entity and always share all semantics (flat copy).
- Copy and original are different entities with the same semantics, that are copied as well when creating the copy (deep copy).

[9] The PDF manuscript (German) is available at `http://florianjw.de/ba/soop.pdf`

For each behavior there are cases where this particular behavior is desirable and others where it isn't. For example, the elements in a buffer may even switch desirable semantics during the run of a program. To avoid picking a wrong default, we disabled copying in the current implementation, but want to point out that the architecture supports all of these three possibilities.

## References

[Gr92] Gruber, Thomas R.: A Translation Approach to Portable Ontology Specifications. 1992.

[HB11] Horridge, Matthew; Bechhofer, Sean: The OWL API: A Java API for OWL Ontologies. Semant. web, 2(1):11–21, January 2011.

[Je13] Jena, Apache: Apache jena. jena. apache. org [Online]. Available: http://jena. apache. org [Accessed: Mar. 20, 2014], 2013.

[KAH05] Koide, Seiji; Aasman, Jans; Haflich, Steve: OWL vs. object oriented programming. In: the 4th International Semantic Web Conference (ISWC 2005), Workshop on Semantic Web Enabled Software Engineering (SWESE). Citeseer, 2005.

[Kn04] Knublauch, Holger; Fergerson, Ray W.; Noy, Natalya F.; Musen, Mark A.: The Protg OWL plugin: An open development environment for semantic web applications. Springer, pp. 229–243, 2004.

[Or07] Oren, Eyal; Delbru, Renaud; Gerke, Sebastian; Haller, Armin; Decker, Stefan: ActiveRDF: Object-oriented Semantic Web Programming. In: Proceedings of the 16th International Conference on World Wide Web. WWW '07, ACM, New York, NY, USA, pp. 817–824, 2007.

[Pu08] Puleston, Colin; Parsia, Bijan; Cunningham, James; Rector, Alan: The Semantic Web - ISWC 2008: 7th International Semantic Web Conference, ISWC 2008, Karlsruhe, Germany, October 26-30, 2008. Proceedings. Springer Berlin Heidelberg, Berlin, Heidelberg, chapter Integrating Object-Oriented and Ontological Representations: A Case Study in Java and OWL, pp. 130–145, 2008.

[SBF98] Studer, Rudi; Benjamins, V.Richard; Fensel, Dieter: Knowledge engineering: Principles and methods. Data and Knowledge Engineering, 25(1-2):161 – 197, 1998.

[We16] Weber, Florian: Semantische Objektorientierte Programmierung. Bachelor thesis, Karlsruhe Institute of Technology, Germany, 2016. http://florianjw.de/ba/soop.pdf.