# Generating Transparent and Query-Based RDF Layers

Nils Rollshausen[1], Eduard Kamburjan[2] and Martin Giese[2]

[1]Technical University of Darmstadt, Germany
[2]University of Oslo, Norway

**Abstract**
We present `inkblot`, a software tool that generates object-oriented API code to represent the data found in a semantic model in a way suitable for programmers not familiar with semantic technologies. The API interacts with an RDF triple store via a SPARQL endpoint, but hides this connection from the programmer. The approach generates adequately typed fields and methods, based on the types and cardinalities in the data. Unlike previous work, the generation is not driven by an OWL ontology or similar declarative description of the application domain, but by a SPARQL query that accesses the pertinent data.

**Keywords**
Code generation, Object-orientation, Graph query

## 1. Introduction

**Motivation.** Object-oriented (OO) software development strives to produce designs containing classes with fields and relationships corresponding to entities in the application domain. The same holds true for semantic technologies, where significant effort is invested to produce ontologies that accurately formalize a conceptualization of the domain.

It is thus inevitable when interfacing between application software and semantic technologies that domain concepts will be represented twice, once in the form of types in the Resource Description Framework (RDF) sense, and once in an object-oriented class hierarchy. Moreover, boilerplate code needs to be written that transfers information between the two representations. The need for these manually created and maintained abstraction layers is a source of errors and presents a significant barrier for adoption of semantic web technologies.

This situation naturally leads to the idea of automatically generating program code that fits a semantic model in the form of a library or API that exposes a class hierarchy that is consistent with a semantic model of the domain, and that incorporates services such as synchronization with a triple store, reasoning, etc. Due to the wealth of research on ontology languages like OWL as a means of expressing domain knowledge, a variety of approaches have been proposed to generate such code from an ontology [1, 2, 3, 4]. However, we believe that there are some fundamental difficulties when using an ontology for this purpose.

First, the ontology describes the domain, and not the information available. For example, an ontology about human beings may rightly express that every human has a mother. Translating this to a requirement for the OO code stating that the 'mother' field may never be null is

*SofLiM4KG'24: Software Lifecycle Management for Knowledge Graphs Workshop, November 11–12, 2024, Baltimore, US*
nrollshausen@seemoo.tu-darmstadt.de (N. Rollshausen); eduard@ifi.uio.no (E. Kamburjan); martingi@ifi.uio.no (M. Giese)

© 2024 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

nonsensical since any given dataset will not contain information about the mother of every human mentioned. Typing, cardinalities, and non-nullness in the code should be informed by what information will have to be represented, and not the realities of the domain as such. Second, the typing of relations in a semantic model does not fit that of associations in object orientation. E.g., domain statements are inherited upwards in the class hierarchy: if $R$ has domain $C$, then $R$ also has domain $C$' for any superclass of $C$. But if an OO class $C$ has a field $f$, then any *subclass* of $C$ will also have the field.

Both of these are instances of the so-called *impedance mismatch* [5, 6] or *semantic gap* [7, 8] between the object model of RDF, geared towards data-driven tasks, and the object model of programming languages, geared towards typeability and modularity.

**Approach.** We explore a different approach that uses a closed-world formalism to describe the shape of the information to be represented. Specifically, we require the user to formulate a SPARQL Protocol and RDF Query Language (SPARQL) query that loads the information required. Based on this query and some additional configuration information (field names, multiplicities, etc.), we generate API code that provides an object oriented representation of the information, as well as implementations to lazily load and store data from and to a SPARQL endpoint.

The core challenge here lies in automatically generating *writeback* SPARQL updates given a single retrieval query, including updates for object creation, modification, and deletion. An approach for seamless *read-only* integration of semantic web data into object-oriented languages in a query-based manner has already been described in [8].

Our proposed API includes a runtime that supports caching and handles consistency of the loaded objects. Furthermore, it includes an *extendable* monitoring mechanism that (a) validates assumptions on the RDF data using SHACL shapes or SPARQL validation queries, and (b) verifies assumptions on the program data that must hold before the writeback to the RDF store. All local changes are collected until a commit to improve performance, and the API works with every write-enabled SPARQL endpoint. The generated API also supports creating completely new objects, writing them into the store, and deleting existing objects from the RDF data.

**Contribution.** Our main contribution is a tool for query-based *two-way* integration of OO and RDF through API generation from SPARQL queries. The tool is available as open source at https://github.com/smolang/inkblot and has a modular backend that currently supports Kotlin code generation, which can be used from any JVM-based language.

## 2. Motivating Example

We first introduce an example to illustrate our notion of transparent and query-based RDF layers. Consider a software written in the domain of bicycles that is used to manage the inventory of a shop. The inventory is stored in an RDF store, and must be manipulated using software in an object-oriented language.

