On the interplay between validation and inference in SHACL - an investigation on the Time Ontology

Livio Robaldo [a,*] and Sotiris Batsakis [b,c]
[a] *HRC School of Law, Swansea University, Wales, UK*
E-mail: livio.robaldo@swansea.ac.uk
[b] *Hellenic Mediterranean University, Greece*
E-mail: sbatsakis@hmu.gr
[c] *Huddersfield University, UK*
E-mail: s.batsakis@hud.ac.uk

Abstract.
This paper presents a novel framework for validating the Time Ontology (https://www.w3.org/TR/owl-time). The Time Ontology is currently a W3C candidate recommendation draft and is widely recognized as the "de facto" standard for representing temporal data in the Semantic Web. However, its current axiomatization in OWL is unable to enforce several constraints on temporal data, which are instead captured by the SHACL formalization proposed in this paper. Besides providing a practical tool for processing temporal data in RDF within applications, this paper also offers insights into the combined use of SHACL shapes and SHACL-SPARQL rules to properly capture the interplay between validation and inference in knowledge graphs. Specifically, we demonstrate that SHACL shapes alone are insufficient for validating certain knowledge graphs that can be asserted using the current vocabulary of the Time Ontology. To ensure proper validation, we first compute the inferred knowledge graphs using SHACL-SPARQL rules, then validate the inferred graphs through SHACL shapes. We argue that these findings extend beyond the Time Ontology and apply more broadly, even in the context of more advanced reasoning rules. In light of this, we see our work as a call to action for the Semantic Web community to systematically investigate the representational requirements of different use cases in order to identify the minimal inference rules necessary for validating data in those use cases. The SHACL shapes and SHACL-SPARQL rules that define the proposed framework are freely available on the GitHub repository https://github.com/liviorobaldo/TimeOntologyInSHACL, along with Java programs and clear instructions for processing them.

Keywords: Time Ontology, Temporal relations, Validation and inference in SHACL

1. Introduction

Time is a fundamental aspect of reality and the representation of dynamic phenomena. These appear in many application domains, e.g., planning, robotics, compliance checking, real-time systems, computer aided engineering, and any other application that requires to model actions, changes, or behaviors.

There are various aspects of time that need to be taken into account when providing definitions of temporal elements [1]: time can be bounded or not, discrete or continuous, fuzzy or non-fuzzy, periodic, absolute or relative,

*Corresponding author. E-mail: livio.robaldo@swansea.ac.uk.

1570-0844/$35.00 © 0 – IOS Press. All rights reserved.

linear or branching. In addition, temporal representations can be focused on points or intervals, which in turn can be crisp or fuzzy, bounded or unbounded, convex or non-convex, open or closed.

These aspects have been widely studied in past literature within specialized Temporal Logics [2][3][4] and extensively used within tasks such as modeling the behavior and properties of state transition systems (model checking) [5]. As a result, model checkers such as NuSMV [6][7], among others, have been developed.

The Semantic Web also requires a corresponding representation of time [8]. Semantic Web standards are based on Resource Description Framework (RDF) [9] for representing interconnected data on the Web and on Web Ontology Language (OWL) [10] for reasoning over data encoded in RDF. OWL is based on the theoretical work done in Description Logics [11]; currently, the mostly used version of OWL is perhaps OWL-2 [12].

Some proposals for extending Description Logics and OWL with the expressiveness of Temporal Logics have been investigated, although they have not been incorporated into the family of Semantic Web standards. For instance, [13] and [14] introduced Temporal Description Logics as a formal approach for representing time in Description Logics. Similarly, [15] proposed an extension of OWL2-QL for ontology-based data access.

Nevertheless, these and other similar initiatives have only been developed at the theoretical level, with the primary aim of addressing computational complexity and decidability issues, and they lack implementations.

More generally, while reviewing the relevant literature, briefly outlined below in Section 2, we observed that most proposed approaches focus on defining vocabularies of RDF resources for *representing* temporal knowledge in the Semantic Web. However, they largely overlook the *inferences* that can be drawn from the represented knowledge. In fact, the same observation seems to apply to the Semantic Web as a whole, not just to the representation of temporal knowledge (cf. [8]). In light of this, we believe that more efforts should be devoted by the Semantic Web community to advancing automated reasoning on knowledge graphs.

The starting point of this paper is that OWL is intrinsically inadequate for representing and reasoning with temporal data, primarily because its vocabulary lacks constructs for comparing and processing quantitative temporal entities such as dates, hours, and minutes.

This paper specifically investigates the Time Ontology[1], which is currently a W3C Candidate Recommendation Draft and is regarded as a "de facto" standard for representing temporal knowledge in the Semantic Web. Several ontologies across various domains incorporate the Time Ontology to model temporal entities.

The vocabulary of the Time Ontology includes RDF resources for representing both quantitative and qualitative aspects of instants and intervals of time across various reference systems, such as the standard Gregorian calendar, non-Gregorian calendars, Unix time, and geologic time [16]. Notably, the Time Ontology's vocabulary features RDF properties corresponding to the well-known Allen's temporal relations [17]. However, the full version of Allen's interval algebra [17] is not currently implemented in the Time Ontology, as will be explained later in this paper.

The Time Ontology is currently implemented in OWL-2 DL. Therefore, as explained above, it only enables inferences that are irrelevant from the perspective of temporal processing. As a consequence, in the Time Ontology it is currently possible to encode absurd temporal data such as intervals that end before they start or pairs of intervals that are disjoint but, at the same time, they overlap. OWL is unable to identify these intervals as inconsistent or, more precisely, as *invalid*, thereby significantly reducing the usefulness of the Time Ontology within existing applications.

It must be however observed that OWL was designed for *reasoning* with RDF knowledge graphs, whereas for *validating* RDF knowledge graphs another format has been released through an official W3C recommendation: the Shapes Constraint Language (SHACL). An increasing number of researchers have started using SHACL, e.g., [18], [19], [20], [21], and this paper represents a new contribution to this growing body of literature.

SHACL has two main components: SHACL Core[2] and SHACL-SPARQL.[3] SHACL Core is the standard, built-in set of constraints that can be used to validate RDF data without requiring additional programming. It includes predefined constraint types such as cardinality, value ranges, and datatype restrictions. SHACL-SPARQL, on the other hand, extends SHACL Core by allowing users to define custom constraints using SPARQL queries. By incorporating SPARQL, it provides greater flexibility and expressiveness, enabling complex validation rules that go beyond SHACL Core, such as advanced logic, conditional constraints, and cross-graph dependencies.

[1] https://www.w3.org/TR/owl-time
[2] https://www.w3.org/TR/shacl/#core-components
[3] https://www.w3.org/TR/shacl/#sparql-constraints

This paper presents a set of SHACL shapes to validate the RDF resources in the Time Ontology. However, we do not consider *all* RDF resources within the Time Ontology, but instead focus on a specific *fragment*, discussed below in Section 3: the fragment of the ontology related to the `xsd:dateTime` datatype. This is the single datatype for which SPARQL v1.1 includes operators for comparison.[4]

This paper demonstrates that SHACL shapes alone are insufficient for validating RDF knowledge graphs, even when considering only a few simple RDF classes and properties. While we will demonstrate this with respect to the fragment of the Time Ontology discussed in Section 3, we believe that our findings are broadly applicable: SHACL shapes are not expressive enough to identify all patterns of invalid RDF triples *in general*.

To detect these patterns, it is necessary to *infer* additional triples from the explicitly asserted ones. After that, SHACL shapes can be written to identify the invalid patterns *within the inferred knowledge graph*. Since OWL cannot be used for these inferences, because it does not support inferences on temporal data, as explained above, this paper uses SHACL shapes in combination with *SHACL rules*, as defined in the W3C Working Group Note from 08 June 2017[5]. Note that SHACL rules have not yet been released as part of an official W3C recommendation.

In particular, we used SHACL-SPARQL rules[6], which, as the name suggests, are based on SPARQL and thus capable of processing `xsd:dateTime` values. SHACL-SPARQL rules are SPARQL queries in the form `CONSTRUCT-WHERE` that create new triples in the knowledge graph based on the triples already asserted.

Note that executing the SHACL shapes *after* the SHACL-SPARQL rules does not affect the validation results: SPARQL queries in the form `CONSTRUCT-WHERE` only *add* triples to the knowledge graph. Consequently, if the initial knowledge graph already contains invalid triples, these will also be present in the inferred knowledge graph.

In addition, similar to standard OWL reasoners such as HermiT [22], which re-execute OWL axioms until no further triple is inferred, this paper provides evidence that SHACL-SPARQL rules should also be re-executed until no further triple is inferred, prior to validation. This approach is not prescribed in the aforementioned Working Group Note, nor is it implemented by existing libraries that process SHACL shapes and rules, such as the TopBraid SHACL Java library v.1.3.2[7], which we used in our implementation. These libraries execute SHACL rules only once. In light of this, the software available on the GitHub repository associated with this paper programmatically re-executes the SHACL-SPARQL rules until no further triple is inferred, after which it validates the inferred knowledge graph against the SHACL shapes. This paper argues that this approach should be adopted *in general* by any software designed to validate knowledge graphs using SHACL. Specifically, we believe that what our implementation on GitHub "forcibly" accomplishes programmatically should be instead officialized by the W3C.

The rest of the paper is organized as follows. The next section briefly reviews the current literature on representing time in the Semantic Web. Section 3 then focuses on describing the Time Ontology, specifically the fragment that will be considered in the proposed formalization, i.e., only the RDF resources related to the `xsd:dateTime` datatype. Sections 4, 5, 6, and 7 present the SHACL shapes and SHACL-SPARQL rules for the Time Ontology fragment discussed in Section 3, designed to validate temporal data encoded with its vocabulary. These four sections could be merged into a single one; however, since their content is rather long, we chose to split them into four parts to improve readability. Section 8 concludes the paper and outlines directions for future work.

2. Background: representing temporal data in the Semantic Web

Time is not inherently integrated into Semantic Web standards, and maintaining compatibility with these standards requires representations that incorporate reasoning rules into existing ontologies, rather than relying on specialized reasoning software. There are two main components for representing time in a machine-readable format, as needed for application use:

[4] See https://www.w3.org/TR/xmlschema-2/#dateTime. However, it is worth noting that most SPARQL implementations "unofficially" extend the coverage of the official SPARQL v1.1 operators and functions to other datatypes, such as `xsd:date` or `xsd:duration`. Future official recommendations of the standard may also incorporate these extensions.

[5] https://www.w3.org/TR/shacl-af, retrieved March 10, 2025

[6] https://www.w3.org/TR/shacl-af/#SPARQLRule

[7] https://repo1.maven.org/maven2/org/topbraid/shacl/1.3.2

a. The representation of temporal concepts such as time points and intervals.

b. The representation of dynamic properties (fluent properties) of objects and events using the above mentioned temporal concepts.

Core temporal concepts, their properties, and constraints are defined using temporal ontologies, while the application of these properties in specific domains is an orthogonal dimension. Temporal ontologies include definitions of temporal intervals and points, among others, and these definitions can be used to represent temporal properties in various ways, such as through 4D-fluents or reification.

Although the focus of this paper is on (a), specifically the representation of temporal concepts based on the definitions in the Time Ontology, the next two subsections will briefly survey past literature on representing both aspects of temporal representation in the Semantic Web, i.e., (a) and (b) above, respectively.

2.1. Temporal Ontologies

Time is a fundamental aspect of the world and temporal concepts are involved on almost all knowledge representation tasks since many properties of objects are dynamic. Thus, many temporal ontologies have been proposed in the literature [1]. Among these, the Time Ontology in OWL, also know as OWL-Time[8], is the most widely used[9]. A recent survey on the representation and management of temporal data is provided in [23].

The Time Ontology is a temporal ontology and W3C candidate recommendation draft for providing definitions of temporal points and intervals and their relations. However, as explained in the Introduction, it features limited reasoning capabilities over these. Having the status of a W3C candidate recommendation draft, it is widely used, but since the adoption of the Time Ontology as a standard is an ongoing work, several alternative temporal ontologies have also been proposed. Since the essence of the Semantic Web is the definition of common vocabularies, this work focuses on enhancing the Time Ontology thus contributing to the standardization effort rather than proposing yet another temporal ontology.

Other temporal ontologies include Resusable Time Ontology [24], TimeML [25], which offers a translation to DAML-OIL, the predecessor of OWL, GFO-Time [26], which is part of the upper ontology GFO, TOWL [27], which extends OWL with temporal constructs but is not compliant with standard Semantic Web tools, and TL-OWL [28], which also extends Semantic Web standards with temporal concepts, similarly to OWL-Met [29]. In PSI-ULO [30] temporal concepts are part of an upper ontology defined in PSI-Time [31] while in TimeLine ontology [32] definitions for temporal concepts for digital music are provided. Temporal RDF [33] proposes extending RDF with temporal annotations while SWRL-Time [34], CNTRO [35] and SOWL [36] define reasoning mechanisms based on Semantic Web Rule Language (SWRL) [37].

This paper proposes using SHACL and SHACL-SPARQL rules as an alternative to SWRL in order to enhance the current version of the Time Ontology. The reason for this, as explained below, is that SWRL, unlike SHACL-SPARQL, does not allow for the creation of new individuals, which is necessary to validate certain temporal data modeled through the properties in the Time Ontology's vocabulary.

While SHACL has been an official W3C recommendation since 2017[10], neither SWRL nor SHACL-SPARQL rules have (yet?) achieved standard status. In other words, currently both are only *proposals* for W3C recommendations. SWRL has been a proposal since 2004[11], while SHACL-SPARQL rules have been a proposal since 2017[12].

Summarizing, when definitions of temporal concepts are needed several alternatives are proposed without a single standard currently in common use, which is a problem since standardization and use of common vocabularies and definitions is at the core of the Semantic Web vision. The Time Ontology, being a W3C candidate recommendation draft is the most important of the above proposals and the focus of the current work. The Time Ontology will be further described in the next section.

---

[8]https://www.w3.org/TR/owl-time
[9]Example applications are listed in: https://www.w3.org/2015/spatial/wiki/OWL_Time_Ontology_adoption
[10]https://www.w3.org/TR/shacl
[11]https://www.w3.org/submissions/SWRL
[12]https://www.w3.org/TR/shacl-af

2.2. Representation of dynamic/fluent properties

Almost all conceivable application domains involve objects with dynamic properties, also known as *fluent properties*, that change over time. The definitions of the temporal concepts involved (e.g., the temporal instant when an event occurs or the temporal interval during which a dynamic property holds) are provided by temporal ontologies. However, their application in specific domains is not straightforward, as fluent properties are not binary (e.g., they involve a subject, an object, and a temporal instant or interval). As a result, they cannot be directly represented as object or datatype properties of a class, leading to many different approaches in practice for using temporal concepts.

Integrating temporal concepts defined in temporal ontologies with representations of fluent properties in the Semantic Web can be achieved in various ways such as extending repositories with support for ternary relations [38] (standard RDF is based in triple stores), versioning [39], which is based on creating a new copy of the knowledge base when a property is modified, named graphs [40], the generic method of reification, and the 4D fluents approach proposed in [41]. Various methods are presented and compared in [42] and they have been extended to cover spacial properties of objects in [43]. The work in [42] retains compatibility with OWL/RDFS and offers integrated reasoning capabilities which is not the case of other approaches such as versioning and temporal annotated named graphs.

