Kala Technical Report

Sven de Ridder Flavius Frasincar

Erasmus University Rotterdam
PO Box 1738, NL-3000 DR
Rotterdam, the Netherlands

In this report, we describe *Kala*[1], our proof-of-concept implementation of the proposed temporal conceptual model. In Section 1, we outline the technology choices for the implementation. Section 2 describes our methods of extending the OWL API for Kala. The implementations of the constucts that Kala exposes are outlined in Section 3. Lastly, we describe the operations of querying and manipulation of the temporal conceptual model in Section 4 and parsing and serialization of the supported representation schemes in Section 5.

1 Technology choices

The implementation of Kala depends on a number of technology choices. Table 1 lists these technologies, as well as their versions used for implementation and a brief description of the purpose for which the technologies were adopted. In the following sections, we describe the technology choices in detail and provide rationales.

Technology	Version	Purpose
Java	1.6-1.7	Application platform and programming language.
OWL API	3.4.3	Interface for OWL ontologies.
Google Guava	14.0-RC2	Advanced support for collections.
Joda Time	2.2	Advanced support for time types.

Table 1: Technology choices

1.1 Java

Java[2] is a combination of application platform and programming language that allows the development, deployment, and execution of software appli-

[1]Kālá is the Sanskrit word for *time*.
[2]`http://www.java.com/`

cations in a cross-platform computing environment. Our choice of Java as a target platform for Kala is motivated through a number of strengths.

Java's relative power and simplicity, combined with its cross-platform nature, have ensured that the language enjoys a broad community of practitioners, and, perhaps because of its prevalence in university-level programming courses, it is a particularly popular language in academic research and development. The language supports the *object-oriented programming* paradigm, similar to C++, and features a very convenient linking mechanism for modules, both features facilitating the development of large-scale applications. Finally, the choice for Java enables the use of libraries such as the OWL API (see below), and a later integration in frameworks such as Protégé-4 [7][3], a mature and widely-used ontology editor.

1.2 The OWL API

The OWL API [6][4] provides a high level Java API for working with OWL ontologies. It has been used in the development of a number of important projects, including Protégé-4, and the Pellet [12][5] and HermiT [10][6] reasoners.

The OWL API provides an *axiom-centric* abstraction of ontologies: an ontology is viewed as a collection of axioms and annotations, and this results in a close correspondence between constructs exposed by the OWL API and the OWL 2 specification [8]. This is in contrast to APIs such as Jena [2][7] that treat axioms at a lower level, that of RDF triples. In this way, developers can generally utilize the OWL API without concern for issues related to representation, in particular those related to the parsing and serialization of data structures, and working with OWL ontologies at a convenient level of abstraction.

Furthermore, the OWL API is designed for extension: it provides a useful separation between interfaces and class definitions, which encourages the development of custom implementations. Such development is further encouraged through the use of *dependency injection*, which allows the default implementation to interact with any customized components as expected. The design clearly emphasizes separation of responsibilities between the various components. Querying is straight-forward, but manipulation must be done through explicit change operations, which helps centralize the messaging and observation functionality in contexts such as caching and multithreading.

The level of abstraction offered by the OWL API presents a useful guide-

[3]`http://protege.stanford.edu/`

[4]`http://owlapi.sourceforge.net/`

[5]`http://clarkparsia.com/pellet/`

[6]`http://www.hermit-reasoner.com/`

[7]`http://jena.apache.org/`

line for the introduction of new constructs and functionality that extend the OWL model, and the nature of its design enables Kala to extend it in a clear and consistent way, which we hope will aid in Kala's adoption by any experienced users of the OWL API that wish to work with temporal ontologies.

1.3 Google Guava

Guava[8] (originally the *Google Collections Library*) is a set of open-source, common Java libraries developed and maintained by Google, and positions itself as an informal expansion on the Java Collections Framework, with a strong emphasis on *generic programming.* Internally, Kala makes use of Guava implementations for multimaps, bimaps, and variadic collection initializers.

1.4 Joda Time

Joda Time[9] is an open-source Java API for date and time handling. It has a number of advantages over the Java classes `java.util.Date` and `java.util.Calendar` that are included in the standard Java libraries: in particular, the library has very clear time concepts (e.g., allowing easy distinction between local times and times qualified with time zones), *value semantics* for time objects (i.e., a time object is an immutable object whose identity is based on its state), predictable performance characteristics, and a wide range of conversions between time zones, calendars, and representations. In contrast, the standard Java implementations have unclear semantics, and the mutability of their data structures and unpredictable performance can lead to obscure bugs. Because of the short-comings of the standard Java implementation, Joda Time lead to the Java SE 8 Date and Time API[10].

While we would eventually like to design Kala to allow the user to employ any genericized time type, the initial version of the library has been developed specifically around the `org.joda.time.DateTime` type.