The software must manipulate RDF structures, but these structures are optimized for the domain and data storage, not necessarily the application. More precisely, the data loaded from

```
1  SELECT ?bike ?mfg ?fw ?bw ?bells {
2    ?bike a bk:bike; bk:hasFrame [bk:frontWheel ?fw; bk:backWheel ?bw]
3    OPTIONAL { ?bike bk:mfgYear ?mfg }
4    OPTIONAL { ?bike bk:hasFrame [bk:hasBell ?bells] } }
5  SELECT ?wheel ?diameter {
6    ?wheel a bk:wheel; bk:diameter ?diameter. }
7  SELECT ?bell ?color WHERE {
8    ?bell a bk:bell;
9    bk:color ?color }
```

Listing 1: SPARQL retrieval queries for bike, wheel and bell objects.

and written to the RDF store must be transformed into OO structures. Take, for example, the three SPARQL queries in Lst. 1 that load all bikes, wheels, and bells from the RDF store.

The first query loads all bikes, their wheels, and manufacturing dates. Note that there are bike *frames* in the RDF store, to which the wheels are attached. However, they are *not loaded* because they are not manipulated by the program, yet they are part of the ontology. Management applications do not need to consider the semantic connotations of data, and software developers prefer to manipulate the objects in terms of their programming language (except during I/O). For example, the code in Lst. 2 implements two business logic operations: The first loads all bikes lacking a manufacturing date and sets their date to the current year. The second creates a new bicycle and equips it with a bell removed from an existing bike.

All SPARQL queries in this example are hidden in the special classes `Bike`, `Wheel`, and `Bell` and their associated `Factory` classes that manage the I/O of data – only when we have to provide a filter to extend the load query (l. 4), semantic technologies are accessed directly.

Our tool, `inkblot`, generates such code from the given SPARQL queries, provides basic operations on the loaded data, and offers a way to extend the generated structures with application specific code. It furthermore manages unambiguous writeback and consistency, provides an architecture to monitor its assumptions about the data at runtime, and supports lazy loading of resources handled through a pointer structure.

```
1  Inkblot.endpoint = "http://example.com/sparql"
2
3  // add manufacturing year where unset
4  val undated = BikeFactory.commitAndLoadSelected("!bound(?mfg)")
5  undated.forEach{ cycle -> cycle.mfgYear = 2023 }
6
7  // create a tricycle and transfer an existing bell to it
8  val scrapBike = BikeFactory.loadFromUri("...")
9  val bell = scrapBike.bells[0]
10 scrapBike.bells_remove(bell)
11 val wheel1 = WheelFactory.create(diameter=20.0)
12 [...]
13 val newCycle = BikeFactory.create(wheel1, wheel2, 2023, listOf(bell))
14 Inkblot.commit() // commit all changes
```

Listing 2: Usage examples of the generated library classes

## 3. Architecture

The workflow of `inkblot`, pictured in Fig. 1, has three steps that transform a set of SPARQL queries or configuration files into a set of classes in the target language, together with some auxiliary structures to use at runtime.

**Input** The input to `inkblot` is a set of SPARQL queries. These queries are then translated into a configuration file in which the user can refine the generated abstraction of the class: its fields, its connection to other classes, and the multiplicity of fields.

**Generation** The library generation itself analyzes the input query (*Variable Path Analysis*) and synthesizes several other queries for manipulating the RDF related to an object (*Query Synthesizer*). The user can override some of the generated queries, especially if different deletion behavior is required.

Based on analysis results and synthesized queries, the actual generation generates code in the target language (*Semantic Object Generator*). To ensure that runtime data indeed follows the assumptions used during generation (e.g., multiplicity of fields), SPARQL queries and SHACL constraints for monitoring are generated (*Validation Generator* and *SHACL Generator*). The SPARQL queries are included in the generated code, while the SHACL constraints are intended for external monitoring.

**Output** The generated RDF layer consists of the `inkblot` runtime in the target language, and a class and class factory for each query, as well as optional wrapper classes that can be used in the type hierarchy of the application.

We will continue to go over the components of `inkblot`, as well as its input, in detail.