Again, due to certain limitations of OWL regarding property chains, temporal reasoning in [42], as well as in SWRL-Time [34] and CNTRO [35], is achieved through SWRL.

In [44], the CHRONOS ED tool was proposed, offering enhanced performance compared to the work in [42]. However, this approach was somewhat ad-hoc, as it relied on specialized reasoning software that was compatible with specific ontologies, rather than using generic semantic OWL reasoners such as HermiT [45] or Pellet [46], which limited its overall applicability.

It is important to note that temporal ontologies, which provide definitions for temporal instances and intervals, can be used in conjunction with the above-mentioned approaches. This paper precisely proposes a novel set of SHACL shapes and SHACL-SPARQL rules to extend the validation and reasoning capabilities of the Time Ontology, designed to work alongside the aforementioned approaches for representing and reasoning with fluent properties.

3. The Time Ontology

The Time Ontology is currently a W3C candidate recommendation draft[13] publicly available online and downloadable in Turtle format[14]. Its IRI is "`http://www.w3.org/2006/time#`"; in this paper, as well as in the associated GitHub repository, we will refer to this IRI with the prefix "`time:`". The version of the Time Ontology used in this paper is the one retrieved on March 10, 2025. A copy of this version is available on the GitHub repository associated with this paper; of course, subsequent versions of the Time Ontology could not be compatible with the implementation proposed in this paper.

While the full formal definitions of the ontology's resources (classes, individuals, and properties) are available at the Time Ontology's homepage, this section will focus on the resources that may be processed via SHACL, namely the ones associated with the `xsd:dateTime` datatype. As pointed out in the Introduction, `xsd:dateTime` is the single temporal datatype for which SPARQL 1.1 defines comparison operators[15], therefore it is the single one for which it is currently possible to assert SHACL shapes and rules. These resources are shown in Figure 1.

---

[13] https://www.w3.org/TR/owl-time

[14] https://www.w3.org/TR/turtle

[15] See https://www.w3.org/TR/2013/REC-sparql11-query-20130321/#OperatorMapping. In addition, note that the timezone is *optional* in `xsd:dateTime`. Although the algorithm comparing `xsd:dateTime` values produce an output even when the timezone is omitted (see section "3.2.7.4 Order relation on dateTime" at https://www.w3.org/TR/2004/REC-xmlschema-2-20041028/#dateTime), this output lacks meaningful application value. For this reason, the datatype property `inXSDDateTime` has been *deprecated* in the current version of the Time Ontology while the datatype property `inXSDDateTimeStamp` has been recommended in its place. The latter associates instances of `Instant` with values of the `xsd:dateTimeStamp` datatype, which is similar to the `xsd:dateTime` datatype but requires the timezone to be explicitly specified (see https://www.w3.org/TR/xmlschema11-2/#dateTimeStamp). From the perspective of the research presented in this paper, this is not a critical issue. All `xsd:dateTimeStamp` values can be converted to `xsd:dateTime` values using the SPARQL 1.1 function `strdt` (https://www.w3.org/TR/sparql11-query/#func-strdt). Since this is a mere technicality, for simplicity the SHACL shapes and rules below will use `xsd:dateTime`, although we will include a shape that requires the timezone in `xsd:dateTime` values, shown below in (4).

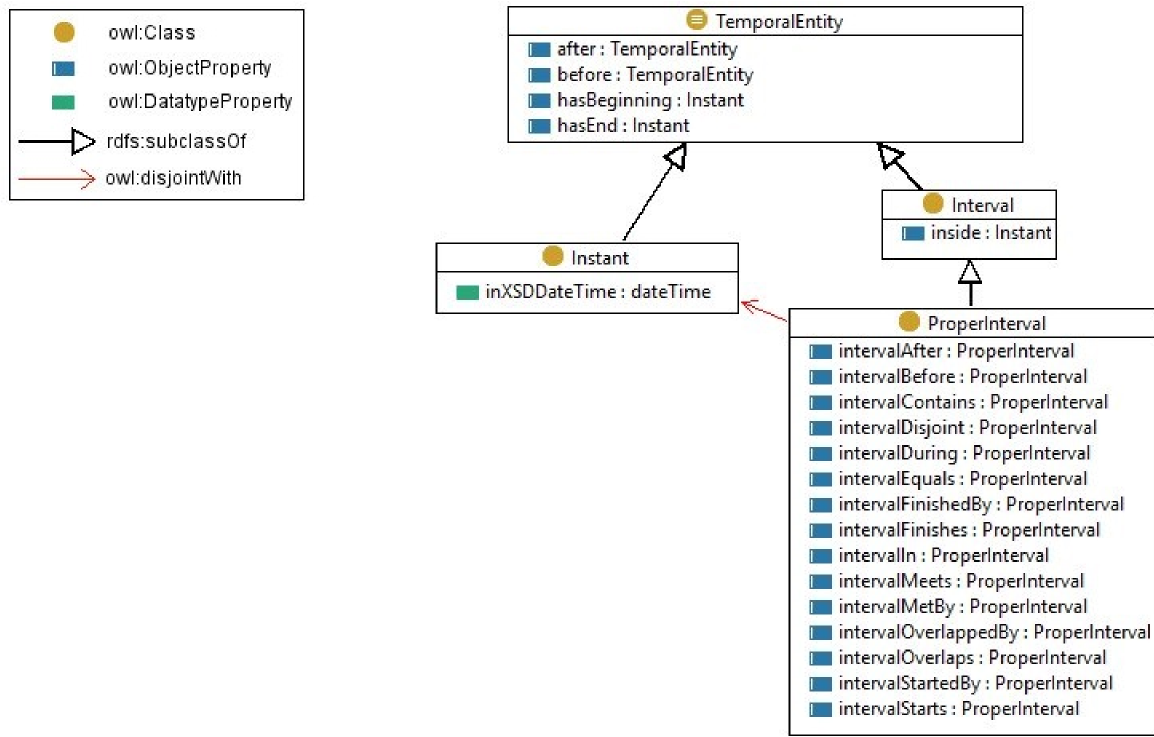

Fig. 1. Classes and properties of the Time Ontology considered in this paper. This figure is a reduced version of the figure at https://www.w3.org/TR/owl-time/#topology (retrieved on March 10, 2025)

The top class is `TemporalEntity`, which has two direct subclasses[16]: `Instant` and `Interval`. These represent the two main conceptual entities of the ontology. Individuals of `Instant` refer to exact moments in time while individuals of `Interval` refer to spans of infinite and contiguous instants between a start and an end instant. The latter are respectively identified by the object properties `hasBeginning` and `hasEnd`, inherited from the upper class `TemporalEntity`.

The class `Interval` has a direct subclass called `ProperInterval`, which is defined as an `Interval` "for which the value of the beginning and end are different". This constraint does not hold for the upper class `Interval`, whose instances can then have the *same* `Instant` as beginning and end, in which case they are instants themselves. For this reason, the two classes have been declared as *disjoint* (`owl:disjointWith`) in the Time Ontology.

This is one of the only two disjointness statements currently asserted in the Time Ontology. The other one holds between the two object properties `intervalEquals` and `intervalIn`, which are related via `owl:propertyDisjointWith`. Therefore, the Time Ontology currently permits any assertion but those that do not comply with these two disjointness statements.

In addition, the Time Ontology also includes several further `owl:allValuesFrom`, `owl:hasValue`, and `owl:cardinality` restrictions as well as several `owl:FunctionalProperty`, `owl:inverseOf`, etc. properties, but most of these are not relevant for this paper's objectives.

Concerning properties, this paper considers a single datatype property, i.e., `inXSDDateTime`, which connects instances of `Instant` (domain) with values of the datatype `xsd:dateTime` (range). The object properties `hasBeginning` and `hasEnd` respectively specify the beginnings and the ends of the temporal entities; these two properties connect instances of `TemporalEntity` (domain) with instances of `Instant` (range). The property `inside` is similar to `hasBeginning` and `hasEnd` but its domain is the class `Interval` rather

[16] `TemporalEntity` is actually defined as the *union* of `Instant` and `Interval`; in other words, all instances of `TemporalEntity` are also instances of either `Instant` or `Interval` (or both).

than TemporalEntity; the property inside connects intervals with instants occurring therein. The properties after and before allows encoding of temporal orders between temporal entities. Finally, the fifteen properties occurring in Figure 1 within the box associated with ProperInterval are those denoting Allen's temporal relations. They connect instances of ProperInterval (domain) with instances of the same class (range); further discussion about these fifteen properties can be found in Subsection 3.1 below.

It is easy to re-implement the RDFS and OWL axioms in the Time Ontology as SHACL shapes or SHACL-SPARQL rules. However, not all of these axioms from the Time Ontology have been incorporated into the proposed approach. Our implementation includes only the following:

(1) - SHACL-SPARQL rules to carry out the inferences denoted by RDFS:subclassOf, RDFS:domain, RDFS:range, and RDFS:subPropertyOf.

- SHACL-SPARQL rules to carry out the inferences denoted by owl:inverseOf.

- A SHACL shape to implement the property owl:disjointWith holding between the classes Instant and ProperInterval, namely to invalidate knowledge graphs that contain individuals occurring as instances of both classes.

However, these shapes and rules are omitted from this paper, though they can still be found in the GitHub repository. In addition, the property owl:propertyDisjointWith, which relates intervalEquals and intervalIn, is also included in our implementation. However, in Section 6 below, we will propose a more general scheme for relating the properties listed in Figure 1 within the box associated with ProperInterval. This scheme encompasses several other disjunctions among these properties besides the one between intervalEquals and intervalIn.

On the other hand, besides the constraints currently asserted in the Time Ontology, many further constraints should be imposed on temporal information, as explained in the Introduction. For example, it must be checked that, for every ProperInterval, its beginning temporally occurs *before* its end. However, OWL is not expressive enough to encode them. Therefore, for example, the Time Ontology currently considers as valid an absurd ProperInterval whose hasBeginning is 31st December 2023 and whose hasEnd is 1st January 2023.

These additional constraints will be discussed in the following sections. Before that, however, it is worthwhile to provide some further details about the properties of the Time Ontology that represent Allen's temporal relations, as three sections below (from Section 5 to Section 7) are entirely dedicated to their validation and inference.

3.1. The Time Ontology properties denoting Allen's temporal relations

Allen's temporal relations are thirteen disjoint basic relations that can hold between two proper temporal intervals. They were originally defined in [17] and have since been incorporated into major approaches to temporal reasoning proposed in the literature.

In the Time Ontology, each Allen's temporal relation is represented by one of the object properties shown in Figure 1 within the box associated with ProperInterval.

One of these relations is Equal, denoted by the Time Ontology's property intervalEqual. If two intervals are related by Equal, then they are the same interval, meaning that both their beginnings and their ends coincide. Six of the other twelve relations, i.e., Before, During, Meets, Starts, Finishes, and Overlaps, which are respectively denoted by the Time Ontology's properties intervalBefore, intervalDuring, intervalMeets, intervalStarts, intervalFinishes, and intervalOverlaps, may be thought as the "main" ones, while the remaining six, i.e., After, Contains, MetBy, StartedBy, FinishedBy, and OverlappedBy, which are respectively denoted by the Time Ontology's properties intervalAfter, intervalContains, intervalMetBy, intervalStartedBy, intervalFinishedBy, and intervalOverlappedBy, are their *inverse* properties.

The six "main" properties are defined as follows:

(2) a. `T1 Before T2`: `T1` temporally occurs before `T2`.
 b. `T1 During T2`: `T2` *properly* contains `T1`.
 c. `T1 Meets T2`: the end of `T1` coincides with the beginning of `T2`.
 d. `T1 Starts T2`: the beginning of `T1` coincides with the beginning of `T2`, while the end of `T1` occurs temporally before the end of `T2`.
 e. `T1 Finishes T2`: the end of `T1` coincides with the end of `T2`, while the beginning of `T1` occurs temporally after the beginning of `T2`.
 f. `T1 Overlaps T2`: the beginning of `T1` occurs temporally before the beginning of `T2` and the end of `T1` occurs temporally before the end of `T2`.

In addition to the thirteen basic temporal relations, Allen adds the relation `In` as "a predicate that summarizes the relationships in which one interval is wholly contained in another". The predicate is defined as follows:

$$\forall_{T1,T2}[\mathtt{In}(T1,T2) \leftrightarrow (\mathtt{During}(T1,T2) \vee \mathtt{Starts}(T1,T2) \vee \mathtt{Finishes}(T1,T2))]$$

In the Time Ontology, the temporal relation `In` corresponds to the object property `intervalIn`, of which `intervalDuring`, `intervalStarts`, and `intervalFinishes` are declared as sub-properties. Note that `In` and `intervalIn` denote *proper* inclusion, i.e., inclusions of an interval within another one but without the two intervals being coincident. For that reason, in the Time Ontology `intervalIn` and `intervalEquals` are declared as disjoint, i.e., related through the OWL property `owl:propertyDisjointWith`.

A final (fifteenth) property included in the Time Ontology is the property `intervalDisjoint`, which relates two intervals that do not share any instant. The properties `intervalBefore` and `intervalAfter` are declared as sub-properties of `intervalDisjoint`. Contrary to `intervalIn`, there is no temporal relation corresponding to `intervalDisjoint` in Allen's original framework from [17].

Allen also defines a *composition table* for the thirteen basic relations. Given the temporal relation `r1` between two intervals `T1` and `T2`, and the temporal relation `r2` between `T2` and a third interval `T3`, the table indicates the possible temporal relations between `T1` and `T3`.

For example, if `r1` is `Meets` and `r2` is `Finishes`, we know that `T1`'s end coincides with `T2`'s beginning, that `T3`'s end coincides with `T2`'s end, and that `T3`'s beginning occurs before `T2`'s beginning. Nevertheless, as shown by the left and right arrows in Figure 2, we do *not* know the position of `T3`'s beginning with respect to `T1`'s beginning. Therefore, there are three options for the relation between in `T1` and `T3`, shown at the bottom of Figure 2: either `T1` occurs `During` `T3`, or it `Starts` at `T3`, or it `Overlaps` with it.

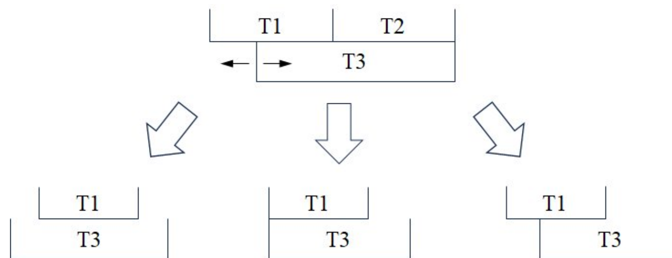

Fig. 2. Composing `Meets` and `Finishes`: If `T1` meets `T2` and `T2` finishes `T3`, then `T1` can either be included in, start, or overlap with `T3`.