2 Extending the OWL API

In order to guide the extension of the OWL API with temporal constructs, we pose two strong requirements:

1. The newly introduced temporal constructs must be completely separated from the constructs already supported by the OWL API in order to ensure the orthogonality of the non-temporal ontology and the temporal model; and

[8]`http://code.google.com/p/guava-libraries/`

[9]`http://joda-time.sourceforge.net/`

[10]`http://jcp.org/en/jsr/detail?id=310`

2. Kala must be compatible with custom implementations of the OWL API.

The orthogonality of the non-temporal components of the ontology and the temporal model is crucial for the correct functioning of the temporal model as an abstraction. We cannot, for example, decide that time instants and time intervals are subclasses of OWL individuals: time instants and time intervals are not necessarily represented as individuals in the representation scheme, and we need to restrict the operations that are permitted on these special entities in order to ensure that the temporal model will always have a correct mapping to the representation schemes. A statement such as, for example, "Bob is a friend of time instant t_1" does not make much sense in this context. Similar arguments hold for the fluent properties.

The compatibility with custom implementations of the OWL API is similarly crucial in the context of such developments as OWL database backends (for an examplar OWL database backend, see OWLDB [5]). In order to support custom implementations, we implement the extended functionality through the use of the *Decorator* design pattern [3]. A Decorator, essentially, wraps an object, but exposes an extended interface and implements the new functionality in terms of the wrapped object. Our application of the Decorator design pattern is described in more detail, below.

We implement the additional Kala functionality by providing Decorators for the following three OWL API interfaces:

The ontology. The OWL API views an `OWLOntology`, essentially, as a collection of `OWLAxioms` and `OWLAnnotations`. It provides methods to query these axioms and annotations, directly or through convenience methods, and collaborates with other objects to change the contents of this internal collection. Introducing a new category of axioms, the `TemporalAxioms`, then, necessitates extending `OWLOntology` in such a way that it can also store these new axioms, but without altering its existing behavior.

The data factory. The OWL API `OWLDataFactory` presents the interface for producing the entities, class expressions, and axioms that form the building blocks of the OWL ontology. The `OWLDataFactory` follows the *Factory* design pattern [3]. We extend `OWLDataFactory` to support the construction of the entities and axioms of the temporal conceptual model.

The ontology manager. The `OWLOntologyManager`, lastly, is responsible for the creation, loading, saving, and manipulation of ontologies. Since we need to create a new type of ontology, the temporal ontology, we need to extend the behavior of the `OWLOntologyManager` so that it can properly manage these temporal ontologies.

To construct our Decorators, we first declare *forwarding classes* [1, pages 81–86]: these are classes that simply wrap an object and implement the interface by forwarding any calls to the wrapped object. These objects can then be trivially extended to Decorators.

An example `TemporalDataFactory` Decorator for the existing interface `OWLDataFactory` is shown in Listing 1, along with its base class `ForwardingOWLDataFactory`. We only show the forwarding for the `getOWLClass()` method of the original `OWLDataFactory`, and the declaration and implementation of a newly introduced `getTimeInstant()` method. Similiar forwarding classes, Decorator interfaces, and Decorator implementations are provided in Kala for `TemporalOntology` and `TemporalOntologyManager`.

3 Temporal model constructs

The temporal model constructs that Kala exposes may be further subdivided into *temporal entities*, which extend the vocabulary of a temporal ontology, and *temporal axioms*, which express accepted truths about the temporal entities.

3.1 Temporal entities

The `TemporalEntity` represents the temporal model equivalent of the OWL entities, the named terms of an ontology [8]. It encompasses the temporal primitives, `TimeInstant` and `TimeInterval`, and the fluent properties, `FluentObjectProperty` and `FluentDataProperty` (see Figure 1). Temporal entities are uniquely identified by an IRI.

Functionality is added to the temporal entities through the *Visitor* design pattern [3], similarly to the OWL API. The Visitor pattern provides a *double-dispatch* mechanism, which allows a method call to be dispatched on the basis of the run-time type of two objects; in this case, the type of the temporal entity and the type of the specific Visitor that encapsulates the desired functionality. In this way, use of the Visitor pattern presents a clean separation of data structures and functionality. Examples of functionality that is provided through the use of the `TemporalEntityVisitor` interface are the insertion of temporal entity declarations (see below) into a temporal ontology, or the serialization of temporal entities to their equivalents in a particular representation scheme.

Temporal entities are produced through the `TemporalDataFactory` and support *value semantics*: if two separate temporal entities are produced, but they are of the same type and have the same IRI, the temporal entities are functionally identical. Furthermore, they are immutable. Value semantics allow the temporal entities to be used, for example, as proper keys in maps.

Listing 1: The `TemporalDataFactory` decorator

```java
// Reusable forwarding class for OWLDataFactory.
public class ForwardingOWLDataFactory implements OWLDataFactory {

  private final OWLDataFactory d;

  public ForwardingOWLDataFactory(OWLDataFactory d)
      { this.d = d; }

  public OWLClass getOWLClass(IRI iri)
      { return d.getOWLClass(iri); }

  // ... The rest of the forwarding methods go here.
}

// Interface for the TemporalDataFactory Decorator.
public interface TemporalDataFactory extends OWLDataFactory {
  // One of the new methods provided by TemporalDataFactory.
  public TimeInterval getTimeInterval(IRI iri);

  // ... Other method declarations go here.
}

// The TemporalDataFactory Decorator implementation.
public class TemporalDataFactoryImpl extends
    ForwardingOWLDataFactory implements TemporalDataFactory {
  public TemporalDataFactoryImpl(OWLDataFactory d)
      { super(d); }

  // Implementation of the new method.
  public TimeInterval getTimeInterval(IRI iri)
      { return new TimeIntervalImpl(iri); }

  // ... Other method implementations go here.
}
```