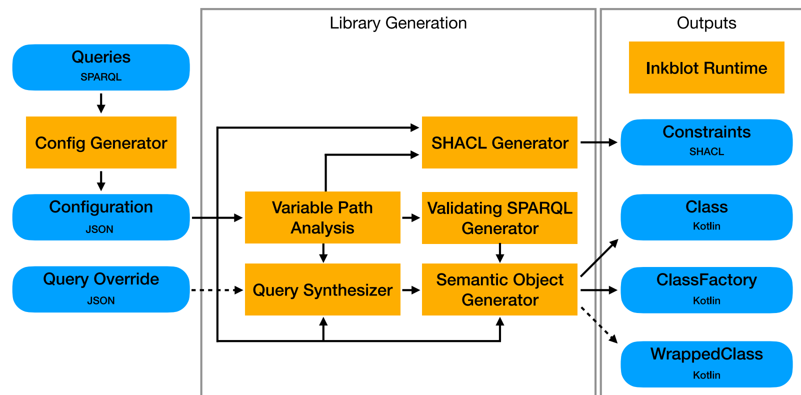


**Figure 1:** Workflow of `inkblot`. Rectangles are `inkblot` components, rounded rectangles are input/output files. Dashed arrows indicate optional inputs/outputs.

### 3.1. Input

The input to `inkblot` is a set of SPARQL queries. A configuration record is generated for each query. The configuration record for the query for wheels is given in Lst. 3. Each record has a

name and contains (a) the original query, (b) the anchor variable, (c) the RDF type of the anchor variable, and (d) a list of property records. The anchor variable is used to load the node in the RDF graph that is loaded into the object, the so-called anchor node. Two objects are considered to model the same structure if they have the same anchor node.

Each property record has a name and contains (a) a SPARQL variable name from the query, (b) a type, and (c) a cardinality. The SPARQL query must contain a path from the anchor variable to each variable used in a property record, but parts of the path may be optional. The type of a property is either an XSD datatype, the name of a property record, or the constant `inkblot:rawObjectReference`. The cardinality is either optional `"?"`, solitary `"!"` or many `"*"`.

Each property record becomes a field of the class generated for the query, with the name of the record being the name of the field, loaded from query results using the contained SPARQL variable. The type decides the type of the field – an XSD datatype becomes a primitive type, another record name becomes a reference to that class, and anything else is handled as a URI. Cardinality decides whether the field can have a `null` value (for `?`) or is a list (for `*`).

In the default, unedited configuration, the most generic cardinality is used and the name of each property is set to the name of the SPARQL variable. A property record is generated for every variable in the result set of the input query.

```
1  "Wheel": {
2    "anchor": "wheel",
3    "type": "http://example.com/ns/class",
4    "query": "... SELECT ?wheel ...",
5    "properties": {
6      "diameter": {
7        "sparql": "diameter",
8        "type": "[classnameOrXSD]",
9        "cardinality": "*"
10     }
11   }
12 }
```

Listing 3: Excerpt of the default JSON configuration for wheels.

### 3.2. Generation

The configuration file is then processed by `inkblot` to generate the program code. The generated code provides access to the fields defined for the class but also generates further queries and structures for writeback, consistency, and validation.

**Analysis.** As a preprocessing step to synthesize queries that relay changes between the RDF store and the program, we need to analyze variable dependencies in the input. The original query, which we denote *retrieval query*, must have explicit paths from the anchor variable to any variable used in a record.

The `inkblot` *Variable Path Analysis* component traverses the retrieval query and collects all triple patterns contained in it, keeping track of potential optional contexts. The analysis itself ensures that the query conforms to the limitations outlined below and builds a *dependency graph* in which nodes are SPARQL variables, constant URIs or values, and edges are the RDF

predicates that are used to relate them to one another in the input query. It then computes the set of all *simple paths* leading from the *anchor variable* to any variable (including those not in the result set) or literal and verifies that the graph is connected. Paths traversing the same nodes using different edges are considered to be distinct. The additional non-result variables and literals are used later to update the RDF graph correctly.

**Synthesis.** The *QuerySynthesizer* component uses the results of the path analysis to create additional SPARQL queries to update the RDF store. We distinguish between initialization, addition, removal, change, and deletion updates. All of these updates can operate on either literal values or object references without any changes. We will therefore only refer to general *values* in the following, which can either be literal values or URIs representing object references.

To create an update that adds a new value to a property, for example, we first restrict the dependency graph to only edges that are *safe* (i.e. not optional) in the context of the variable representing the property. If that variable is defined inside of an *Optional* block, this includes all edges found in the same block or any parent blocks. We then build a sub-graph by taking the neighbourhood of the target variable and recursively extending it by the neighbourhood of any non-anchor variables contained in it. All edges in this sub-graph are translated into triples and form the basis of the generated update, substituting the object's URI for the anchor variable and the value to be added for the target variable. Triples that are known to exist already, even if the property in question is not set yet, are included as the *where* condition of the SPARQL update while all other triples are inserted.

To remove a value, we follow a similar procedure and build the same neighbourhood sub-graph. We include the full sub-graph in the *where* condition but only delete triples that form the last edge on a simple path from the anchor variable to the target variable. This ensures that any annotations belonging to the removed value and used in the retrieval query for filtering purposes stay intact.