The full composition table is reported in Table 1. For space constraints, the columns and the cells of the Table only report the symbols associated with the temporal relations; the symbols are defined in each cell of the first column between round brackets. In each cell, the listed symbols are all (mutually exclusive) options for the temporal relation between `T1` and `T3`. `Equals` is omitted from the table because it is trivial: if `r1` is `Equal`, then the temporal

T2 **r2** T3 / T1 **r1** T2	<	>	d	di	o	oi	m	mi	s	si	f	fi
Before (<)	<	*any*	< o m d s	<	<	< o m d s	<	< o m d s	<	<	< o m d s	<
After (>)	*any*	>	> oi mi d f	>	> oi mi d f	>	> oi mi d f	>	> oi mi d f	>	>	>
During (d)	<	>	d	*any*	< o m d s	> oi mi d f	<	>	d	> oi mi d f	d	< o m d s
Contains (di)	< o m di fi	> oi di mi si	o oi d si fi di f s =	di	o di fi	oi di si	o di fi	oi di si	o di fi	di	oi di si	di
Overlaps (o)	<	> oi di mi si	o d s	< o m di fi	< o m	o oi d si fi di f s =	<	oi di si	o	di o fi	o d s	< o m
OverlappedBy (oi)	< o m di fi	>	oi d f	> oi di mi si	o oi d si fi di f s =	> oi mi	o di fi	>	oi d f	> oi mi	oi	oi di si
Meets (m)	<	> oi mi di si	o d s	<	<	o d s	<	f fi =	m	m	o d s	<
MetBy (mi)	< o m di fi	>	d f oi	>	d f oi	>	s si =	>	d f oi	>	mi	mi
Starts (s)	<	>	d	< o m di fi	< o m	oi d f	<	mi	s	s si =	d	< o m
StartedBy (si)	< o m di fi	>	oi d f	di	o di fi	oi	o di fi	mi	s si =	si	oi	di
Finishes (f)	<	>	d	> oi mi di si	o d s	> oi mi	m	>	d	> oi mi	f	f fi =
FinishedBy (fi)	<	> oi mi di si	o d s	di	o	oi di si	m	oi di si	o	di	f fi =	fi

Table 1

The composition table for the twelve Allen temporal relations, adapted from [47]. The `Equal` (=) relation is omitted due to space constraints.

relation between `T1` and `T3` is `r2`, regardless of its value; symmetrically, if `r2` is `Equal`, then the temporal relation between `T1` and `T3` is `r1`, regardless of its value.

The composition table is used by Allen to define a *temporal algebra* for proper intervals. However, the basic ele-

ments of the temporal algebra are not the thirteen basic relations themselves, but rather *vectors* of these relations[17], which represent disjunctions of the thirteen basic relations.

Since the vocabulary of the Time Ontology does not include RDF resources to represent vectors of Allen's temporal relations, the ontology is currently unable to fully implement Allen's temporal algebra. Instead, it only implements the sub-algebra that admits only the thirteen unary vectors corresponding to the thirteen basic properties, which essentially correspond to the properties themselves.

This paper aims to validate the current vocabulary of the Time Ontology, i.e., only the thirteen basic properties. However, as will be discussed below in Section 8, devoted to conclusions and future work, incorporating the full Allen's temporal algebra into the Time Ontology is not straightforward. This not only requires extending the ontology's vocabulary but also necessitates the definition of constraint propagation algorithms that, as demonstrated in [48], exhibit NP-hard complexity.

4. Adding SHACL shapes and SHACL-SPARQL rules to the Time Ontology: relating temporal entities with the instants occurring therein

The previous sections explained that the Time Ontology currently lacks crucial validation checks, e.g., validation checks that temporally compare instants and intervals; this is due to the limitation of OWL, the format in which the Time Ontology is encoded, whose vocabulary does not include constructs to compare and work with temporal data.

On the other hand, temporal comparisons are enabled in SHACL, which incorporates SPARQL 1.1, although only between values of `xsd:dateTime`, the single XSD datatype for which the official SPARQL 1.1 recommendation defines comparison operators. Therefore, this paper proposes a set of SHACL shapes to validate instances of the Time Ontology's resources shown above in Figure 1, i.e., those associated with `xsd:dateTime` values. However, as it will be shown below, SHACL shapes are unable to detect all invalid triples that can be encoded through the vocabulary of the Time Ontology. Therefore, the paper also defines SHACL-SPARQL rules that must be (iteratively) executed before the shapes, for the latter to identify the invalid triples within the *inferred* knowledge graph.

As explained above, Figure 1 is the portion of the Time Ontology related to the `xsd:dateTime` datatype, whose values can be associated with instants through the datatype property `inXSDDateTime`. Therefore, the very first SHACL shape that must be imposed is the one in (3), which flags as "invalid" all objects of `inXSDDateTime` that do not comply with the `xsd:dateTime` datatype.

(3)
```
[rdf:type sh:NodeShape;
 sh:targetObjectsOf time:inXSDDateTime;
 sh:datatype xsd:dateTime;
 sh:message "Invalid datatype: xsd:dateTime is required"].
```

Furthermore, since, as mentioned in footnote 15 above, the timezone is *optional* for the `xsd:dateTime` datatype, for practical reasons we also added the SHACL shape in (4), which requires the objects of `inXSDDateTime` to specify the timezone. The shape in (4) checks whether the timezone is specified or not by using the SHACL Core property `sh:pattern`.[18]

(4)
```
[rdf:type sh:NodeShape;
  sh:targetObjectsOf time:inXSDDateTime;
  sh:pattern "(Z|(\\+|-)[0-9]2:[0-9]2)";
  sh:message "Invalid datatype: it does not specify the timezone."].
```

The two SHACL shapes in (3) and (4) are written in SHACL Core. As explained in the introduction, SHACL Core allows defining simple constraints on RDF resources. However, it is evident that SHACL Core lacks the expressiveness needed for more advanced validation checks.

[17] The term "vector" is introduced by [48]; Allen does not use this term in [47] and [17].

[18] https://www.w3.org/TR/shacl/#PatternConstraintComponent

One such example is the validation check we discuss next: ensuring that if two or more `xsd:dateTime` values are assigned to the same instant through `inXSDDateTime`, these values must be the same. SHACL Core provides the property `sh:maxCount`[19] to impose a limit on the number of values assigned to an RDF resource via a property. However, `sh:maxCount` counts the *occurrences* of these properties. As a result, even if an RDF resource is assigned the same value multiple times through separate occurrences of the same property, each occurrence is counted separately. In other words, SHACL Core does not support constraints that ensure identical values are counted only once.

On the other hand, SPARQL does. As mentioned in the introduction, SPARQL provides operators to compare `xsd:dateTime` values. Therefore, the required validation check can be implemented using the SHACL-SPARQL shape in (5). This ensures that if the same instant appears as the subject of multiple instances of `inXSDDateTime`, then all associated `xsd:dateTime` values must be identical. Otherwise, an error message is returned.

(5)
```
[rdf:type sh:NodeShape;
   sh:targetSubjectsOf time:inXSDDateTime;
    sh:sparql[sh:prefixes ... ;
       sh:select """SELECT $this ?dt1 ?dt2
           WHERE{$this time:inXSDDateTime ?dt1. $this time:inXSDDateTime ?dt2.
                 FILTER(?dt1!=?dt2)}""";
       sh:message "Invalid Instant {$this}: two different xsd:dateTime values
                   are associated with it: {?dt1} and {?dt2}."]].
```

As shown in (5), SHACL-SPARQL shapes incorporate SPARQL queries using the `SELECT-WHERE` structure. Consequently, these shapes tend to be more verbose than SHACL Core shapes. To enhance readability and maintain focus, we will specify only the four key components of SHACL-SPARQL shapes throughout this paper: `sh:target*`, `SELECT`, `WHERE`, and `sh:message`. The shapes in the GitHub repository, being executable, are instead provided in full. For example, the shape in (5) will be represented in a more compact form as:

(6)
```
sh:targetSubjectsOf time:inXSDDateTime;
SELECT $this ?dt1 ?dt2
WHERE{$this time:inXSDDateTime ?dt1. $this time:inXSDDateTime ?dt2.
      FILTER(?dt1!=?dt2)}
sh:message "Invalid Instant {$this}: two different xsd:dateTime values are
            associated with it: {?dt1} and {?dt2}."
```

Having introduced the SHACL shapes to validate the single datatype property occurring in Figure 1, the next subsections introduce SHACL shapes and SHACL-SPARQL rules to validate the object properties occurring therein, except the ones that denote Allen's temporal relations, which will be instead the focus of the next two sections.

4.1. `hasBeginning` *and* `hasEnd`

The properties `hasBeginning` and `hasEnd` of the Time Ontology already provide examples of what has been discussed in the Introduction above, i.e., that validation and inference are intimately connected and, therefore, that in order to properly validate a knowledge graph it is necessary to execute the SHACL shapes on the *inferred* knowledge graph obtained by iteratively applying the SHACL rules to the initial one.

As shown in Figure 1, the two properties connect instances of `TemporalEntity` (domain) with instances of `Instant` (range). The two instants associated with a temporal entity via `hasBeginning` and `hasEnd` respectively denote the start and the end of the temporal entity.

Obviously, a temporal entity cannot begin *after* it ends. Therefore, in cases where the two instants are associated with two different `xsd:dateTime` values, it must be checked that the `xsd:dateTime` value associated with the instant as object of the property `hasBeginning` does not occur *after* the `xsd:dateTime` value associated with the instant as object of the property `hasEnd`. If, instead, it does, the temporal entity is flagged as invalid.

[19] https://www.w3.org/TR/shacl/#MaxCountConstraintComponent

Furthermore, note that nothing in the Time Ontology prevents the same temporal entity to be connected via `hasBeginning` (or via `hasEnd`) to two or more *different* instants. In such a case and if these instants are associated with `xsd:dateTime` values, it must be also checked that these values are the same; if not, the temporal entity is flagged as invalid. For example, the temporal entity `te` in (7) is invalid because it begins at two instants associated each with a *different* `xsd:dateTime` value.

(7)
```
:te time:hasBeginning :t1. :te time:hasBeginning :t2.
:t1 time:inXSDDateTime "2024-01-01T00:00:00Z".
:t2 time:inXSDDateTime "2025-01-01T00:00:00Z".
```

To identify the invalid configurations just discussed, three SHACL shapes, shown in (8), are introduced. The first shape correctly infers the triples in (7) as invalid. The second shape in (8) symmetrically checks that a temporal entity does not end at two different `xsd:dateTime` values. Finally, the third shape checks that a temporal entity does not begin after it ends.

(8)
```
sh:targetSubjectsOf time:hasBeginning;
SELECT $this ?dt1 ?dt2
WHERE{$this time:hasBeginning/time:inXSDDateTime ?dt1.
      $this time:hasBeginning/time:inXSDDateTime ?dt2. FILTER(?dt1!=?dt2)}
sh:message "Invalid TemporalEntity {$this}: it begins at two different
            xsd:dateTime values: {?dt1} and {?dt2}."

sh:targetSubjectsOf time:hasEnd;
SELECT $this ?dt1 ?dt2
WHERE{$this time:hasEnd/time:inXSDDateTime ?dt1.
      $this time:hasEnd/time:inXSDDateTime ?dt2. FILTER(?dt1!=?dt2)}
sh:message "Invalid TemporalEntity {$this}: it ends at two different
            xsd:dateTime values: {?dt1} and {?dt2}."

sh:targetSubjectsOf time:hasBeginning;
SELECT $this ?dtb ?dte
WHERE{$this time:hasBeginning/time:inXSDDateTime ?dtb.
      $this time:hasEnd/time:inXSDDateTime ?dte. FILTER(?dte<?dtb)}
sh:message "Invalid TemporalEntity {$this}: it ends at {?dte}
            but it begins at {?dtb}."]].
```

However, the invalid knowledge graphs that can be encoded through the properties `hasBeginning` and `hasEnd` could be really more complex than the ones just discussed.

Instances of the class `TemporalEntity` might be also instances of the class `Instant`; and, an instant occurring as object of either `hasBeginning` or `hasEnd` might be in turn connected to other instants via *chains* of these properties, but an instant can only begin or end at itself. Figure 3 shows an example of such a pattern.
In Figure 3, while we do not know whether the temporal entity `te` is also an instance of `Instant` rather than of `ProperInterval`, we surely know that `te11`, `te12`, `te13`, `te21`, `te22`, and `te23` are instances of `Instant` because the `RDFS:range` of both properties `hasBeginning` and `hasEnd` is the class `Instant`.

Therefore, the knowledge graph in Figure 3 is actually invalid: as stated earlier, an instant can only begin or end at itself, from which it can be deduced that also `te11` and `te12` occur at `"2024-01-01T00:00:00Z"` and that also `te21` and `te22` occur at `"2025-01-01T00:00:00Z"`. However, `te` cannot begin at two different times.

The first SHACL shape in (8) is unable to detect this invalid pattern because the shape triggers only when the two instants *directly* connected with the temporal entity are associated with different `xsd:dateTime` values.

To solve this problem, a possible solution is to introduce the two SHACL-SPARQL rules shown in (9). The first of these rules searches for chains of `hasBeginning` and `hasEnd` properties that include at least *two* properties and that start with `hasBeginning`. For each chain as such, the rule *directly* connects the first temporal entity in the chain to the last instant in the chain through the property `hasBeginning`. The second rule in (9) does the same for chains starting with the property `hasEnd`.

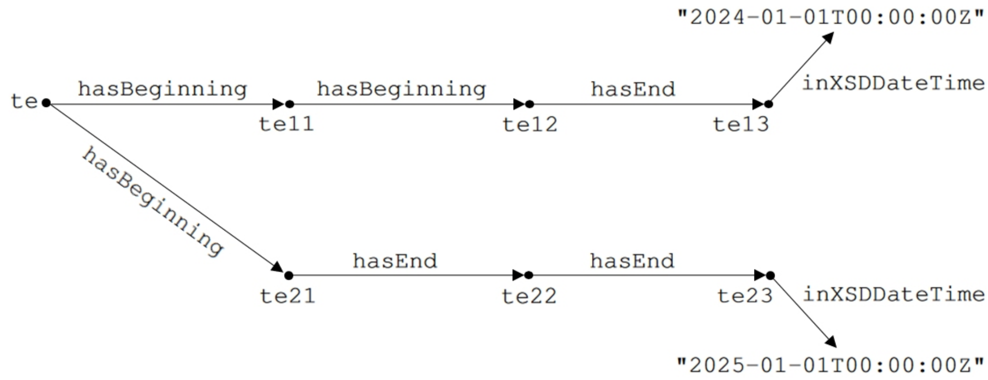

Fig. 3. Example of invalid triples on the properties `hasBeginning`, `hasEnd`, and `inXSDDateTime`.

Throughout the paper, SHACL-SPARQL rules are shown in the same compact notation used for the SHACL shapes, displaying only their three main components: `sh:target*`, `CONSTRUCT` (i.e., the rule's consequent), and `WHERE` (i.e., the rule's antecedent). Full definitions of the rules are available in the GitHub repository.