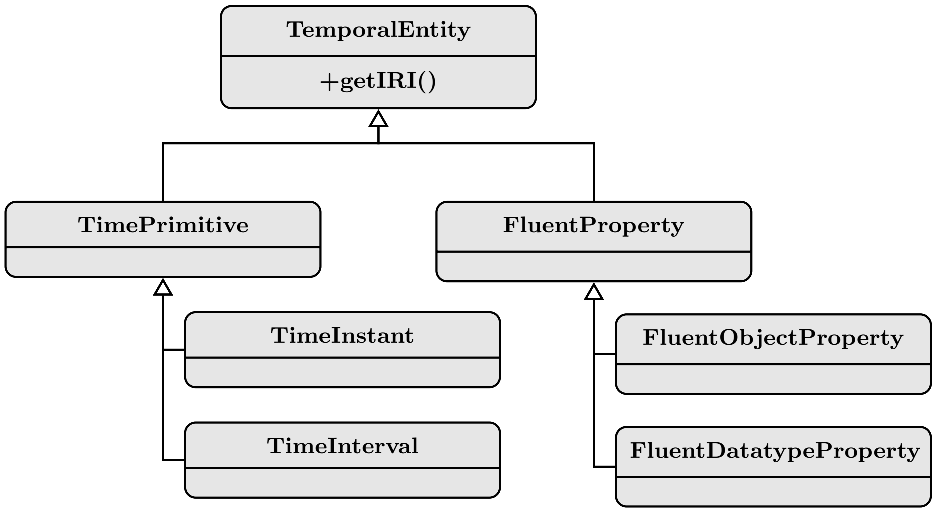

Figure 1: The temporal entities supported by Kala.

3.2 Temporal axioms

Kala presents the `TemporalAxiom` as the temporal equivalent of the OWL axioms, the statements of truth that compose an ontology [8]. Kala supports several temporal axioms (see Figure 2).

Like temporal entities (see Section 3.1), the classes representing temporal axioms are solely concerned with the representation of the data structure, and additional functionality is provided through the *Visitor* design pattern, through implementations of the `TemporalAxiomVisitor` interface. Also similarly to temporal entities, temporal axioms support *value semantics*, and must be constructed through the `TemporalDataFactory`.

3.2.1 Declaration axioms

The `TemporalDeclarationAxiom` is the Kala equivalent of the OWL declaration axiom. A declaration of a temporal entity E in a temporal ontology O accomplishes two things: it explicitly introduces E as part of the vocabulary of O, and it associates E with an entity type — E is established in O either as a time instant, a time interval, a fluent object property, or a fluent datatype property.

3.2.2 Time axioms

The relations of the time primitives to the time axis and to one another are expressed through the time axioms. Specifically, a discrete time may be assigned to a time instant through the `DiscreteTimeAxiom`, and intervals are defined through their endpoints using the `IntervalStartAxiom` and `IntervalEndAxiom`. The `InstantRelationAxiom` expresses ordinal