A *change* update is equivalent to a *removal* update for the old value followed by an *addition* update to set the new value. Together, these three types of updates cover everything needed to modify the properties of our generated classes. To create and delete entire objects themselves, we need additional initialization and deletion updates.

For object creation, we distinguish between the *base creation update* and additional *initializer* updates. The base creation updates insert all triples corresponding to safe edges in the dependency graph. Since all variables contained in this subgraph are strictly required, we can bind them to matching required constructor arguments. For constructor arguments corresponding to optional properties, we use initializer updates that are only executed if a non-null / non-empty constructor argument is passed. Initializer updates are identical to addition updates but are considered distinct to allow overwriting one but not the other.

For object deletion, we delete all triples that contain the anchor node of the deleted object in either subject or object position. Other nodes are not deleted recursively to avoid unintentional loss of data that may be required by other applications.[1]

[1]In terms of our ongoing example, while our application may reasonably delete *frame* nodes when deleting a bike, other applications with a different view of the data may expect the frame to continue to exist even if the bike itself is deleted and used for spare parts.

These updates, particularly the deletion update we have just discussed, assume specific semantics for their operations. As these semantics may be application-specific, the user has the option to provide their own update queries and override the ones generated by `inkblot`. In this way, users can make adaptations to their specific application while still benefiting from fully automatic code generation and the same level of runtime support as with the default updates.

**Generation.** Equipped with the synthesized SPARQL updates, the actual *Semantic Object Generator* generates three classes per configuration record: a core class, a factory for this class, and a wrapper class for the core class, which is a default implementation to add business logic around the core class. We will describe them in detail in the next section. The factory class uses slightly modified versions of the retrieval query to load individual instances or subsets of all instances. For access to the defined properties, we generate standard getter and setter methods that operate on locally cached property values. For references to other objects, we only store the URI of the referenced object — the corresponding getter method transparently lazy-loads and instantiates the referenced object on access. The details of lazy loading are described by Kamburjan et al. [8]. All setter methods record changes made to the property using the previously generated SPARQL updates in addition to updating the cached property value. This creates the local changelog of SPARQL updates that will be applied to the data store when local changes are committed.

The generated code is statically typed, using the type and cardinality information contained in the configuration file. Boolean values, strings, and all numeric xsd types are supported natively and mapped to appropriate Kotlin type equivalents. Other xsd types, such as dates, are exposed to the user in their string representation. The communication with the RDF store over the SPARQL endpoint is implemented using Apache Jena[2].

**Validation.** The annotated type and cardinality information is used to generate well-typed code. However, this corresponds to assumptions about the actual data in the RDF store, which may not hold at runtime. We therefore provide two mechanisms to validate the used data: For one, `inkblot` generates SPARQL queries selecting instances with invalid types or cardinalities using variations of the retrieval query with different added filter expressions. These validating queries are embedded into the generated Factory classes and automatically validate the consistency of the data store before any objects are loaded or created. For another, `inkblot` automatically generates SHACL constraints for all classes, encoding all assumptions made in the configuration. The user can manually check these constraints using any SHACL validator.

**Language portability** Currently, only Kotlin is supported as a target language, as it can use the numerous Java libraries for RDF and be used from other JVM-based languages, but the generation backend is designed modularly and can be adapted to other languages. Only the runtime and the semantic object generator components are specific to the target language; all other components are language-independent and can be reused for different backends. New backends can be implemented as extensions of the `AbstractSemanticObjectGenerator` class and can be selected by the user during generation.

[2]In our current Kotlin-backend, there are no Jena-specific design decisions in `inkblot`.

**Limitations.** In terms of the supported SPARQL fragment, `inkblot` supports queries that provide *unambiguous writeback locations* for all variables to allow persisting changes to the data store. This means that each variable may only be bound in a single location and that the provided graph patterns form connected graphs around the anchor node. Language features such as subqueries, explicit variable bindings, and set difference operations also do not allow unambiguous writeback and are not currently supported.

## 4. Generated API

The output of `inkblot` is a set of classes to be used together with the `inkblot` runtime. The API is documented in greater detail at https://github.com/smolang/inkblot; we present only its main capabilities here.

### 4.1. Runtime

The runtime of `inkblot` maintains a global cache of all loaded objects to ensure that each anchor node is represented only by a single object at a given time and local changes remain consistent. The cache is transparent to the user.

Additionally, it maintains a log of all changes to loaded objects and the SPARQL updates required to persist these changes to the data store. When the user decides to commit these local outstanding changes, the gathered updates are batched together and sent to the SPARQL endpoint. To avoid unexpected behaviour when loading new instances from the data store in the presence of uncommitted local changes, all methods that load multiple instances also commit changes to the data store. The API assumes that no other party is modifying the RDF data.