(9)
```
sh:targetSubjectsOf time:hasBeginning;
CONSTRUCT{$this time:hasBeginning ?i}
WHERE{$this time:hasBeginning/(time:hasBeginning|time:hasEnd)+ ?i}

sh:targetSubjectsOf time:hasEnd;
CONSTRUCT{$this time:hasEnd ?i}
WHERE{$this time:hasEnd/(time:hasBeginning|time:hasEnd)+ ?i}
```

It is easy to see that by executing *first* the SHACL-SPARQL rules in (9) and *then* the SHACL shapes in (8), the triples in Figure 3 are properly inferred as invalid: the first rule in (9) infers and adds to the knowledge graph the `hasBeginning` and `hasEnd` properties shown in blue in Figure 4. Then, the first shape in (8) identifies this knowledge graph as invalid thanks to the triples that now *directly* connect `te` with `te13` and `te23`.

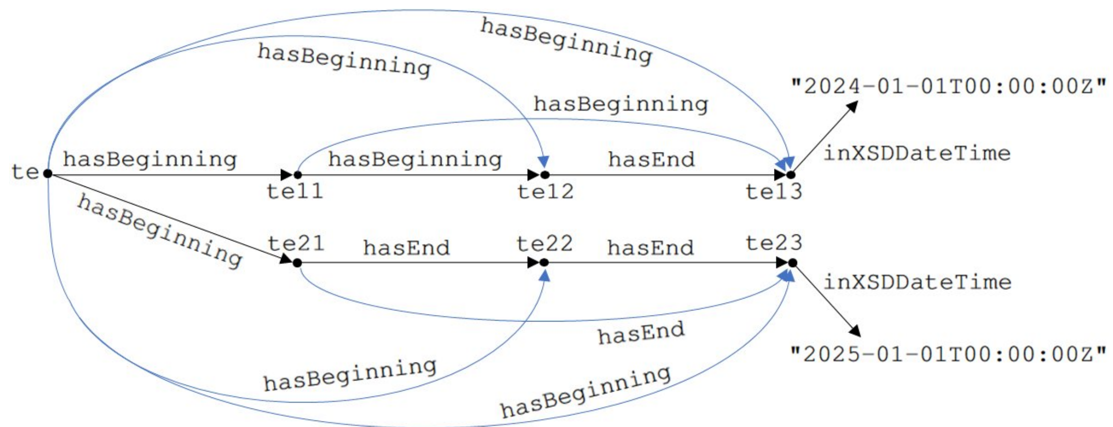

Fig. 4. The knowledge graph inferred from the one in Figure 3 through the rules in (9). The inferred properties are shown in blue.

Alternatively, rather than introducing new SHACL-SPARQL rules that compute the "transitive closure" of the `hasBeginning` and `hasEnd` properties, as the rules in (9) basically do, we could simply evolve the SHACL shapes in (8) by using the SPARQL property path operators[20] fit to identify invalid knowledge graphs such as the

[20] https://www.w3.org/TR/sparql11-query/#propertypaths

one in Figure 3. Specifically, in order to do so, the WHERE clauses of the SHACL shapes in (8) could simply match the property paths matched within the WHERE clauses of the SHACL-SPARQL rules in (9).

Although this solution would indeed work for the knowledge graph shown in Figure 3, the main finding of this paper is that it does *not* always work: the SPARQL property path operators are not expressive enough to identify as invalid some knowledge graphs that can be built with the hasBeginning and hasEnd properties (nor via Allen's interval algebra, as it will be shown below). An example of these invalid knowledge graphs is shown in Figure 5:

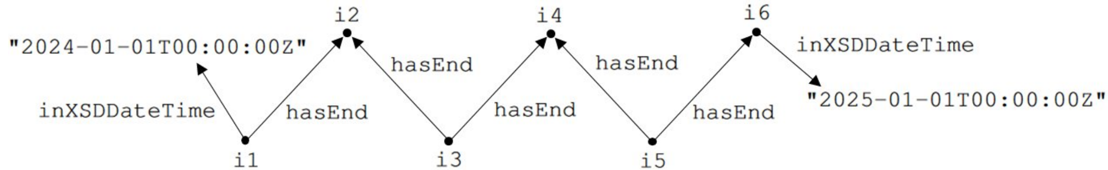

Fig. 5. Example of invalid triples on the properties hasEnd and inXSDDateTime.

From Figure 5, it might be inferred that i1, i2, i4, and i6 are instances of the class Instant: these occur as either subject of the property inXSDDateTime or as object of the property hasEnd. Conversely, i3 and i5 are not necessarily instances of Instant: the RDFS:domain of hasEnd is the more abstract class TemporalEntity.

Nevertheless, it is easy to see that when i3 and i5 are indeed also instances of Instant, e.g., because they are explicitly asserted as such, *then the knowledge graph is invalid*.

In fact, if i5 is an instant that ends at i6, then also i5 occurs at 2025-01-01T00:00:00Z. In light of the same considerations, also i1, i2, i3, and i4 occur at 2025-01-01T00:00:00Z. But, by applying the same line of reasoning in the other direction, all instants also occur at 2024-01-01T00:00:00Z, which is invalid.

There is no way to extend the SHACL shapes in (8) with the SPARQL property path operators fit to recognize "zigzag" patterns such as the one in Figure 5, with an arbitrary number of nodes being all instances of Instant. We could use the operator "^" to cross the properties in backward direction, i.e., from the object to the subject, but that operator does not allow to impose further constraints on the subject, i.e., checking that its type is Instant.

Conversely, in order to properly identify the pattern exemplified in Figure 5 as invalid, the SHACL-SPARQL rule in (10) is introduced. For every instant, this rule searches for chains of hasBeginning or hasEnd that begin at the instant; then, for each of these chains, it infers that the instant at the end of the chain both begins and ends at the instant at the beginning of the chain. Note that the rule in (10) basically parallels the ones in (9) but in backward direction and only for instances of the class Instant.

(10)
```
sh:targetClass time:Instant;
CONSTRUCT{?i time:hasBeginning $this. ?i time:hasEnd $this}
WHERE{$this (time:hasBeginning|time:hasEnd)+ ?i}
```

In addition, in order to identify as invalid knowledge graphs such as the one exemplified in Figure 5, the SHACL-SPARQL rule in (10) *must be iteratively re-executed multiple times*, specifically until no further triple is inferred, because the "zigzag" pattern exemplified in the figure, as pointed out above, could be arbitrarily long.

It is easy to see that by iteratively re-executing the SHACL-SPARQL rules shown in (9) and (10) on the knowledge graph shown in Figure 5, it is eventually inferred that all instants occurring therein begin and end at each other as well as at themselves. After that, the SHACL shapes shown in (8) identify all these instants as invalid due to the different xsd:dateTime values associated with i1 and i6.

On the other hand, by iteratively re-executing the rules in (9) and (10), until no further triple is inferred, on the knowledge graph in Figure 8, it is inferred that the three instants te11, te12, and te13 are the same (i.e., they all begin and end at each other as well as at themselves) and same for the three instants te21, te22, and te23. Nevertheless, a closer look reveals that *all* six instants in Figure 8 are the same, because te cannot begin at two different instants. The rules in (9) and (10) are unable to infer so; therefore, the following rules are added:

(11)
```
sh:targetSubjectsOf time:hasBeginning;
CONSTRUCT{?i1 time:hasBeginning ?i2}
WHERE{$this time:hasBeginning ?i1. $this time:hasBeginning ?i2}

sh:targetSubjectsOf time:hasBeginning;
CONSTRUCT{?i1 time:hasEnd ?i2}
WHERE{$this time:hasEnd ?i1. $this time:hasEnd ?i2}
```

We conclude this section with the SHACL-SPARQL rule in (12), which is the dual of the one in (10). The rule in (12) infers a temporal entity as instance of the class `Instant` if either it begins at the same instant in which it ends or it begins and ends at two different instants associated with the same `xsd:dateTime` value.

(12)
```
sh:targetSubjectsOf time:hasBeginning;
CONSTRUCT{$this a time:Instant}
WHERE{{$this time:hasBeginning ?b. $this time:hasEnd ?b}UNION
      {$this time:hasBeginning/time:inXSDDateTime ?dtb.
       $this time:hasEnd/time:inXSDDateTime ?dte. FILTER(?dtb=?dte)}}
```

4.2. `before` *and* `after`

This subsection presents SHACL shapes and SHACL-SPARQL rules to validate the properties `before` and `after`, which connect pairs of instances of `TemporalEntity`. Actually, this subsection only focuses on the property `before`. On the other hand, since `after` is the inverse property of `before` and, as explained in (1) above, our implementation includes SHACL-SPARQL rules that infer all inverse properties, there is no need to introduce further SHACL shapes on the property `after`, symmetric to the ones defined for the property `before`: the SHACL shapes defined on the property `before` also cover the property `after`.

The first SHACL shape proposed in this section is shown in (13). This SHACL shape validates pairs of instants associated with specific `xsd:dateTime` values and connected through a chain of properties `before` plus, optionally, the properties `hasBeginning` and `hasEnd`; the latter are also considered in the chain in order to identify invalid knowledge graphs such as the one shown in Figure 6. The `xsd:dateTime` values of the two instants must of course comply with the temporal order denoted by the property `before`.

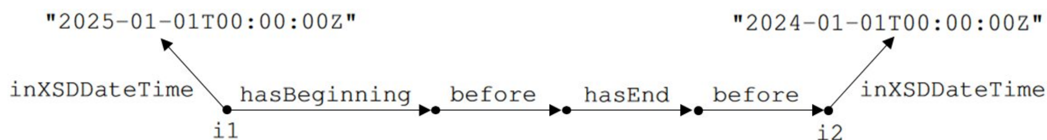

Fig. 6. Invalid knowledge graph involving the properties `before`, `hasBeginning` and `hasEnd`.

(13)
```
sh:targetSubjectsOf time:before;
sh:select SELECT $this ?i ?dt1 ?dt2
WHERE{$this time:before ?i.
   $this(^time:before|^time:hasBeginning|^time:hasEnd)*/time:inXSDDateTime ?dt1.
   ?i(time:before|time:hasBeginning|time:hasEnd)*/time:inXSDDateTime ?dt2.
   FILTER(?dt1>=?dt2)}""";
sh:message "Invalid triple '{$this} time:before {?i}': it connects the
            xsd:dateTime values {?dt1} and {?dt2}."
```

However, the property `before` also establishes *qualitative* partial orders between the connected temporal entities, even when these are not associated with any `xsd:dateTime` value. Since partial orders correspond to directed

acyclic graphs, it is necessary to define a SHACL shape that invalidates knowledge graphs containing *cyclic chains* of `before` properties (and, again, optionally, `hasBeginning` and `hasEnd`).[21] This shape is shown in (14):

(14)
```
sh:targetSubjectsOf time:before;
SELECT $this ?i
WHERE{$this time:before ?i.
      ?i (time:before|time:hasBeginning|time:hasEnd)+ $this}
sh:message "Invalid triple '{$this} time:before {?i}':
            it is part of a cycling path."
```

Finally, we need a shape that parallels the third one in (8), namely a shape that invalidates temporal entities that end *before* they begin. This shape is shown in (15); the difference between (15) and the third shape in (8) is that the former *qualitatively* checks that a temporal entity does not end before it begins, while the latter does so *quantitatively*.

(15)
```
sh:targetSubjectsOf time:before;
SELECT $this ?te
WHERE{$this time:before/(time:before|time:hasBeginning|time:hasEnd)* ?t.
      ?te time:hasEnd $this. ?te time:hasBeginning ?t}
sh:message "Invalid TemporalEntity {?te}: it ends before it begins."
```

As it will be discussed below in Subsubsection 4.4.2, the three SHACL shapes in (14), (13), and (15) are enough to validate the property `before`, provided that we also introduce SHACL-SPARQL rules that, whenever some instants occur either within the subject or within the object of `before`, infer that the property `before` also holds between these instants and the temporal entities in which they occur.

As observed earlier, the subject and the object of `before` are instances of `TemporalEntity`. Therefore, as shown in Figure 7, they may occur as subject of other properties `hasBeginning`, `hasEnd`, and `inside`; in such cases, it is clear that: (1) every instant connected through these three properties to the subject of `before` also occurs before its object as well as any instant included therein and (2) the subject of `before` does not only occur before its object but also before any instant included therein.

Thanks to the SPARQL operators `BIND` and `UNION`, the inferences shown in Figure 7 can be carried out via a single SHACL-SPARQL rule, shown in (16). This rule infers that the property `before` holds for the Cartesian product {`te1`, `i11`, `i12`, `i13`}×{`te2`, `i21`, `i22`, `i23`}.

(16)
```
sh:targetSubjectsOf time:before;
CONSTRUCT{?i1 time:before ?te2. ?i1 time:before ?i2}
WHERE{$this time:before ?te2.
 {BIND($this AS ?i1)}UNION{$this (time:hasBeginning|time:inside|time:hasEnd) ?i1}
 {BIND(?te2 AS ?i2)}UNION {?te2 (time:hasBeginning|time:inside|time:hasEnd) ?i2}}
```

Two final inferences for the property `before` must be considered. As shown in Figure 8, in cases where the instant that *ends* a temporal entity occurs before another temporal entity, it is possible to infer that *the whole* temporal entity ending at that instant does. Symmetrically, if the instant that *begins* a temporal entity occurs after another temporal entity, it is possible to infer that *the whole* temporal entity beginning at that instant does. After that, the previous rules will infer that the property `before` also occurs between *every* instant occurring in the two involved temporal entities and not only the ones that end the former and begin the latter.

The inferences depicted in Figure 8 are carried out by the following SHACL-SPARQL rules:

(17)
```
sh:targetSubjectsOf time:before;
CONSTRUCT{?te1 time:before ?te}
WHERE{$this time:before ?te. $this ^time:hasEnd ?te1}
```

[21] It is worth noting that an unofficial extension of SHACL Core, called DASH, introduces a property to specify that a property or path must not reference itself, i.e., it must not form cyclic paths. For more details, see https://datashapes.org/constraints.html#NonRecursiveConstraintComponent.

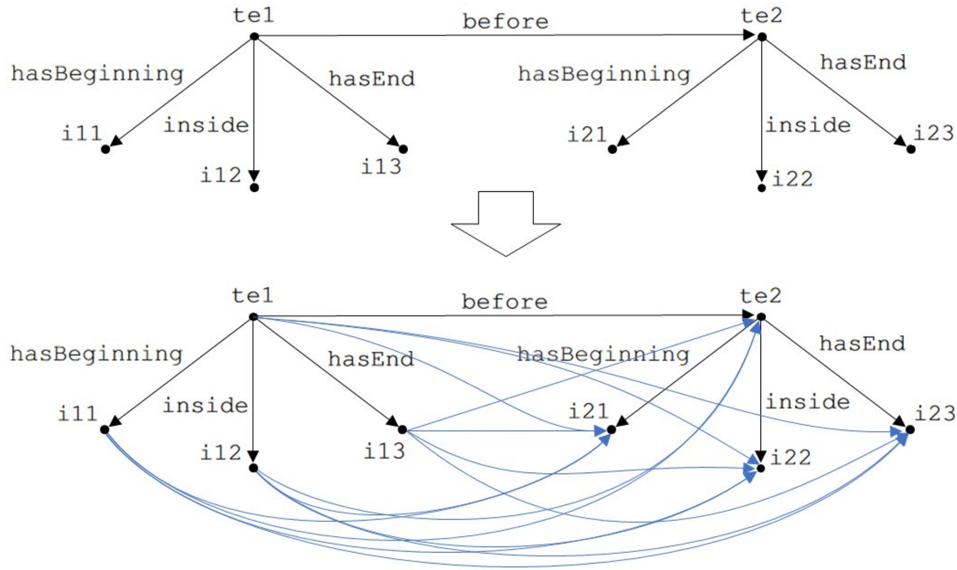

Fig. 7. Inferences on the property `before`; the inferred properties `before` are shown in blue.