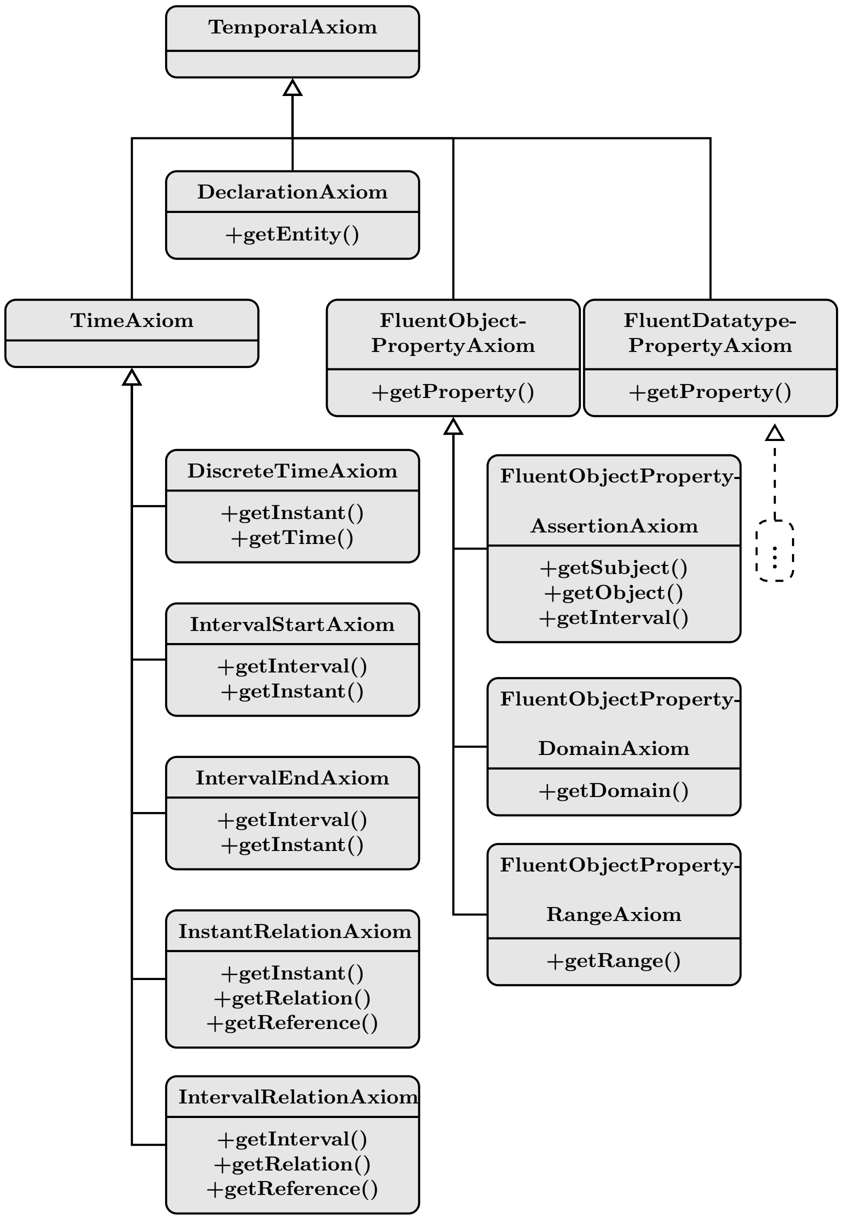

Figure 2: The temporal axioms supported by Kala. This diagram does not show the `FluentDatatypePropertyAxiom` hierarchy.

relations between time instants, and the `IntervalRelationAxiom` fulfills the same role for time intervals.

3.2.3 Fluent property domain and range restrictions

Kala allows the domains and ranges of fluent object properties and fluent datatype properties to be restricted in the same way as the equivalent OWL property restrictions. The domain and range of a fluent object property may be restricted to any class expression. Similarly, the domain of a fluent datatype property may be restricted to any class expression, and its range to any data range. As with OWL domain and range restrictions, multiple restrictions of the same type for the same fluent property are interpreted as representing the *intersection* of the restrictions.

3.2.4 Fluent property assertion axioms

Comparable to the OWL object property assertion axiom, the Kala temporal axiom type `FluentObjectPropertyAssertionAxiom` expresses that an individual (the subject) is connected to another individual (the object) by a fluent object property during a particular time interval.

The `FluentDatatypePropertyAssertionAxiom`, expresses that the value of a fluent datatype property for an individual is expressed by a specific literal during a particular time interval.

4 Querying and manipulation

The `TemporalOntology` object allows the user to retrieve the temporal axioms contained in the ontology, and accepts `TemporalAxiomVisitors` that may perform operations (such as rendering or tabulation) on the set of axioms. At the time of writing, however, Kala is still lacking many convenience methods, such as look-ups of property assertions by subject or property. The library is also lacking the equivalent of the OWL API `OWLReasoner` interface and ontology query answering components specifically suited for temporal ontologies. The implementation of these is currently beyond the scope of the Kala reference implementation, but if it proves amenable, a proper data manipulation language may later be developed, for example by extending SPARQL [4], the de facto standard semantic web querying language, or OWL adaptations such as SPARQL-DL [11] or SQWRL [9].

Manipulation of the temporal ontology, specifically the insertion and removal of temporal axioms, is managed by using of the `AddTemporalAxiom` and `RemoveTemporalAxiom` objects. These objects follow the *Strategy* design pattern [3]; they encapsulate commands that may be aggregated in a collection and then be passed to collaborating objects for execution.

A similar approach is used in the OWL API to solve the *impedance mismatch* problem that may occur when axioms are added or removed one-by-one, which may leave the ontology in an inconsistent state in-between updates, and is particularly problematic when the ontology has registered observers (i.e., implementations of the *Observer* design pattern [3]) that depend on its consistency. Processing a collection of `AddTemporalAxiom` and `RemoveTemporalAxiom` objects in bulk mitigates such issues.

5 Parsing and serialization

Parsing and serialization are performed using the `Parser` and `Serializer` interfaces, respectively. Each has a representation-scheme-specific specialization (e.g., `FluentsSerializer`), and all are initialized with data on the capabilities and vocabulary of the chosen representation through the `RepresentationScheme` object.

The `Parser` follows the *Builder* design pattern [3]: it iterates through a provided `OWLOntology` object that represents the temporal ontology using a particular representation scheme, and builds and modifies structures that together represent the eventual `TemporalOntology` as it goes. As the final step in the build process, the `TemporalOntology` is built when the `build()` method is called.

The `Serializer`, on the other hand, is implemented as a *Visitor* that visits every `TemporalAxiom` in the ontology and serializes it as one or more OWL axioms. The insight here is that all `TemporalAxioms` can be serialized independently of one another (something that is not true for *parsing*, where generally multiple OWL axioms must be parsed in combination in order to extract a single `TemporalAxiom`). The `Serializer` internally stores the set of generated axioms, and produces an `OWLOntology` from this set when the `createOntology()` method is called.

In the treatment of the parsing and serialization of the reification and 4D-fluents schemes, below, we express the used algorithms in pseudocode, as snippets of the actual Java code would simply be too verbose and would distract from the key procedures that we wish to illustrate.

5.1 Reification representation schemes

This section presents the methods of conversion between the reification representation scheme and the temporal conceptual model. We shall illustrate the conversion for the general pattern of the reification representation, since this pattern is both more common and more elaborately documented than its alternative.