The API of the runtime permits the following operations to the user:

**Loading** Using the `endpoint` property, the URL of the used SPARQL endpoint for the RDF connection can be set.

**Writing** Using the `commit()` method, all changes in the loaded objects are forced to be mirrored in the RDF store. This method is also called by several of the generated classes, which we detail below.

**Monitoring** The runtime monitors the consistency of local data in addition to the previously described SPARQL and SHACL validation mechanisms for online data. This includes checks for data types that do not have a one-to-one mapping to JVM primitive types, such as the XSD *NegativeInteger*, verifying that values assigned to such properties conform to the limitations of the underlying XSD type. If any such check fails, listeners added/removed using the `addViolationListener()`/`removeViolationListener()` methods are notified of this constraint violation, even before any changes are committed.

The runtime also implements lazy loading: whenever an anchor node is loaded, it is first checked whether the runtime has an object for it in the cache. Similarly, when a loaded object refers to another object, the second object is only loaded when the access actually happens. If

the object is never accessed, it is not loaded. For example, if the program loads a bike but never accesses its wheels, then the wheels are not loaded at all.

## 4.2. Generated Classes

For each configuration record *C*, three classes are generated, as pictured in Fig. 2. The core class

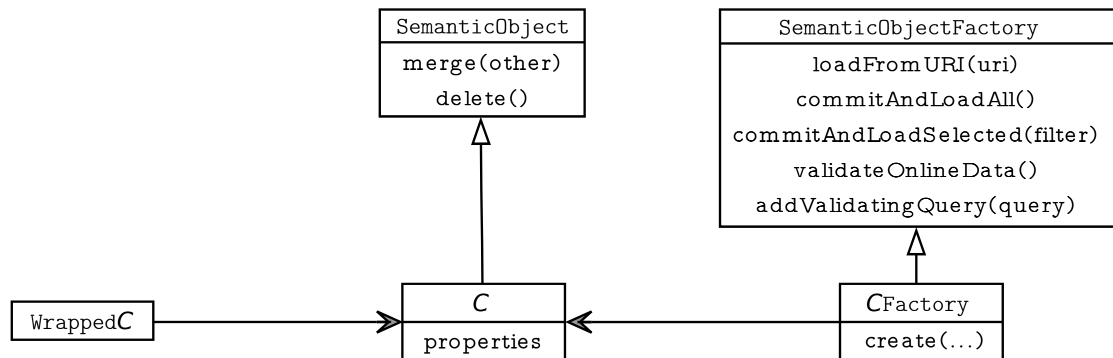


**Figure 2:** UML diagram of the generated classes for a configuration record *C*.

contains all the data, operations for changes, and manages the relation of the object to the RDF store. The factory is used to create new core class objects either from scratch or by loading them from the store. A wrapper class is available to include core classes in the type hierarchy of the software application.

**Core Class.** A core class object provides read access to an object's unique URI and full read/write access to all properties defined in the configuration record. Using Kotlin's built-in support for property getter and setter methods, we render functional properties as first-class Kotlin properties, making changelogging, lazy loading of object references and runtime constraint checks completely transparent to the user. We also use Kotlin's distinction between nullable and non-nullable types to offload enforcement of 'exactly one' cardinality constraints to the type system. For non-functional properties that can have multiple values, we still provide access using first-class language features in the form of an immutable Kotlin list. Addition and removal of list entries is handled using dedicated methods.

Inspired by the OpenCitations `ocdm` API [9], we also provide a *merge* operation that combines two instances of the same class and redirects all references to either instance to the new unified instance. All core classes implement the `SemanticObject` interface.

**Factory.** For each configuration record, `inkblot` generates the class itself as well as an associated *Factory* class used for creating new instances (`create()`) as well as loading existing instances from the data store. The factory provides methods to load all instances found in the data store (`commitAndLoadAll()`), a single instance specified by a URI (`loadFromURI()`), or a subset of instances specified by a SPARQL filter expression (`commitAndLoadSelected()`).

```
"Bike": {
  "anchor": "bike",
  "type": "http://example.com/ns/Bike",
  "query": "... SELECT ?bike ...",
  "properties": {
    "frontWheel": {"sparql": "fw", "type": "Wheel", "cardinality": "!" }
    "backWheel": {"sparql": "bw", "type": "Wheel", "cardinality": "!"  }
    "bells": {"sparql": "bell", "type": "Bell", "cardinality": "*" }
    "mfgYear": {"sparql": "mfg", "type": "xsd:int", "cardinality": "?"  }
  }
}
```

```
sh:targetClass <http://example.com/ns/Bike>;
sh:property [
 sh:datatype xsd:int;
 sh:path <http://example.com/ns/bike#mfgDate>
 sh:maxCount 1;
 sh:name "..."; sh:message "...";
].
```

Listing 4: JSON configuration for class 'bike' and an example SHACL shape.