Fig. 8. Inferences on the property `before`; the inferred properties are shown in blue.

```
sh:targetSubjectsOf time:before;
CONSTRUCT{$this time:before ?te2}
WHERE{$this time:before ?te. ?te ^time:hasBeginning ?te2}
```

4.3. `inside`

This subsection presents SHACL shapes and rules to validate the object property `inside`, which connects instances of `Interval` (domain) with instances of `Instant` (range).

The definition of the property `inside` in the Time Ontology specifically states that an instant occurring as its object is "An instant that falls inside the interval. It is not intended to include beginnings and ends of intervals.". Therefore, if an interval is connected to two instants through the properties `hasBeginning` and `hasEnd` respectively, these two instants do *not* belong to the set of instants inside the interval.

From this, it might be also deducted that the `RDFS:domain` of `inside` is actually `ProperInterval`, and not the more abstract class `Interval`, which also subsumes `Instant`. In other words, an instance of `Instant` cannot be the subject of the property `inside`: an instant begins and ends at itself, therefore no other instant can occur in between. In light of these considerations, the `RDFS:domain` of `inside` should be perhaps set to `ProperInterval`; alternatively, the following SHACL Core shape should be asserted:

(18)
```
sh:targetSubjectsOf time:inside;
sh:not [sh:class time:Instant];
sh:message "Invalid Interval {$this}: the Interval as subject of the
            property ‘inside’ is also an Instant. Nevertheless, instants
            cannot (properly) contain other instants."
```

Apart from (18), we should now define further SHACL shapes that, in cases where the subject of `inside` also occurs as subject of `hasBeginning` and `hasEnd`, (1) check that neither of the objects of `hasBeginning` and `hasEnd` is also the object of `inside` and (2) in cases where these are indeed different instants, check that if the three instants are associated with `xsd:dateTime` values, the one associated with the object of `inside` respectively occurs after and before the ones associated with the objects of `hasBeginning` and `hasEnd`.

Four SHACL shapes would be needed to validate (1) and (2). However, similarly to what has been done in the previous subsection for the property `after`, rather than defining *four* shapes, we found more convenient to define the *two* SHACL-SPARQL rules in (19). These two rules check whether the property `hasBeginning` and `hasEnd` are also defined on the subject of `inside`; if so, they respectively infer that the object of `hasBeginning` occurs before the one of `inside` and that the object of `inside` occurs before the one of `hasEnd`. Thus, the SHACL shapes on the property `before`, presented in the previous subsection, validate the triples so inferred.

(19)
```
sh:targetSubjectsOf time:inside;
CONSTRUCT{?b time:before ?i}
WHERE{$this time:hasBeginning ?b. $this time:inside ?i}

sh:targetSubjectsOf time:inside;
CONSTRUCT{?i time:before ?e}
WHERE{$this time:hasEnd ?e. $this time:inside ?i}
```

4.4. Evaluation

The previous subsections introduced SHACL shapes and SHACL-SPARQL rules to validate the Time Ontology's properties `hasBeginning`, `hasEnd`, `inside`, `before`, and `after`. This subsection further investigates, through a case-based evaluation, whether the proposed shapes and rules are able to properly validate all knowledge graphs that can be built by combining these properties in all possible ways. The analysis below indeed allowed us to identify preliminary versions of the shapes and the rules that were either unable to invalidate some invalid knowledge graphs or that were wrongly invalidating some valid knowledge graphs. We therefore incrementally redesigned the shapes and rules until we achieved the final versions reported above.

4.4.1. `hasBeginning` *and* `hasEnd`

The properties `hasBeginning` and `hasEnd` connect instances of `TemporalEntity` with instances of `Instant`. Therefore, every individual occurring as object of these two properties is inferred as instance of `Instant` through the SHACL-SPARQL rule that implements `RDFS:range` (omitted from this paper but available in the GitHub repository, as explained in (1) above).

Concerning the individuals occurring as subject of the `hasBeginning` and `hasEnd`, it is convenient to distinguish cases when these individuals are also instances of `Instant` from cases when they are not, e.g., when they are instances of either `ProperInterval` or its superclasses `Interval` or `TemporalEntity`.

Let's first address the cases in which they are also instances of `Instant`, i.e., when they are either explicitly asserted as such or when they occur as subject of `inXSDDateTime`, as object of `inside`, or as object of other properties `hasBeginning` or `hasEnd`; in the second cases, exemplified in Figure 9, the SHACL-SPARQL rules implementing `RDFS:domain` and `RDFS:range` infer them as instances of `Instant`.

When also the subject of `hasBeginning` or `hasEnd` is an instance of the class `Instant`, the SHACL-SPARQL rules shown above in (9) and (10) infer that the subject and the object of the two properties indeed denote the same instant, because these rules infer that they begin and end at one another and at themselves.

Therefore, only two patterns must be checked in Figure 9: (1) when the two instants are associated with *different* `xsd:dateTime` values via the property `inXSDDateTime` and (2) when either they are associated with the same `xsd:dateTime` value or when only one of the two is associated with a `xsd:dateTime` value. (1) is invalid while (2) is not. This result is indeed achieved by the shapes introduced above in Subsection 4.1; specifically, the shapes in (8) invalidate the pattern in (1) while no shape invalidates the one in (2).

Let's now consider the cases in which the subject of `hasBeginning` or `hasEnd` is not an instance of `Instant` but rather an instance of the more abstract class `TemporalEntity`. This may hold when the subject of

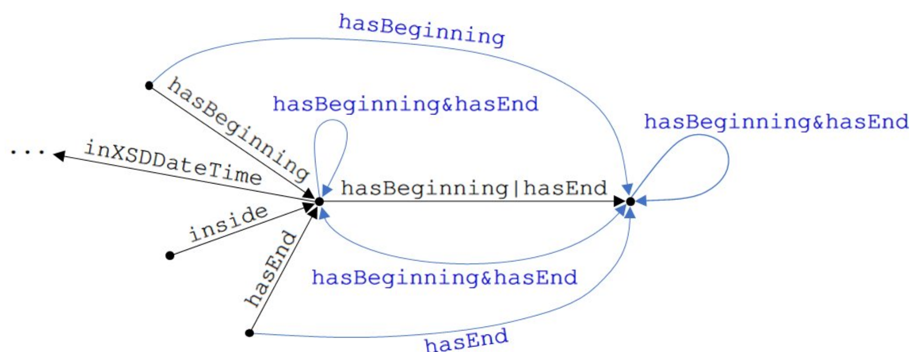

Fig. 9. Inferences on the property `hasBeginning` or `hasEnd`; the inferred properties are shown in blue.

`hasBeginning` or `hasEnd` either (1) occurs as either subject or object of `before`, `after`, or the properties denoting Allen's temporal relations, (2) occurs as subject of the property `inside`, or (3) occurs as subject of *other* properties `hasBeginning` or `hasEnd`.

The patterns in (1) and (2) will be discussed in the next subsubsections, thus we will not consider them in this subsection as well. Concerning (3), three sub-patterns are possible: when both properties are `hasBeginning`, as shown in Figure 10 on the left, when both properties are `hasEnd`, as shown in Figure 10 in the middle, and when one of the two properties is `hasBeginning` and the other one is `hasEnd`, as shown in Figure 10 on the right.

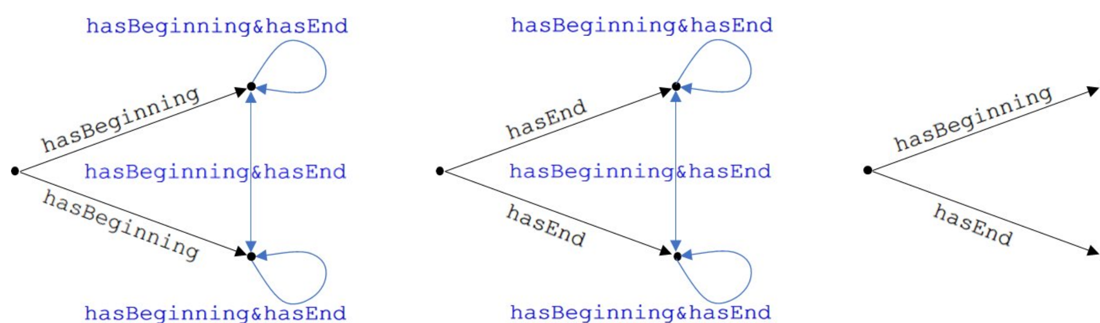

Fig. 10. Inferences on the property `hasBeginning` or `hasEnd`; the inferred properties are shown in blue.

In Figure 10 on the left and in the middle, the SHACL-SPARQL rules in (11) infer that the objects of the two properties denote the same instant; therefore, in cases where these are associated with different `xsd:dateTime` values or in cases where they are connected through the property `before` (or the property `after`), either the shapes in (8) or the one in (15) will report them as invalid.

Finally, in Figure 10 on the right, the pattern is invalid only if the temporal entity ends before it begins. This can be asserted *quantitatively*, i.e., when the objects of the two properties are both associated with `xsd:dateTime` values and the one associated with the object of `hasEnd` is lower than the one associated with the object of `hasBeginning`, or *qualitatively*, i.e., when the object of `hasEnd` occurs before the object of `hasBeginning`. These two patterns are respectively invalidated by the third shape in (8) and by the shape in (15).

4.4.2. `before` *and* `after`

Section 4.2 above introduced three SHACL shapes on the property `before` and a single SHACL-SPARQL rule on the property `after`. The latter converts every triple on the property `after` into the inverse triple on the property `before`; this rule avoids to introduce three further SHACL shapes on the property `after`, symmetric to the ones defined on the property `before`. The three SHACL shapes defined on the property `before` are:

- The shape in (13), which searches in the knowledge graph for chains of properties `before` (and, optionally, `hasBeginning` and `hasEnd`) connecting two instants associated both with `xsd:dateTime` values, and *quantitatively* checks that these values comply with the temporal order denoted by `before`.

- The shape in (14), which *qualitatively* checks that the knowledge graph does not contain *cycling* chains of properties `before` (and, optionally, `hasBeginning` and `hasEnd`).
- The shape in (15), which *qualitatively* checks that the knowledge graph does not contain temporal entities that end before they begin.

Contrary to the properties `hasBeginning` and `hasEnd`, both the subject and the object of the property `before` are instances of `TemporalEntity`; in other words, they can be either instants or intervals. Therefore, in order to evaluate the shapes in (13), (14), and (15), rather than reasoning on the possible classes to which the subject or the object of `before` may belong, it is convenient to consider in which other possible properties these subject and object may be also involved. Specifically, these four cases must be considered:

(20) a. When the subject of `before` occur as subject of other properties.
b. When the subject of `before` occur as object of other properties.
c. When the object of `before` occur as subject of other properties.
d. When the object of `before` occur as object of other properties.

In (20.a), as shown in Figure 11 on the left, when the subject of `before` is connected to other temporal entities through another property `before`, there are two *independent* chains of `before` (and, optionally, `hasBeginning` and `hasEnd`). Nothing can be said about the temporal order between the two objects of `before` or the other temporal entities connected to them. Consistently with this observation, the shapes in (13), (14), and (15) do not invalidate these patterns because they only consider the beginning and the end of *single* chains of `before` (and, optionally, `hasBeginning` and `hasEnd`). On the other hand, when the subject of `before` is connected to other instants via `hasBeginning`, `inside`, and `hasEnd`, the rule in (16) infers that these instants occur before the object of `before`; the validation of the asserted property `before` is then made equivalent to the validation of these inferred properties. For example, if both the object of the initial property `before` and the instants that either begin, end, or occur inside its subject are associated with `xsd:dateTime` values but the ones associated with the latter are equal or superior to the one associated with the former, then the shape in (13) detects the invalid pattern. Another example is shown in Figure 11 on the right: if the instants that begin, end, or occur inside the subject of `before` are in turn connected with other instants such that at the end of the chains there is the object of the initial property `before`, then the SHACL-SPARQL rule in (16) is *iteratively* applied until a cycle is inferred between the object of the initial property `before` and itself; this cycle is then invalidated by the SHACL shape in (14).

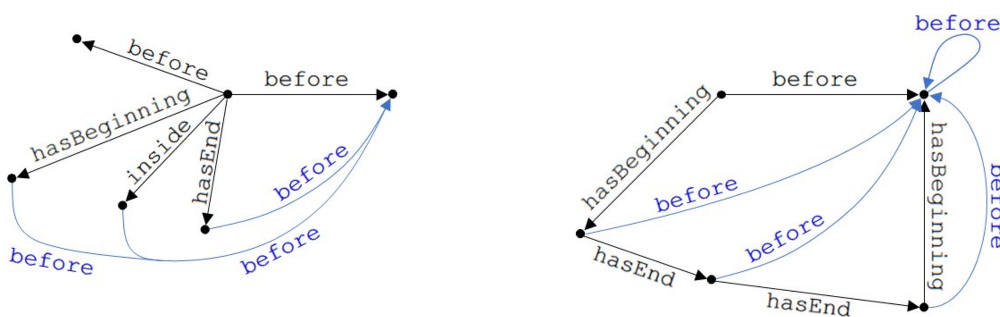

Fig. 11. Sample knowledge graphs involving `before`, `hasBeginning`, `hasEnd`, and `inside`; the properties in blue are inferred.

Let us now consider the case in (20.a). When the property `before` comes after another property `before` or a property `hasBeginning`, as shown in Figure 12 on the left, then the knowledge graph is invalid only when: (1) the involved temporal entities are all instants and they are all associated with `xsd:dateTime` values that do not comply with the temporal order denoted by `before` or (2) the object of the property `before` is the subject of the properties that precede it, i.e., when there is a cycling. The two shapes in (13) and (14) respectively detect these invalid patterns because they apply to any chain of properties `before`, `hasBeginning`, and `hasEnd` that

includes at least one property `before`. On the other hand, the case in which the property `before` comes after the property `inside`, also shown in Figure 12 on the left, is even simpler to understand: the temporal relation between the interval as subject of `inside` and the temporal entity as object of `before` cannot be determined: the object of `inside` is an instant that occurs before that temporal entity, but this does not tell anything about the temporal relation between that temporal entity and the whole temporal entity that contains the instant as object of `inside`. For that reason, we did not consider the property `inside` in the SHACL shapes in (13) and (14): if a property `inside` occurs in a chain of properties `before` (and, optionally `hasBeginning` and `hasEnd`), there are actually *two* chains of properties to be (separately) validated, as the property `inside` "cuts" the chain in two.