We assume the presence of an exhaustive list of names of fluent properties in the model in order to aid the conversion from the reification representation scheme to the temporal conceptual model. Such a list is necessary, because there is no clear method of distinguishing the classes and instances that represent fluent properties and their assertions otherwise. Furthermore, for conversions in both directions, we need to be able to determine, for every fluent property, which identifiers are used for the properties that connect the reified fluent property assertion to its subject and object (for fluent object properties), or value (for fluent datatype properties). These identifiers may either be explicitly provided for each fluent property, or they may be generated automatically; for example, by attaching an appendix (e.g. "_s" for the subject relation, and "_o" for the object relation) to the name of the fluent property. In the conversion algorithms below, we pass the list of names of fluent properties as *fluents*, and these come associated with the identifiers for the properties that link the subject and the object to the reified relation as r_s and r_o, respectively.

In Kala, the `ReificationParser` and `ReificationSerializer` accept a `ReificationConfiguration` configuration object for this list of fluent properties. This object stores, for each fluent property intended to be parsed or serialized, the following information:

- The IRI of the fluent property. In a temporal ontology using the reification representation scheme, this IRI coincides with the IRI of the class that represents the fluent property.
- (Optional) The *pattern* for naming the subject relation (r_s): this may be either a constant or a method for decorating the name, e.g., appending "_s" after the fluent property name. The "_s" suffixing is used as the default.
- (Optional) The *direction* of the subject relation, i.e. either from the reified property to the subject, or from the subject to the reified property. The value for this option defaults to the former.
- (Optional) The *pattern* for naming the object relation (r_o): this may be either a constant or a method for decorating the name, e.g., appending "_o" after the fluent property name. The "_o" suffixing is used as the default.
- (Optional, fluent object properties only) The *direction* of the object relation, i.e. either from the reified property to the object, or from the object to the reified property. The value for this option defaults to the former. For fluent datatype properties, the direction is always from the reified property to the value.

Figure 3 shows an example of a fluent object property assertion in the reification representation scheme:

```
FluentObjectPropertyAssertion(a:sam,
    a:ceoOf,a:ibm,_:i1)
```

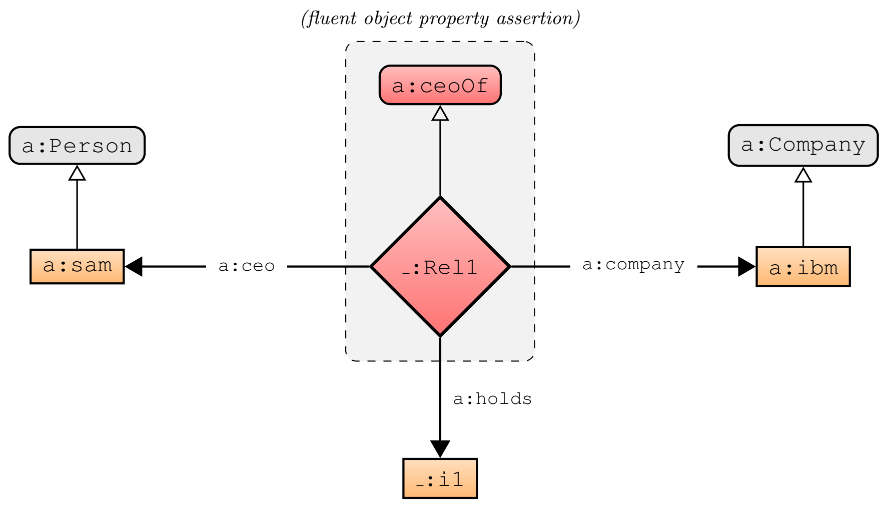

Figure 3: A fluent object property assertion in the reification scheme. The area enclosed by the dashed line represents our abstraction of the fluent object property assertion.

The IRI of the fluent object property is `a:ceoOf`. The fluent object property has subject relation r_s `a:ceo` and object relation r_o `a:company`. The directions of these relations are from the reified property to the subject and to the object, respectively. The serialized equivalent, using the OWL Abstract Syntax, is the following:

```
Individual(_:Rel1
  type(a:ceoOf)
  value(a:ceo a:sam)
  value(a:company a:ibm)
  value(a:holds _:i1))
```

The algorithms are somewhat simplified in two aspects. Firstly, the direction of the r_s property is mostly arbitrary: it may be defined as directed from the reified relation to the intended subject or vice versa, and the choice will affect some details of the implementation of the reification scheme: for example, whether the domain or the range restrictions of r_s correspond to the domain restrictions of the fluent property f, and whether the subject or object of assertions of r_s correspond to the subject of assertions of f. We have left out these details in order not to distract from the general procedure of converting to or from a reification representation scheme. A similar argument holds for r_o. In the algorithms illustrating the conversions, we assume r_s to be directed from the reified relation towards the intended subject, and r_o to be directed from the reified relation to the intended object.

Secondly, the algorithms make no distinction between (fluent) object properties and (fluent) datatype properties. For (fluent) object properties, consider the range of the properties to be restricted to class expressions, and the object of (fluent) object property assertions to be an individual. Similarly, for (fluent) datatype properties, consider the range of the properties to be restricted to a data range, and the object of (fluent) datatype property assertions to be a datatype value.