These filter expressions are the only place where the user is exposed to the underlying semantic technology.

The factory also allows users to perform validation of instances in the remote data store at runtime using `validateOnlineData()`, optionally using custom validation queries (added by `addValidatingQuery()`) in addition to automatically generated ones. Contrary to the monitoring of the runtime, which is performed on the local objects, these checks are performed on the triplestore. The generated validating SPARQL queries, for example, are used here.

**Wrapper.** To allow users to extend generated classes with custom application logic, inkblot can optionally create wrapper classes that contain a reference to a class instance and expose the same set of properties lifted from that instance without containing any logic on their own.

## 5. Usage

Let us illustrate the overall workflow that connects the queries and the program code in Sec. 2 and follow the query for bike throughout generation. The generated default configuration record is analogous to the one given for wheels in Sec. 3. After modification to fit our intents, the configuration record is as shown in Lst. 4. The RDF type of the anchor variable is updated (l. 3), the names of the properties are different from the SPARQL variables used to retrieve them (l. 6,7,8,9) and the cardinality is changed. Every bike has exactly one front wheel and one back wheel (l. 6, l. 7), at most one manufacturing year (l. 9) and arbitrarily many bells (l. 8). The types of the properties are updated to correspond to an XSD datatype or the name of another configuration record.

The following artifacts are then generated. First, the SHACL shapes, for example the shape in Lst. 4 checking the cardinality and type of the manufacturing date. Second, the core class `Bike`, given in Lst. 5. As we can see, it has a property for each variable defined in its configuration record. The `bells` list variable is not exposed, but an API is generated so users cannot manipulate

internals. Also note the cached loading of front wheels at l. 8 – the front wheel is only loaded via the cache once accessed. Lst. 6 describes the full details of property access and modification for `mfgYear`. While the loading is straightforward, changing the value requires generating updates depending on whether the value is set to `null` and is deleted, set to a value for the first time, or changed to a new value. These queries are added to the global changelog and only executed when the changelog and cache are committed. Until then, the database is not updated.

```
1  class Bike internal constructor(uri: String, frontWheel: String,
2                        backWheel: String, bells: List<String>,
3                        mfgYear: Int?) : SemanticObject(uri) {
4  [...]
5  // frontWheel
6  private var _inkbltRef_frontWheel: String = frontWheel
7  var frontWheel: Wheel
8     get() = Wheel.loadFromURI(_inkbltRef_frontWheel)
9     set(value) {...}
10 // backWheel
11 private var _inkbltRef_backWheel: String = backWheel
12 var backWheel: Wheel
13    get() = Wheel.loadFromURI(_inkbltRef_backWheel)
14    set(value) {...}
15 // bells, extra operations instead of setter
16 private val _inkbltRef_bells = bells.toMutableSet()
17 val bells: List<Bell>
18    get() = _inkbltRef_bells.map { BellFactory.loadFromURI(it) }
19 fun bells_add(obj: Bell) {...}
20 fun bells_remove(obj: Bell) {...}
21 // manufacturing year
22 var mfgYear: Int? = mfgYear
23    set(value) {...}
24 // special merge operation
25 fun merge(other: Bike) {...}
26 }
```

Listing 5: Excerpt of core class ‘bike’

The `BikeFactory` handles creation of `Bike` objects and the connection to the RDF store. Most of the interface is inherited from the generic `SemanticObjectFactory`, only the method to create a `Bike` from scratch (`create()`) is added, as well as an internal helper function to create a `Bike` from a query result.

As the `Bike` class is part of the `inkblot` type hierarchy and contains all the logic for managing RDF references and lazy loading, `inkblot` can optionally generate a wrapper/interface class `WrappedBike` (Lst. 7) that has a `Bike` as a field and provides quick access to its fields as well as a space to add custom application logic.