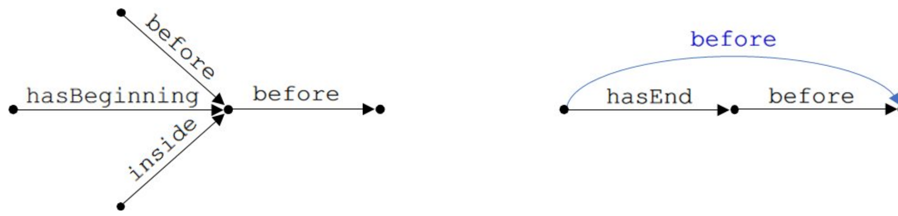

Fig. 12. Sample knowledge graphs involving `before`, `hasBeginning`, `hasEnd`, and `inside`; the property in blue is inferred.

The case in which the property `before` comes after the property `hasEnd`, as shown in Figure 12 on the right, is different. As explained at the end of Subsection 4.2, if a temporal entity comes after the end of another temporal entity, then it also comes after every instant occurring in the latter. For these reasons, the first SHACL-SPARQL rule shown in (17) above infers the property `before` shown in blue in Figure 12 on the right, i.e., it infers that the whole temporal entity as subject of `hasEnd` occurs before the object of `before`.

Without the SHACL-SPARQL rules in (17), several invalid knowledge graphs would not be invalidated. An example of these invalid knowledge graphs is shown in Figure 13. The beginning instant of a temporal entity cannot occur *after* an instant that occurs after the end of the same temporal entity. This is exactly the case in Figure 13, because the subject of `hasBeginning` is associated with an `xsd:dateTime` value superior to the one associated with the object of `before`. The SHACL shapes in (13), (14), and (15) alone are unable to detect this invalid pattern. However, by inferring the property `before` between the temporal entity as subject of `hasEnd` and the instant as object of `before`, the SHACL-SPARQL rule shown above in (16) infers that every instant occurring within that temporal entity, including the one that begins it, also occurs before the instant as object of `before`. Thus, the SHACL shape in (13) detects the invalid knowledge graph.

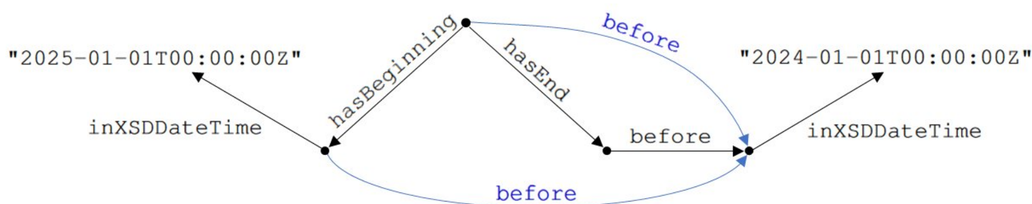

Fig. 13. Invalid knowledge graph involving `before`, `hasBeginning`, `hasEnd`, and `inXSDDateTime`; the properties in blue are inferred.

The case in (20.c) is the simplest one to validate. As for the case in (20.b) and as shown in Figure 14, if the object of the property `before` occurs as subject of other properties `before`, `hasBeginning`, or `hasEnd`, the SHACL shapes in (13), (14), and (15) invalidate knowledge graphs in which the objects of these properties either form a cycle or are associated with `xsd:dateTime` values that do not comply with the temporal order denoted by the property `before`. The reason is that, as explained above, these shapes consider chains that include at least one property `before` and an optional arbitrary number of properties `before`, `hasBeginning`, and `hasEnd`. In addition, the SHACL-SPARQL rule in (16) infers that the property `before` also holds between the subject of the asserted property `before` and the objects of `hasBeginning`, `inside`, and `hasEnd` properties, thus making the validation of the asserted property `before` equivalent to the validation of these inferred properties.

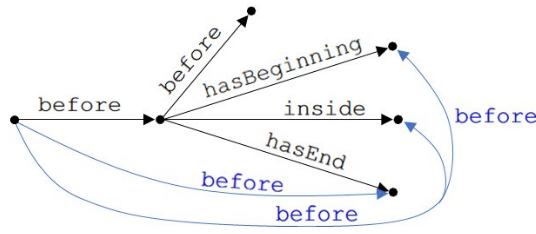

Fig. 14. Invalid knowledge graph involving `before`, `hasBeginning`, `hasEnd`, and `inXSDDateTime`; the properties in blue are inferred.

Finally, we consider the case in (20.d), in which the object of the property `before` also occurs as object of other properties. As shown in Figure 15 on the left, the knowledge graph is valid in the general case, because no temporal order is established between the objects of the two involved properties. Indeed, none of the SHACL shapes and the SHACL-SPARQL rules trigger in these general patterns: the shapes only consider single chains of properties `before` (and, optionally, `hasBeginning` and `hasEnd`) while the single rule that triggers is the second one shown above in (17), which connects the subject of the property `before` with the subject of the property `hasBeginning`; thanks to this inferred property we achieve validations symmetric to the ones already commented above in Figure 13, associated with the *first* rule shown above in (17).

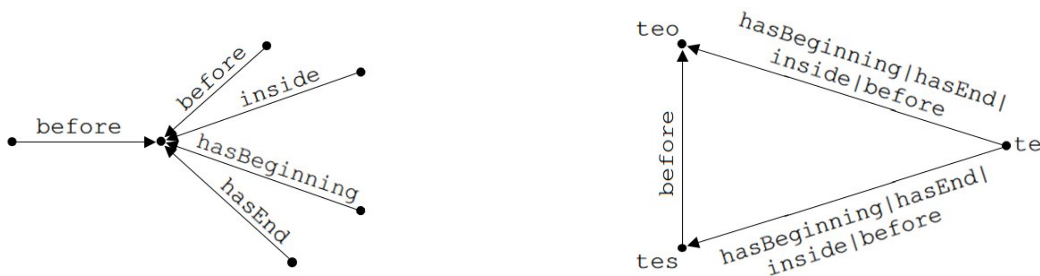

Fig. 15. Knowledge graph involving `before`, `hasBeginning`, `hasEnd`, and `inside`.

Nevertheless, there is an exception to the general case that deserves further investigation: when also the subject of the property `before` occurs as object of the other properties, as exemplified in Figure 15, on the right. Five sub-cases need to be distinguished, depending on the specific properties involved:

(21) a. When `te` is connected to `teo` via the property `before`, the knowledge graph is invalid unless also the property connecting `te` to `tes` is `before`. This invalid pattern is indeed detected, because the rule in (16) infers that also `teo` is connected to `tes` through the property `before` and so the shape in (14) detects the cycling path. All other patterns in which at least two properties `before` occur are valid and, indeed, the defined shapes and rules acknowledge them as such.

b. Both properties from `te` to `tes` and `teo` are either `hasBeginning` or `hasEnd`. In such cases, the knowledge graphs are invalid: a temporal entity cannot begin or end at two instants such that one of them occurs before the other. These invalid knowledge graphs are detected: the rules in (11) infer that the subject and the object of `before` are the same instant, the rule in (16) infers that `tes` or `teo` occur before themselves, and the shape in (14) detects the cycling path.

c. Both properties from `te` to `tes` and `teo` are `inside`. The knowledge graph is valid; we know that one of the two instants inside the interval occurs before the other one but no other information contradicts so. The rules and shapes defined do not infer the knowledge graph as invalid, which is correct.

d. Either (1) `tes` is the object of the property `hasBeginning` while `teo` is the object of the property `inside` or the property `hasEnd`, or (2) `tes` is the object of the property `inside` while `teo` is the

object of the property `hasEnd`. The knowledge graph is valid, and indeed the rules and shapes defined do not infer it as invalid, which is correct.

e. Either (1) `tes` is the object of the property `hasEnd` while `teo` is the object of the property `inside` or the property `hasBeginning`, or (2) `tes` is the object of the property `inside` while `teo` is the object of the property `hasBeginning`. This patters are the opposite of the ones considered in the previous point, so that they are all invalid: an interval cannot end before it begins while an instant inside this interval cannot occur before the beginning of the interval or before its end. The shape in (15) properly invalidates the case in which `hasEnd` and `hasBeginning` are involved; on the other hand, in the two cases involving the property `inside`, the rules in (19) infer the opposite properties `before`, which denote the right temporal order, so that the shape in (14) detects the cycling paths and invalidates the knowledge graph.

4.4.3. `inside`

The property `inside` has been basically already evaluated in the previous subsubsection, because it can lead to invalid knowledge graphs only when it is used together with the property `before`.

The SHACL shape shown above in (18) imposes the subject of the property `inside` to be a proper interval. This shape contributed to reduce the possible patterns considered in the evaluation: only those that have a proper interval in the subject of `inside`. The shape in (18) has been introduced to comply with the definition of the property in the Time Ontology: an instant cannot properly contain other instants, because an instant can only begin and end at itself. As explained in Subsection 4.3, this shape can be avoided by setting the `RDFS:domain` of `inside` to `ProperInterval`, and we are indeed suggesting so for the future releases of the Time Ontology.

Apart from the patterns invalidated by the shape in (18), the only other invalid patterns that involve the property `inside` are those in which the interval as its subject either begins after or ends before the instant as its object.

Rather than defining specific shapes to invalidate these patterns, we chose to define the two SHACL-SPARQL rules shown above in (19). As shown in Figure 16, in cases where the subject of `inside` also occurs as subject of `hasBeginning` and/or `hasEnd`, the two rules infer additional properties `before` that denote the correct temporal order. Then, the shapes defined on `before` invalidate the knowledge graph in cases where this contains further RDF triples that do not comply with that temporal order. In particular, the shape in (13) *quantitatively* invalidates potential `xsd:dateTime` values, associated with the two instants, that do not comply with that temporal order; conversely, as already explained above in (21.e), the shape in (14) *qualitatively* invalidates knowledge graphs that, after new inferred properties `before` have been added therein, contain a cycling path.

Fig. 16. Sample knowledge graphs of `hasBeginning`, `hasEnd`, and `inside` properties; the `before` properties in blue are inferred.

5. Adding SHACL shapes and SHACL-SPARQL rules to the Time Ontology: relating Allen's temporal relations with the instants occurring within the involved (proper) intervals

This and the next two sections introduce SHACL shapes and SHACL-SPARQL rules for validating the Time Ontology's properties that denote Allen's temporal relations. Specifically, this section examines how these properties interact with `hasBeginning`, `hasEnd`, and `before`, discussed in the previous section. The following two sections will then focus exclusively on the properties that denote Allen's temporal relations.

Since both the `rdfs:domain` and the `rdfs:range` of all properties denoting Allen's temporal relations are set to the class `ProperInterval`, the first SHACL-SPARQL rule that we introduce is the one in (22), which specifies that every instant beginning a proper interval occurs before the instant that ends it:

(22)
```
sh:targetClass time:ProperInterval;
CONSTRUCT{?b time:before ?e}
WHERE{$this time:hasBeginning ?b. $this time:hasEnd ?e}
```

Therefore, for every temporal entity classified as `ProperInterval` via SHACL-SPARQL rules implementing `rdfs:domain` and `rdfs:range`, the `time:before` property is inferred between its starting and ending instants. This inference may, in turn, trigger some of the rules and shapes introduced in the previous section.

The remainder of this section presents further similar SHACL-SPARQL rules. Depending on the specific property representing one of Allen's temporal relations and linking two proper intervals, these rules will infer the properties `hasBeginning`, `hasEnd`, and `before` between the two intervals or their associated instants.

Once these rules have been iteratively applied until no further triples can be inferred, the SHACL shapes defined in the previous section will validate the resulting knowledge graph.

Nevertheless, as explained below, the properties `intervalIn` and `intervalDisjoint` also require the introduction of specific SHACL shapes. Notably, these two properties are the only ones among those considered that do not correspond to one of the thirteen basic Allen's temporal relations.

5.1. `intervalBefore` *and* `intervalAfter`

`intervalBefore` and `intervalAfter` are respectively sub-properties of `before` and `after`, which hold between instances of the more abstract class `TemporalEntity`. Since our implementation includes SHACL-SPARQL rules that implement `RDFS:subPropertyOf` (which this paper nonetheless omits, as explained in (1) above), every triple involving the properties `intervalBefore` or `intervalAfter` is inferred as a triple involving the properties `before` or `after`, respectively.

Conversely, the other way round can be inferred only when the two temporal entities related through the property `before` or `after` are also instances of `ProperInterval`. We therefore add the following SHACL-SPARQL rule to our implementation on GitHub:

(23)
```
sh:targetSubjectsOf time:before;
CONSTRUCT{$this time:intervalBefore ?pi2}
WHERE{$this time:before ?pi2; a time:ProperInterval. ?pi2 a time:ProperInterval}
```

Given these implications between `intervalBefore`/`intervalAfter` and `before`/`after`, there is no need to introduce further shapes and rules on the former: the ones defined on the latter also cover for the former.

5.2. `intervalStarts` *and* `intervalStartedBy`

These two properties are the inverse of one another. In line with what has been done above with `before` and `after`, their relation with the properties `hasBeginning`, `hasEnd`, and `before` is defined only in terms of one of them (`intervalStarts`), while every triple asserted on `intervalStartedBy` is converted into the inverse triple asserted on `intervalStarts`.

In the Time Ontology, the property `intervalStarts` is defined as follows: "If a proper interval `T1` is `intervalStarts` another proper interval `T2`, then the beginning of `T1` is coincident with the beginning of `T2`, and the end of `T1` is before the end of `T2`". The first part of the definition is implemented via the SHACL-SPARQL rule in (24): if both intervals occur as subject of `hasBeginning`, then it is inferred that the two objects of `hasBeginning` are indeed the same instant, i.e., they begin and end at one another.

(24)
```
sh:targetSubjectsOf time:intervalStarts;
CONSTRUCT{?b1 time:hasBeginning ?b2. ?b2 time:hasBeginning ?b1.
          ?b1 time:hasEnd ?b2. ?b2 time:hasEnd ?b1}
WHERE{$this time:intervalStarts ?te2.
      $this time:hasBeginning ?b1. ?te2 time:hasBeginning ?b2}
```

Concerning the second part of the definition, note that if the *end* of the interval as subject of `intervalStarts` occurs before the end of its object, then it is *the whole interval* as subject that actually occurs before the end of its object. The SHACL-SPARQL rule in (25) implements this inference. Then, the rule shown above in (16) will in turn infer that also any instant within that interval occurs before the end of `intervalStarts`'s object.