5.1.1 The interpretation function

Algorithm 5.1 Read from reification representation scheme

```
function READ_REIF($m$, $\mathit{fluents}$)
    $r \leftarrow \emptyset$
    $t \leftarrow \emptyset$
    for all $f \in \mathit{fluents}$ do
        $r_s, r_o \leftarrow$ REIF_CONFIGURATION($f$)
        $D \leftarrow$ RANGE($m, r_s$)
        $R \leftarrow$ RANGE($m, r_o$)
        $f \leftarrow$ FLUENT_PROPERTY($f : D \rightarrow R$)
        $r \leftarrow r \cup f$
        for all $a \in$ INSTANCES($m, f$) do
            $s \leftarrow$ VALUE($m, a.r_s$)
            $o \leftarrow$ VALUE($m, a.r_o$)
            $i \leftarrow$ VALUE($m, a.\mathit{holds}$)
            $a_f \leftarrow$ FLUENT_PROPERTY_ASSERTION($s, f, o, i$)
            $r \leftarrow r \cup a_f$
        end for
        $t \leftarrow t \cup$ TRANSLATED($m, f$)
    end for
    return $r \cup m - t$
end function
```

The function READ_REIF, outlined in Algorithm 5.1, interprets a reification scheme m as an instance of the temporal conceptual model. For each reified fluent property f, it queries the ranges of r_s and r_o in order to determine the intended domain and range restrictions of the fluent property. It then declares the actual fluent property f in the conceptual model. Then, for every instance of the class that represents the fluent property f in the reification scheme m (that is, for every intended assertion of the fluent property), it obtains the subject s, object o, and time interval i for the assertion from the values for r_s, r_o, and *holds*, respectively, and lastly, declares the assertion of the fluent property f from s to o over i.

The algorithm keeps track of the axioms in m that are used in the interpretation of the reification scheme to produce the fluent property f and its assertions, although, for reasons of brevity and clarity, this has not been included in the description of READ_REIF: instead, we use the notation TRANSLATED(m, f) as a shortcut. As the final step in the conversion, all axioms from m that were unused $(m-t)$ are added to the temporal conceptual model to preserve the non-temporal information of m.

5.1.2 The composition function

Algorithm 5.2 shows the function WRITE_REIF, which takes a temporal conceptual model m and produces the reification representation equivalent. It starts by populating the result ontology with all non-temporal information in m. Then, for every fluent property f in m, with domain f_D, range f_R, subject relation r_s and object relation r_o, it introduces a new class to represent f, f_C, introduces r_s and r_o as functional properties from f_C to f_D and f_R, respectively, and introduces existential quantifications on r_s, r_o and *holds* to ensure that every reified relation is completely specified. Lastly, it asserts a new individual for every assertion of the fluent property f and specifies its subject s, object o and time interval i as values for the r_s, r_o and *holds* properties, respectively, and adds the interval i to the resulting ontology.

Algorithm 5.2 Write to reification representation scheme

```
function WRITE_REIF($m$)
    $r \leftarrow$ NONTEMPORAL($m$)
    for all $f[f_D, f_R] \in$ FLUENT_PROPERTIES($m$) do
        $r_s, r_o \leftarrow$ REIF_CONFIGURATION($f$)
        $f_c \leftarrow$ AS_CLASS($f$)
        $r \leftarrow r \cup f_c$
        $r \leftarrow r \cup$ FUNCTIONAL_PROPERTY($r_s : f_c \rightarrow f_D$)
        $r \leftarrow r \cup$ FUNCTIONAL_PROPERTY($r_o : f_c \rightarrow f_R$)
        $r \leftarrow r \cup$ SUBCLASS($f_c \sqsubseteq \exists r_s.f_D \sqcap \exists r_o.f_R \sqcap \exists holds.Interval$)
        for all $a[s, o, i] \in f$ do
            $ai \leftarrow$ NEW_INDIVIDUAL_OF($f_c$)
            $r \leftarrow r \cup ai$
            $r \leftarrow r \cup$ ASSERT($r_s(ai\ s); r_o(ai\ o); holds(ai\ i)$)
            $r \leftarrow r \cup i$
        end for
    end for
    return $r$
end function
```