## 6. Related Work

There are many existing works investigating the static generation of library code for RDF data access. A comprehensive comparison of existing approaches can be found in [7], alongside a detailed discussion of the semantic gap. In terms of Baset and Stoffel’s taxonomy, inkblot is an Active/Static approach that uses SPARQL as the source language.

```
var mfgYear: Int? = mfgYear
set(value) {
 if(deleted)
  throw Exception("...")
 if(value == null) { // Unset value
  val oldValueNode = ResourceFactory.createTypedLiteral(field).asNode()
  val cn = CommonPropertyRemove(uri, "/*mfgYear*/", oldValueNode)
  Inkblot.changelog.add(cn)
 }
 else if(field == null) { // Pure insertion
  val newValueNode = ResourceFactory.createTypedLiteral(value).asNode()
  val cn = CommonPropertyAdd(uri, "/*mfgYear*/", newValueNode)
  Inkblot.changelog.add(cn)
 }
 else { // Change value
  val oldValueNode = ResourceFactory.createTypedLiteral(field).asNode()
  val newValueNode = ResourceFactory.createTypedLiteral(value).asNode()
  val cn = CommonPropertyChange(uri, "/*mfgYear*/", oldValueNode, newValueNode!!)
  Inkblot.changelog.add(cn)
 }

 field = value
 markDirty()
}
```

Listing 6: Excerpt of managing a property in the core class 'bike'

```
object BikeFactory : SemanticObjectFactory<Bike>(listOf(/*queries*/)) {
fun create(frontWheel: Wheel, backWheel: Wheel,
        bells: List<Bell>, mfgYear: Int?):Bike {...}
override fun instantiateSingleResult(lines:List<Solution>):Bike?{...}
}
```

```
class WrappedBike(private val bike: Bike) {
var frontWheel: Wheel
 get() = bike.frontWheel
 set(value) { bike.frontWheel = value }

var backWheel: Wheel
 get() = bike.backWheel
 set(value) { bike.backWheel = value }

val bells: List<Bell>
 get() = bike.bells
fun bells_add(entry: Bell) = bike.bells_add(entry)
fun bells_remove(entry: Bell) = bike.bells_remove(entry)
...
fun delete() = bike.delete()
fun merge(other: WrappedBike) = bike.merge(other.bike)
}
```

Listing 7: Interface of the generated factory for bikes, and the interface class.

Most of the existing approaches use OWL ontologies as their input [1, 2, 3, 4]. All of these works face the issue of the semantic gap between RDF and their respective target languages and address them to varying degrees.

Among these, *Sapphire* [4] stands out as significantly narrowing the semantic gap by sup-

porting open-world reasoning and using JVM bytecode manipulation to allow re-typing objects at runtime, matching RDF semantics.

Also *owlready* [3] includes extensive reasoning support, and addresses the gap by combining open- and closed-world reasoning. It provides a sophisticated API for accessing and modifying OWL ontologies in Python. Owlready redefines several Python methods and makes use of the language's dynamic nature to approximate RDF semantics. Other approaches for integrating RDF with dynamic languages exist [10, 11], but are less relevant to our use case of generating statically typed Kotlin/JVM code.

There are also several open-source tools to transform OWL to Java, implementing the ideas found in [2] and other works. Two of these are *Jastor*[3] and *owl2java*[4], both of which generate APIs that are structurally close to typical inkblot output. The *Protégé* ontology editor also provides a Java code generation plugin using a similar OWL-to-Java mapping[5].

Our approach is not the first to generate API code from SPARQL queries: the tool `grlc` [12] takes the same route. However, `grlc` does not connect to a programming language directly, but merely provides web developers with a more familiar way to access semantic data without hiding much of the underlying complexity.

Scheglmann et al. [13] identify the need for more abstraction and customization in generated APIs. To this end, they propose a method that takes OWL ontologies as an input and transforms them into two intermediate models in custom modeling languages. These models are edited by the user to customize the Java API that is generated from the model in the final translation step.

Parreiras et al. [14] address similar issues of customization and portability with *Agogo*. At its core, Agogo is a domain-specific language (DSL) for describing mappings of semantic concepts to object-oriented APIs. Similarly to the intermediate models in [13], it can then be used to generate API code in a given target language. Agogo is also particularly interesting in that it allows users to define their own SPARQL queries to read and update properties, making it *query-based* in a way that is similar to our approach.

Compared to these approaches, inkblot offers similar possibilities for abstraction and customization but makes different trade-offs, favouring a higher degree of automation and standard formats over unlimited customization.

Another approach that could be described as *query-based* is LITEQ [15], which provides a custom query language that can be used to load and create semantic data dynamically using existing RDFS annotations. From a usage perspective, however, LITEQ appears closer to libraries like Jena than the other presented approaches: while it allows for type-safe access to data, it does not provide the kind of fully encapsulated API typically generated by other tools.

Still, these last three approaches are closer in spirit to inkblot than purely OWL-based generation methods, as they allow for deviations between data access APIs and the underlying data model.

[3]https://jastor.sourceforge.net/
[4]https://github.com/piscisaureus/owl2java
[5]https://protegewiki.stanford.edu/wiki/Protege-OWL_Code_Generator

## 7. Conclusion

Semantic technologies are hard to use for non-experts, and this work describes a step toward enabling programming applications that use semantic data through an automatically generated API that almost completely hides the semantic technologies for the programmer. We envision that `inkblot` configurations are maintained by the developers of the semantic dataset or ontology and provided to external programmers. In this way, semantic web technologies are easier to integrate into bigger applications.