(25)
```
sh:targetSubjectsOf time:intervalStarts;
CONSTRUCT{$this time:before ?e2}
WHERE{$this time:intervalStarts/time:hasEnd ?e2}
```

Finally, a further pattern involving the property `intervalStart` must be considered. Suppose, as depicted in Figure 17, that the object of `intervalStarts` also occurs as subject of the property `hasBeginning`, while its subject does not. The SHACL-SPARQL rule in (24) does not trigger, because it requires *both* intervals to be connected to their beginning instants. Still, it should be possible to infer that the beginning of the `intervalStarts`'s object occurs before every instant that either ends or occurs inside the `intervalStarts`'s subject. This inference is carried out by the SHACL-SPARQL rule in (26):

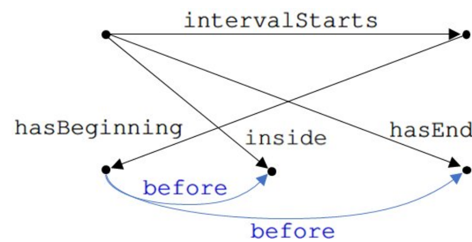

Fig. 17. Inference on the properties `intervalStarts`, `hasBeginning`, `inside`, and `hasEnd`. The inferred properties are shown in blue.

(26)
```
sh:targetSubjectsOf time:intervalStarts;
CONSTRUCT{?b1 time:before ?i2}
WHERE{$this time:intervalStarts/(time:hasEnd|time:inside) ?i2.
      $this time:hasBeginning ?b1}
```

5.3. `intervalOverlaps` *and* `intervalOverlappedBy`

These two properties are also the inverse of one another. Therefore, similarly to what has been done in the previous subsection, our formalization focuses on the property `intervalOverlaps` because every triple on `intervalOverlappedBy` is converted into the inverse triple on `intervalOverlaps`.

In the Time Ontology, the property `intervalOverlaps` is defined as follows: "If a proper interval `T1` is `intervalOverlaps` another proper interval `T2`, then the beginning of `T1` is before the beginning of `T2`, the end of `T1` is after the beginning of `T2`, and the end of `T1` is before the end of `T2`.". This definition is implemented by the three SHACL-SPARQL rules in (27). In light of the same observations made above for the rule in (25), note that the first of the rules in (27) considers the whole interval as object of `intervalOverlaps`: if a temporal entity occurs before the beginning of an interval, then it occurs before the *whole* interval.

(27)
```
sh:targetSubjectsOf time:intervalOverlaps;
CONSTRUCT{?b1 time:before ?te2}
WHERE{$this time:intervalOverlaps ?te2. $this time:hasBeginning ?b1}

sh:targetSubjectsOf time:intervalOverlaps;
CONSTRUCT{?b2 time:before ?e1}
WHERE{$this time:intervalOverlaps/time:hasBeginning ?b2. $this time:hasEnd ?e1}

sh:targetSubjectsOf time:intervalOverlaps;
CONSTRUCT{?e1 time:before ?e2}
WHERE{$this time:intervalOverlaps/time:hasEnd ?e2. $this time:hasEnd ?e1}
```

5.4. `intervalMeets` *and* `intervalMetBy`

These two properties are also the inverse of one another. Therefore, we focus on `intervalMeets` because every triple on `intervalMetBy` is converted into the inverse triple on `intervalMeets`.

In the Time Ontology, the property `intervalMeets` is defined as follows: “If a proper interval `T1` is `intervalMeets` another proper interval `T2`, then the end of `T1` is coincident with the beginning of `T2`.”. This definition is implemented by the following SHACL-SPARQL rule:

(28)
```
sh:targetSubjectsOf time:intervalMeets;
CONSTRUCT{?e1 time:hasBeginning ?b2. ?b2 time:hasBeginning ?e1.
          ?e1 time:hasEnd ?b2. ?b2 time:hasEnd ?e1}
WHERE{$this time:intervalMeets/time:hasBeginning ?b2. $this time:hasEnd ?e1}
```

In addition, similarly to what has been done in Subsection 5.2 above for the property `intervalStarts`, we must also consider cases in which either the end of the subject of `intervalMeets` or the beginning of its object are not specified. As shown in Figure 18, in cases where the beginning or any instant inside the subject of `intervalMeets` are specified, even if the end is not, it can be still inferred that they occur before the object of `intervalMeets`; similarly, in cases where the end or any instant inside the object of `intervalMeets` are specified, even if its beginning is not, it can be still inferred that the subject of `intervalMeets` occurs before them. These inferences are carried out by the rules in (29).

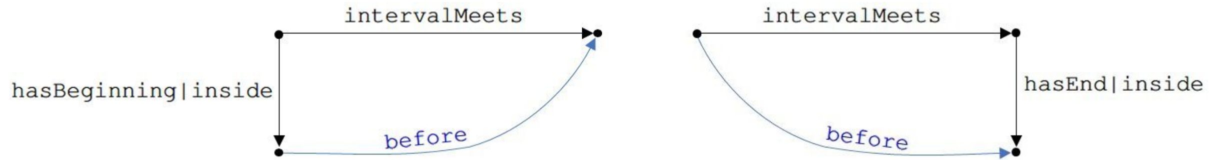

Fig. 18. Inference on the properties `intervalMeets`, `hasBeginning`, `inside`, and `hasEnd`. The inferred properties are shown in blue.

(29)
```
sh:targetSubjectsOf time:intervalMeets;
CONSTRUCT{?i time:before ?te2}
WHERE{$this time:intervalMeets ?te2. $this time:hasBeginning|time:inside ?i}

sh:targetSubjectsOf time:intervalMeets;
CONSTRUCT{$this time:before ?i}
WHERE{$this time:intervalMeets/(time:hasEnd|time:inside) ?i}
```

5.5. `intervalFinishes` *and* `intervalFinishedBy`

These two properties are also the inverse of one another. Therefore, we focus on `intervalFinishes` because every triple on `intervalFinishedBy` is converted into the inverse triple on `intervalFinishes`.

In the Time Ontology, the property `intervalFinishes` is defined as follows: “If a proper interval `T1` is `intervalFinishes` another proper interval `T2`, then the beginning of `T1` is after the beginning of `T2`, and the end of `T1` is coincident with the end of `T2`.”. This definition is of course symmetric to the definition of the property `intervalStarts` seen above. Therefore, we introduce the following SHACL-SPARQL rules, respectively symmetric to the ones shown in (24), (25), and (26) above.

(30)
```
sh:targetSubjectsOf time:intervalFinishes;
CONSTRUCT{?e1 time:hasBeginning ?e2. ?e2 time:hasBeginning ?e1.
          ?e1 time:hasEnd ?e2. ?e2 time:hasEnd ?e1}
WHERE{$this time:intervalFinishes/time:hasEnd ?e2. $this time:hasEnd ?e1}
```

```
sh:targetSubjectsOf time:intervalFinishes;
CONSTRUCT{?b2 time:before $this}
WHERE{$this time:intervalFinishes/time:hasBeginning ?b2}

sh:targetSubjectsOf time:intervalFinishes;
CONSTRUCT{?i1 time:before ?e2}
WHERE{$this time:intervalFinishes/time:hasEnd ?e2.
      $this (time:hasBeginning|time:inside) ?i1}
```

5.6. `intervalDuring` *and* `intervalContains`

These two properties are also the inverse of one another. Therefore, we focus on `intervalDuring` because every triple on `intervalContains` is converted into the inverse triple on `intervalDuring`.

In the Time Ontology, the property `intervalDuring` is defined as follows: "If a proper interval `T1` is `intervalDuring` another proper interval `T2`, then the beginning of `T1` is after the beginning of `T2`, and the end of `T1` is before the end of `T2`.". This definition is implemented by the following SHACL-SPARQL rules, which consider the *whole* interval as subject of `intervalDuring`, similarly to what has been done above for the SHACL-SPARQL rules in (25), (27), and (29).

(31)
```
sh:targetSubjectsOf time:intervalDuring;
CONSTRUCT{?b2 time:before $this}
WHERE{$this time:intervalDuring/time:hasBeginning ?b2}

sh:targetSubjectsOf time:intervalDuring;
CONSTRUCT{$this time:before ?e2}
WHERE{$this time:intervalDuring/time:hasEnd ?e2}
```

5.7. `intervalEquals`

`intervalEquals` states that its subject and its object are the same interval, i.e., that their beginnings, as well as their ends, coincide. `intervalEquals` is a symmetric (and reflexive) property, i.e., it is the inverse property of itself. The property's definition is enforced by the following SHACL-SPARQL rules:

(32)
```
sh:targetSubjectsOf time:intervalEquals;
CONSTRUCT{?b1 time:hasBeginning ?b2. ?b2 time:hasBeginning ?b1.
          ?b1 time:hasEnd ?b2. ?b2 time:hasEnd ?b1}
WHERE{$this time:intervalEquals/time:hasBeginning ?b2.
      $this time:hasBeginning ?b1}

sh:targetSubjectsOf time:intervalEquals;
CONSTRUCT{?e1 time:hasBeginning ?e2. ?e2 time:hasBeginning ?e1.
          ?e1 time:hasEnd ?e2. ?e2 time:hasEnd ?e1}
WHERE{$this time:intervalEquals/time:hasEnd ?e2. $this time:hasEnd ?e1}
```

Furthermore, similarly to what has been done above for the properties `intervalStarts`, `intervalFinishes`, and `intervalMeets`, we must also consider patterns in which `hasBeginning` is specified for only one of the two intervals while for the other one only `hasEnd` is. In such cases, it must be inferred that the object of the specified property `hasBeginning` occurs before the object of the specified property `hasEnd`. The following SHACL-SPARQL rule implements this inference:

(33)
```
sh:targetSubjectsOf time:intervalEquals;
CONSTRUCT{?b time:before ?e}
WHERE{$this time:intervalEquals ?te2.
      {$this time:hasBeginning ?b. ?te2 time:hasEnd ?e}UNION
      {?te2 time:hasBeginning ?b. $this time:hasEnd ?e}}
```

5.8. `intervalIn`

As explained earlier, `intervalIn` (as well as `intervalDisjoint`, which will be discussed in the next subsection) is not one of the basic thirteen properties denoting Allen's temporal relations. It is rather a "derived" property that subsumes `intervalStarts`, `intervalDuring`, and `intervalFinishes`. In other words, `intervalIn` denotes *proper* inclusion, i.e., it states that its subject is included, although it does not coincide, with its object. For this reason, in the Time Ontology `intervalIn` is related with `intervalEquals` through the property `owl:propertyDisjointWith`.

Therefore, if two intervals are related through the property `intervalIn`, it is not possible to infer whether the beginning or the end of one interval either coincide or occur before the beginning or the end of the other interval: it is *unknown* whether the specific property holding between the two intervals is `intervalStarts` rather than `intervalDuring` rather than `intervalFinishes`.

Still, it is *known* that `intervalIn`'s subject is included in its object; therefore, we add to our formalization the two SHACL shapes in (34), which respectively check that the subject of `intervalIn` does not begin before or end after its object; note that the two shapes consider both potential `xsd:dateTime` values associated with the beginnings or ends of the two intervals as well as the property `before`.

(34)
```
sh:targetSubjectsOf time:intervalIn;
SELECT $this ?pi
WHERE{$this time:intervalIn ?pi.
  $this time:hasBeginning ?b1. ?pi time:hasBeginning ?b2.
  {?b1 time:before ?b2}UNION
  {?b1 time:inXSDDateTime ?dt1. ?b2 time:inXSDDateTime ?dt2. FILTER(?dt1<?dt2)}}
sh:message "Invalid triple `{$this} time:intervalIn {?pi}':
            {$this} begins before {?pi}."]].

sh:targetSubjectsOf time:intervalIn;
SELECT $this ?pi
WHERE{$this time:intervalIn ?pi.
  $this time:hasEnd ?e1. ?pi time:hasEnd ?e2.
  {?e2 time:before ?e1}UNION
  {?e1 time:inXSDDateTime ?dt1. ?e2 time:inXSDDateTime ?dt2. FILTER(?dt1>?dt2)}}
sh:message "Invalid triple `{$this} time:intervalIn {?pi}':
            {$this} ends after {?pi}."
```

On the other hand, in some cases it is indeed possible to infer which one of the three specific sub-properties of `intervalIn` holds between the two intervals. In particular: (1) if the beginnings of the two intervals coincide, it is possible to specialize `intervalIn` into `intervalStarts`; (2) if the ends of the two intervals coincide, it is possible to specialize `intervalIn` into `intervalFinishes`; (3) if the beginning of `intervalIn`'s object occurs before the beginning of `intervalIn`'s subject and the end of `intervalIn`'s subject occurs before the end of `intervalIn`'s object, it is possible to specialize `intervalIn` into `intervalDuring`. (1), (2), and (3) are respectively implemented by the three SHACL-SPARQL rules in (35).

(35)
```
sh:targetSubjectsOf time:intervalIn;
CONSTRUCT{$this time:intervalStarts ?te2}
WHERE{$this time:intervalIn ?te2.
  $this time:hasBeginning ?b1. ?te2 time:hasBeginning ?b2.
  {?b1 time:hasBeginning ?b2}UNION
  {?b1 time:inXSDDateTime ?dt1. ?b2 time:inXSDDateTime ?dt2. FILTER(?dt1=?dt2)}}

sh:targetSubjectsOf time:intervalIn;
CONSTRUCT{$this time:intervalFinishes ?te2}
WHERE{$this time:intervalIn ?te2.
  $this time:hasEnd ?e1. ?te2 time:hasEnd ?e2.
  {?e1 time:hasBeginning ?e2}UNION
  {?e1 time:inXSDDateTime ?dt1. ?e2 time:inXSDDateTime ?dt2. FILTER(?dt1=?dt2)}}
```

```
sh:targetSubjectsOf time:intervalIn;
CONSTRUCT{$this time:intervalDuring ?te2}
WHERE{$this time:intervalIn ?te2.
  $this time:hasBeginning ?b1. ?te2 time:hasBeginning ?b2.
  $this time:hasEnd ?e1. ?te2 time:hasEnd ?e2.
  {?b2 time:before ?b1}UNION
  {?b1 time:inXSDDateTime ?dtb1. ?b2 time:inXSDDateTime ?dtb2. FILTER(?dtb2<?dtb1)}
  {?e2 time:before ?e1}UNION
  {?e1 time:inXSDDateTime ?dte1. ?e2 time:inXSDDateTime ?dte2. FILTER(?dte2<?dte1)}
```

5.9. `intervalDisjoint`

When two proper intervals are connected through the property `intervalDisjoint`, no instant can belong to both of them. In other words, one of the two proper intervals must occur before the other one, but, again, it is unknown which one. Therefore, similarly to what has been done in the previous subsection for the property `intervalIn`, both SHACL shapes and SHACL-SPARQL rules are added for the property `intervalDisjoint`.

In particular, we introduce the two SHACL shapes in (36), which check that the two intervals are truly disjoint, i.e., that there is not neither an instant connected to both intervals through two separate chains of `hasBeginning`, `hasEnd`, or `inside` properties, nor two *different* instants connected to the two intervals through two chains as such but associated with the *same* `xsd:dateTime` value.