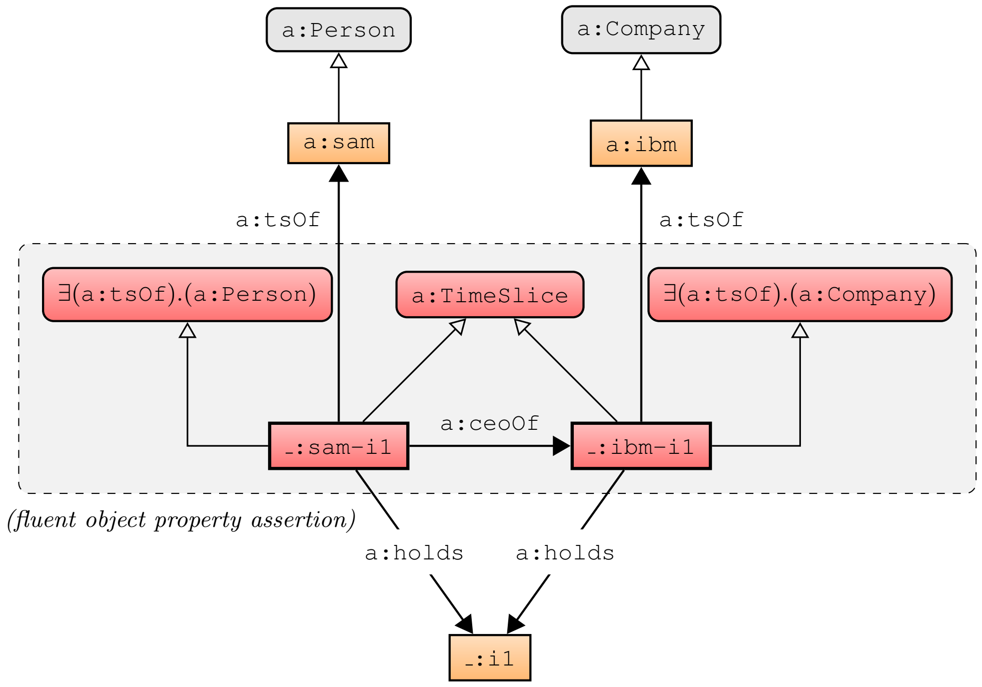

Figure 4: A fluent object property assertion in the 4D-fluents scheme. The area enclosed by the dashed line represents our abstraction of the fluent object property assertion.

5.2 4D-fluents representation schemes

Whereas the conversion functions for the reification representation scheme require the interpretation (and composition) of fluent properties as classes, as well as *a priori* knowledge of which classes represent fluent properties, no such difficulty exists for the 4D-fluents representation scheme. Instead, fluent properties are represented as regular properties, the main difference being that the domains and ranges of these properties are restricted to *timeslices*.

Figure 4 shows an example of a fluent object property assertion in the 4D-fluents representation scheme:

```
FluentObjectPropertyAssertion(a:sam,
    a:ceoOf,a:ibm,_:i1)
```

The serialized equivalent, using the OWL Abstract Syntax, is the following:

```
Individual(_:sam-i1
  type(intersectionOf(
    a:TimeSlice
```

```
      restriction(a:tsOf someValuesFrom(a:Person))))
    value(a:tsOf a:sam)
    value(a:ceoOf _:ibm-i1)
    value(a:holds _:i1))
  Individual(_:ibm-i1
    type(intersectionOf(
      a:TimeSlice
      restriction(a:tsOf someValuesFrom(a:Company))))
    value(a:tsOf a:ibm)
    value(a:holds _:i1))
```

5.2.1 The interpretation function

The function READ_4DF, outlined in Algorithm 5.3, interprets a 4D-fluents representation scheme. We apply the following strategy for the interpretation of a 4D-fluents representation m: for each *timeslice ts* of some individual s over some time interval i, $ts[s, i]$, we obtain the set of property assertions in m that have ts as their subject. We ignore the *timesliceOf* and *holds* properties; any other property that is asserted for ts is necessarily a fluent property.

If the property has not been introduced as a fluent property in the resultant temporal ontology r yet, we need to introduce it to r. The task of determining the domain and range restrictions on the fluent property presents some difficulty, however: the domains and ranges are restricted to *timeslices* of the classes to which we (conceptually) want to restrict the fluent properties. We partially solve this problem by querying for domain and range restrictions of the forms $\exists timesliceOf.D$ and $\exists timesliceOf.R$ for any class expressions D and R, respectively. In the algorithm, this functionality is encapsulated by the functions DOMAIN_TS_RESTRICTION and RANGE_TS_RESTRICTION. However, for the general case, where domain and range restrictions are possibly expressed in a manner different from explicit property restrictions on *timesliceOf*, this remains a problem to be solved (by employing a reasoner to compute the domain and range). Note that fluent datatype properties simply have data ranges as their range restriction, so this problem does not hold for range restrictions on fluent datatype properties.

After ensuring that the fluent property is introduced to the temporal conceptual model, we simply assert the fluent property into the model. For fluent object properties, this requires the additional step of determining the intended object value by obtaining the value of the object timeslice's *timesliceOf* property.

Lastly, as with READ_REIF (see Algorithm 5.1), the algorithm keeps track of the axioms in m that are used in the interpretation of the 4D-fluents scheme, and, similarly, we use the notation TRANSLATED(m, ts) as a shortcut. As the final step in the conversion, all axioms from m that were unused $(m - t)$ are added to the temporal conceptual model to preserve the non-temporal information of m.

Algorithm 5.3 Read from 4D-fluents representation scheme

```
function READ_4DF($m$)
    $r \leftarrow t \leftarrow \emptyset$
    for all $ts[s, i] \in$ TIMESLICES($m$) do
        for all $p(ts, o) \in$ PROPERTY_ASSERTIONS($m, ts$) do
            if $p(ts, o) = tsOf(ts, s)$ or $p(ts, o) = holds(ts, i)$ then
                skip
            end if
            if $p \notin r$ then
                $D \leftarrow$ DOMAIN_TS_RESTRICTION($m, p$)
                if IS_OBJECT_PROPERTY($p$) then
                    $R \leftarrow$ RANGE_TS_RESTRICTION($m, p$)
                else
                    $R \leftarrow$ RANGE($m, p$)
                end if
                $r \leftarrow r \cup$ FLUENT_PROPERTY($p : D \rightarrow R$)
            end if
            if IS_OBJECT_PROPERTY($p$) then
                $o \leftarrow$ VALUE($m, o.tsOf$)
            end if
            $r \leftarrow r \cup$ FLUENT_PROPERTY_ASSERTION($s, p, o, i$)
        end for
        $t \leftarrow t \cup$ TRANSLATED($m, ts$)
    end for
    return $r \cup m - t$
end function
```