We focus on the use of SPARQL queries as the interface between OO and RDF, in contrast to approaches that focus on either generating OO code from an ontology or manipulating raw RDF data. We conjecture that this is a more elegant and practical solution to connecting the worlds of semantic data and programming than including concepts from one side into the other.

**Future Work.** For the next step, we plan to implement a Python backend. As for usability, we plan to investigate using a DSL as part of the runtime to hide the SPARQL filter expressions used currently, and the ability to define subtypes over configurations based on the Liskov principle for RDF data loading [8]. Furthermore, we assume that it is possible to simplify the local change log, similar to the `ocdm` API [9], for performance.


### Acknowledgments

This work was partially supported by the SM4RTENANCE project, an EU H2020 project under grant agreement No. 101123423.


## References


[1] N. M. Goldman, Ontology-oriented programming: Static typing for the inconsistent programmer, in: ISWC, volume 2870 of *Lecture Notes in Computer Science*, Springer, 2003, pp. 850–865.

[2] A. Kalyanpur, D. J. Pastor, S. Battle, J. A. Padget, Automatic mapping of OWL ontologies into Java, in: SEKE, 2004, pp. 98–103.

[3] J.-B. Lamy, Owlready: Ontology-oriented programming in python with automatic classification and high level constructs for biomedical ontologies, Artificial Intelligence in Medicine 80 (2017) 11–28. URL: https://www.sciencedirect.com/science/article/pii/S0933365717300271. doi:https://doi.org/10.1016/j.artmed.2017.07.002.

[4] G. Stevenson, S. Dobson, Sapphire: Generating Java runtime artefacts from OWL ontologies, in: CAiSE Workshops, volume 83 of *Lecture Notes in Business Information Processing*, Springer, 2011, pp. 425–436.

[5] G. P. Copeland, D. Maier, Making Smalltalk a database system, in: B. Yormark (Ed.), SIGMOD, ACM Press, 1984, pp. 316–325. URL: https://doi.org/10.1145/602259.602300. doi:10.1145/602259.602300.

[6] V. Eisenberg, Y. Kanza, Ruby on semantic web, in: ICDE, IEEE Computer Society, 2011, pp. 1324–1327. URL: https://doi.org/10.1109/ICDE.2011.5767945. doi:10.1109/ICDE.2011.5767945.

[7] S. Baset, K. Stoffel, Object-oriented modeling with ontologies around: A survey of existing approaches, Int. J. Softw. Eng. Knowl. Eng. 28 (2018) 1775–1794.
[8] E. Kamburjan, V. N. Klungre, M. Giese, Never mind the semantic gap: Modular, lazy and safe loading of RDF data, in: ESWC, volume 13261 of *Lecture Notes in Computer Science*, Springer, 2022, pp. 200–216.
[9] S. Persiani, M. Daquino, S. Peroni, A programming interface for creating data according to the SPAR ontologies and the OpenCitations data model, in: ESWC, volume 13261 of *Lecture Notes in Computer Science*, Springer, 2022, pp. 305–322.
[10] M. Babik, L. Hluchy, Deep integration of Python with Web Ontology Language, in: C. Bizer, S. Auer, L. Miller (Eds.), 2nd Intl. Workshop on Scripting for the Semantic Web, volume 181 of *CEUR Workshop Proceedings*, 2006. URL: http://CEUR-WS.org/Vol-181/paper1.pdf.
[11] E. Oren, R. Delbru, S. Gerke, A. Haller, S. Decker, ActiveRDF: object-oriented semantic web programming, in: WWW, ACM, 2007, pp. 817–824.
[12] A. Meroño-Peñuela, R. Hoekstra, grlc makes GitHub taste like linked data APIs, in: SALAD@ESWC, volume 1629 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2016.
[13] S. Scheglmann, A. Scherp, S. Staab, Declarative representation of programming access to ontologies, in: ESWC, volume 7295 of *Lecture Notes in Computer Science*, Springer, 2012, pp. 659–673.
[14] F. S. Parreiras, C. Saathoff, T. Walter, T. Franz, S. Staab, APIs à gogo: Automatic generation of ontology APIs, in: ICSC, IEEE Computer Society, 2009, pp. 342–348.
[15] M. Leinberger, S. Scheglmann, R. Lämmel, S. Staab, M. Thimm, E. Viegas, Semantic web application development with LITEQ, in: ISWC (2), volume 8797 of *Lecture Notes in Computer Science*, Springer, 2014, pp. 212–227.