5.2.2 The composition function

Algorithm 5.4 shows the WRITE_4DF function, which converts the temporal conceptual model to an instance of the 4D-fluents representation scheme. As with the WRITE_REIF function (see Algorithm 5.2), we start by populating the result ontology with the non-temporal information in the model m. Then, for every fluent property f with domain restrictions f_D and range restrictions f_R, we add the property $f : \exists tsOf.f_D \rightarrow \exists tsOf.f_R$ or $f : \exists tsOf.f_D \rightarrow f_R$ to the result ontology, depending on whether f is a fluent object property or a fluent datatype property, respectively.

Then, for every assertion of f with subject s and object o over time interval i, we obtain a timeslice for s over i through the TIMESLICE function.

The TIMESLICE function provides an injective (one-to-one) mapping pairs of individuals and time intervals to timeslices, which effectively limits the proliferation of timeslice objects somewhat. Then, depending on whether f is a fluent object property or a fluent datatype property, we either similarly create a timeslice indvidual for the object value and assert the property from ts_s to ts_o, or (in the case of a fluent datatype property), we assert the property from ts_s directly to the value o. Last, we add the interval to the resultant ontology.

Algorithm 5.4 Write to 4D-fluents representation scheme

```
function WRITE_4DF($m$)
    $r \leftarrow$ NONTEMPORAL($m$)
    for all $f[f_D, f_R] \in$ FLUENT_PROPERTIES($m$) do
        if IS_OBJECT_PROPERTY($f$) then
            $p \leftarrow$ PROPERTY($f : \exists tsOf.f_D \rightarrow \exists tsOf.f_R$)
        else
            $p \leftarrow$ PROPERTY($f : \exists tsOf.f_D \rightarrow f_R$)
        end if
        $r \leftarrow r \cup p$
        for all $a[s, o, i] \in f$ do
            $ts_s \leftarrow$ TIMESLICE($s, i$)
            $r \leftarrow r \cup ts_s$
            if IS_OBJECT_PROPERTY($f$) then
                $ts_o \leftarrow$ TIMESLICE($o, i$)
                $r \leftarrow r \cup ts_o \cup$ ASSERT($p(ts_s, ts_o)$)
            else
                $r \leftarrow r \cup$ ASSERT($p(ts_s, o)$)
            end if
            $r \leftarrow r \cup i$
        end for
    end for
    return $r$
end function
```

References

[1] Joshua Bloch. *Effective Java, 2E.* Prentice Hall PTR, Upper Saddle River, NJ, 2008.

[2] Jeremy J Carroll, Ian Dickinson, Chris Dollin, Dave Reynolds, Andy Seaborne, and Kevin Wilkinson. Jena: implementing the semantic web recommendations. In *13th International World Wide Web conference*

on Alternate track papers & posters, pages 74–83, New York, NY, 2004. ACM Press.

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Abstraction and Reuse of Object-Oriented Design*. Springer, Berlin, 2001.

[4] Steven Harris and Andy Seaborne. SPARQL 1.1 query language. W3C Recommendation 21 March 2013, 2013.

[5] Jörg Henß, Joachim Kleb, Stephan Grimm, and Jürgen Bock. A database backend for OWL. In *5th International Workshop on OWL: Experiences and Directions (OWLED 2009)*, 2009.

[6] Matthew Horridge and Sean Bechhofer. The OWL API: A Java API for working with OWL 2 ontologies. In *5th International Workshop on OWL: Experiences and Directions (OWLED 2009)*, 2009. `http://ceur-ws.org/Vol-529/owled2009_submission_29.pdf`.

[7] Holger Knublauch, Ray W Fergerson, Natalya F Noy, and Mark A Musen. The Protégé OWL plugin: An open development environment for semantic web applications. In *3rd International Semantic Web Conference (ISWC 2004)*, pages 229–243. Springer, Berlin, 2004.

[8] Boris Motik, Peter F Patel-Schneider, Bijan Parsia, Conrad Bock, Achille Fokoue, et al. OWL 2 web ontology language: Structural specification and functional-style syntax. W3C Recommendation 11 December 2012, 2012.

[9] Martin J O'Connor and AK Das. SQWRL: a query language for OWL. In *5th International Workshop of OWL: Experiences and Directions (OWLED 2009)*, 2009. `http://ceur-ws.org/Vol-529/owled2009_submission_42.pdf`.

[10] Rob Shearer, Boris Motik, and Ian Horrocks. HermiT: A highly-efficient OWL reasoner. In *5th International Workshop on OWL: Experiences and Directions (OWLED 2008)*, pages 26–27, 2008. `http://www.cs.ox.ac.uk/boris.motik/pubs/smh08HermiT.pdf`.

[11] Evren Sirin and Bijan Parsia. SPARQL-DL: SPARQL query for OWL-DL. In *3rd International Workshop of OWL: Experiences and Directions (OWLED 2007)*, 2007. `http://ceur-ws.org/Vol-258/`.

[12] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical OWL-DL reasoner. *Web Semantics: Science, Services and Agents on the World Wide Web*, 5(2):51–53, 2007.