(36)
```
sh:targetSubjectsOf time:intervalDisjoint;
SELECT $this ?pi ?i
WHERE{$this time:intervalDisjoint ?pi.
      $this (time:hasBeginning|time:hasEnd|time:inside)+ ?i.
      ?pi (time:hasBeginning|time:hasEnd|time:inside)+ ?i}
sh:message "Invalid triple `{$this} time:intervalDisjoint {?pi}':
            the instant {?i} belongs to both {$this} and {?pi}."

sh:targetSubjectsOf time:intervalDisjoint;
SELECT $this ?pi ?i1 ?i2
WHERE{$this time:intervalDisjoint ?pi.
      $this (time:hasBeginning|time:hasEnd|time:inside)+ ?i1.
      ?pi (time:hasBeginning|time:hasEnd|time:inside)+ ?i2.
      ?i1 time:inXSDDateTime ?dt1. ?i2 time:inXSDDateTime ?dt2.
      FILTER(?dt1=?dt2)}
sh:message "Invalid triple `{$this} time:intervalDisjoint {?pi}':
            the instants {?i1} and {?i2}, each of which belongs to one
            of the two intervals, denote the same xsd:dateTime value."
```

Then, we also introduce the SHACL-SPARQL rules in (37) which search for pairs of instants belonging each to one of the two intervals and such that one of the two instants occurs before the other. If two instants as such are found, the same temporal order is inferred also between the two intervals.

(37)
```
sh:targetSubjectsOf time:intervalDisjoint;
CONSTRUCT{$this time:intervalBefore ?te2}
WHERE{$this time:intervalDisjoint ?te2.
  $this (time:hasBeginning|time:hasEnd|time:inside)+ ?i1.
  ?te2 (time:hasBeginning|time:hasEnd|time:inside)+ ?i2.
  {?i1 time:before ?i2}UNION
  {?i1 time:inXSDDateTime ?dt1. ?i2 time:inXSDDateTime ?dt2. FILTER(?dt1<?dt2)}}
```

```
    sh:targetSubjectsOf time:intervalDisjoint;
    CONSTRUCT{?te2 time:intervalBefore $this}
    WHERE{$this time:intervalDisjoint ?te2.
      $this (time:hasBeginning|time:hasEnd|time:inside)+ ?i1.
      ?te2 (time:hasBeginning|time:hasEnd|time:inside)+ ?i2.
      {?i2 time:before ?i1}UNION
      {?i1 time:inXSDDateTime ?dt1. ?i2 time:inXSDDateTime ?dt2. FILTER(?dt1>?dt2)}}
```

5.10. *Evaluation*

In order to assess whether the SHACL-SPARQL rules proposed in the previous subsections, along with the four shapes defined for `intervalIn` and `intervalDisjoint`, adequately cover all knowledge graphs that may be constructed using the properties denoting Allen's temporal relations, we again conducted a case-based evaluation. Specifically, we analyzed each of the 15 properties representing Allen's temporal relations, examining how the associated intervals relate to one another when connected by these properties. This approach offers a clear visual representation of which `hasBeginning`, `hasEnd`, and `before` relations should be inferred. We then verified whether the SHACL-SPARQL rules do correctly perform these inferences.

For example, Figure 19 depicts two intervals connected through the property `intervalOverlaps`. From the figure, it is immediately clear which instants occurring within `intervalOverlaps`'s subject come before which instants occurring within its object. The three SHACL-SPARQL rules shown above in (27) are then respectively asserted to infer each of the three occurrences of `before` in Figure 19. As already explained above in (27), note that the first rule sets the object of the inferred property `before` to the whole object of `intervalOverlaps`: whenever a temporal entity occurs before the instant that begins an interval, then it occurs before the whole interval.

Fig. 19. Inferences on the property `intervalOverlaps`.

Another example is shown in Figure 20, which depicts two proper intervals connected through the property `intervalFinishes`. Again, it is easy to double-check that the SHACL-SPARQL rules in (30) above infer the occurrences of `before` shown in the figure.

Fig. 20. Inferences on the property `intervalFinishes`. The inferred properties are shown in blue.

The other properties denoting Allen's temporal relations can be similarly evaluated. However, this paper omits to provide further details about each of them.

The only two properties that we further discuss are `intervalIn` and `intervalDisjoint`, which also require the introduction of some ad-hoc shapes. As observed earlier, the property `intervalIn` subsumes the properties `intervalStarts`, `intervalFinishes`, and `intervalDuring`. Thus, if two intervals are connected through `intervalIn` it is unknown which one of the three alternatives holds. Similar considerations hold for the property `intervalDisjoint`, which subsumes the properties `intervalBefore` and `intervalAfter`.

Still, it is known that patterns that do not comply with *neither* of the alternatives are invalid. Therefore, it must be checked that `intervalIn`'s subject does not begin before or end after its object, as depicted in Figure 21. The two SHACL shapes in (34) do invalidate patterns as such, both when the involved instants are associated with `xsd:dateTime` values that denote an invalid temporal order and when these instants are related through a `before` property that denotes such an invalid temporal order (as in Figure 21).

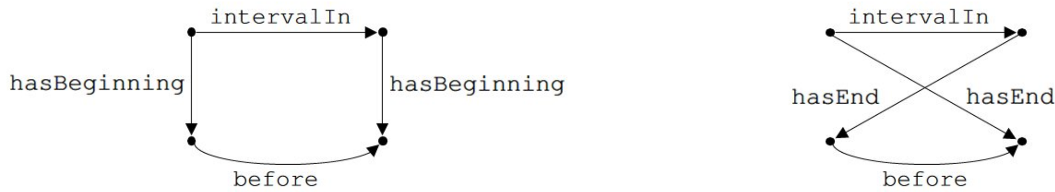

Fig. 21. Invalid patterns involving the property `intervalIn`. The subject of the property cannot begin before or end after its object.

Similarly, it must be checked that two intervals connected through the property `intervalDisjoint` do not share any instant, as depicted in Figure 22. The two SHACL shapes in (36) do invalidate patterns as such, both when there are two different instants associated with the same `xsd:dateTime` value (as in Figure 22) and when there is a single instant connected to both intervals through the properties `hasBeginning`, `hasEnd`, or `inside`.

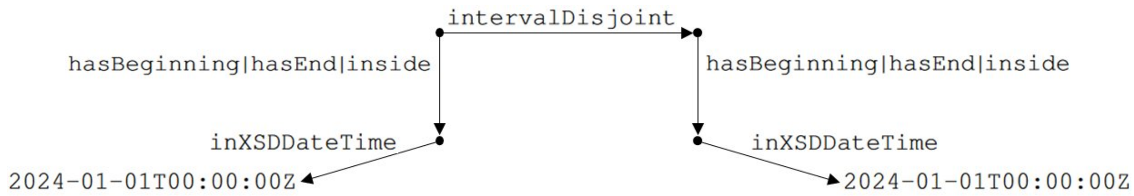

Fig. 22. Invalid patterns involving the property `intervalDisjoint`. The two involved intervals cannot share instants.

Furthermore, for knowledge graphs that comply with the shapes in (34) and (36), it could be possible to infer which one of the alternatives holds on the basis of other triples asserted on the instants within the two intervals.

Note, however, that in some knowledge graphs *two* of these alternatives can be inferred, which is again invalid. Figure 23 shows one of such invalid knowledge graphs. Since the beginning of `intervalIn`'s object begins at the beginning of its subject, the SHACL-SPARQL rules in (35) infer that the property `intervalStarts` holds between the two intervals. However, since it also holds that the end of `intervalIn`'s subject begins at the end of its object, the same rules also infer that the property `intervalFinishes` holds between the same pair of intervals. From these, the rules in (25) and (30) infer `before` properties, between the instants occurring in the two intervals, that are invalid altogether, as detected by the SHACL shapes introduced in Subsection 4.2 above.

Similar considerations hold for the property `intervalDisjoint`, as exemplified in Figure 24. In the knowledge graph shown in the figure, `intervalDisjoint`'s object begins before the beginning of its subject; therefore, the SHACL-SPARQL rules in (37) infer that the property `intervalBefore` holds between `intervalDisjoint`'s subject and object. However, in the knowledge graph it also holds that `intervalDisjoint`'s subject ends before the end of `intervalDisjoint`'s object; therefore, the rules in (37) also infer that the property `intervalBefore` holds between `intervalDisjoint`'s object and subject. From the two inferred properties, it is in turn inferred a cycle of `before` properties, which the SHACL shape in (14) above detects as invalid.

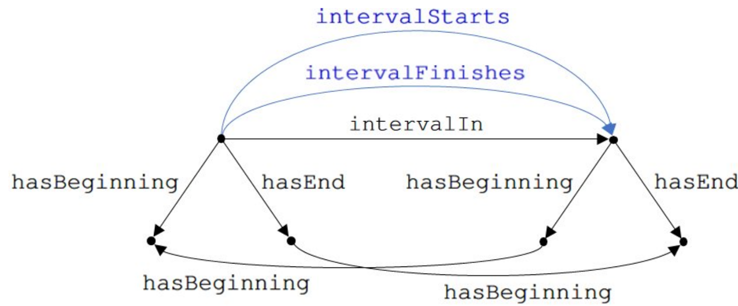

Fig. 23. Invalid patterns involving the property `intervalIn`. The two object properties in blue are inferred.

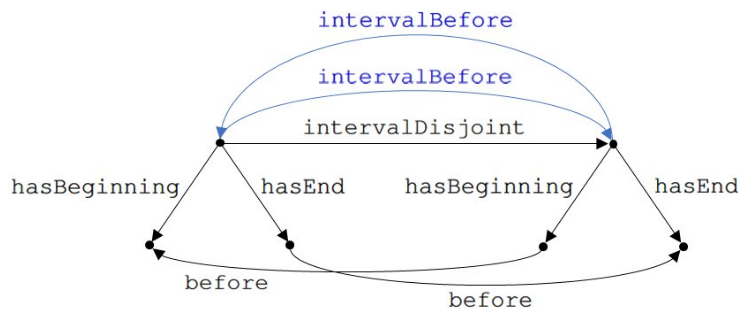

Fig. 24. Invalid patterns involving the property `intervalDisjoint`. The two object properties in blue are inferred.

6. Adding SHACL Shapes and SHACL-SPARQL rules to the Time Ontology: disjointness, symmetricity, and reflexivity of the properties denoting Allen's temporal relations

As explained at the beginning of Section 3 above, the Time Ontology asserts the properties `intervalIn` and `intervalEquals` as disjoint, by connecting them through the property `owl:propertyDisjointWith`.

Nevertheless, many more similar constraints can be defined among the properties denoting Allen's temporal relations. In fact, it is easy to see that:

(38) a. All thirteen properties denoting one of Allen's *basic* temporal relations are disjoint of one another. In addition, `intervalIn` is disjoint with all other properties but its sub-properties `intervalStarts`, `intervalFinishes`, and `intervalDuring` while `intervalDisjoint` is disjoint with all other properties but its sub-properties `intervalBefore` and `intervalAfter`.

b. All fifteen properties but `intervalDisjoint` and `intervalEquals` are asymmetric.

c. All fifteen properties but `intervalEquals` are irreflexive.

In light of this, we think that the current version of the Time Ontology should be extended by encoding *all* constraints in (38.a-c) through OWL axioms that involve the properties `owl:propertyDisjointWith`, `owl:AsymmetricProperty`, and `owl:IrreflexiveProperty`.

In our formalization, which relies on SHACL rather than OWL, we introduce specialized SHACL shapes and SHACL-SPARQL rules to encode (38.a-c). Specifically, this paper presents an alternative approach that not only encodes (38.a-b) but also validates *paths* of properties representing Allen's temporal relations, constructed using the composition table in Table 1. This section will demonstrate how our approach achieves (38.a-b), while the next section will illustrate how it validates *paths* of properties denoting Allen's temporal relations built through Table 1.

The proposed solution employs RDF reification[22] to determine the set of Allen's temporal relations that can *possibly* connect two proper intervals. If an explicitly asserted property is *not* among these possible properties,

[22] https://www.w3.org/TR/rdf-primer/#reification

the knowledge graph is deemed invalid. As explained in subsection 6.1 below, this approach aligns with standard ontology design patterns commonly used in the literature for similar representational problems.

In order to represent the set of *possible* Allen's temporal relations that may connect a pair of proper intervals, we introduce two new special properties: `hasPossibleATR` and `oneOf`. The former connects pairs of proper intervals that must be validated. The latter ascribes possible values to the former; in order to do so, `hasPossibleATR` is reified and used as subject of `oneOf`, which will connect it to these possible values.

Figure 25 shows an example. Suppose that two proper intervals `pi1` and `pi2` are connected through the property `intervalIn`. We therefore need to represent which other properties are compatible with this connection, in both directions. In order to do so, we create two reifications of the property `hasPossibleATR`: one connecting `pi1` with `pi2` and the other one connecting `pi2` with `pi1`. Possible values of the former are `intervalIn` itself but also its sub-properties `intervalStarts`, `intervalDuring`, and `intervalFinishes`. Possible values of the latter are `intervalStartedBy`, `intervalContains`, and `intervalFinishedBy`, i.e., the inverse properties of `intervalStarts`, `intervalDuring`, and `intervalFinishes`.

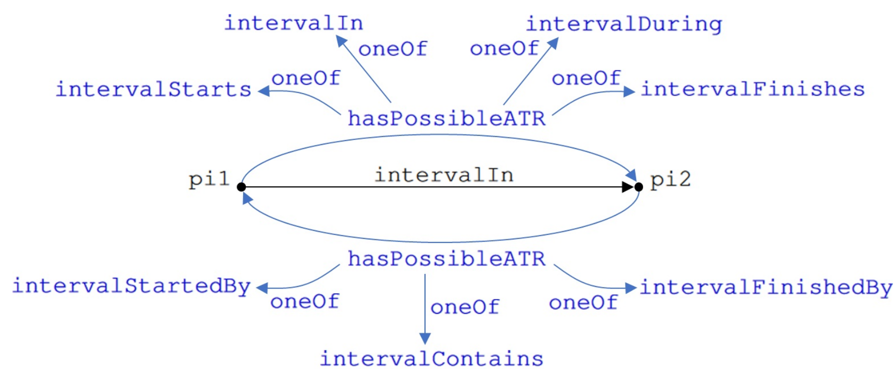

Fig. 25. Inferring *possible* properties among proper intervals. The properties in blue are those that must be inferred from the one in black.

In order to infer the properties `hasPossibleATR` and `oneOf` as explained in Figure 25, the SHACL-SPARQL rules in (39) and (40) are introduced.

(39)
```
sh:targetClass time:ProperInterval;
CONSTRUCT{[rdf:type rdf:Statement,?cp; rdf:subject $this; rdf:object ?pi2;
            rdf:predicate :hasPossibleATR; :includes $this,?pi2; :source ?atr].
           [rdf:type rdf:Statement,?cp; rdf:subject ?pi2; rdf:object $this;
            rdf:predicate :hasPossibleATR; :includes $this,?pi2; :sourcei ?atr]}
WHERE{$this ?atr ?pi2.
       FILTER(strStarts(str(?atr),"http://www.w3.org/2006/time#interval"))
       BIND(IF($this=?pi2, :ContainsCyclingPath, rdf:Statement) AS ?cp)
       NOT EXISTS{?r rdf:type rdf:Statement; rdf:subject $this;
                   rdf:object ?pi2; rdf:predicate :hasPossibleATR; :source ?atr}}
```

(40)
```
sh:targetClass rdf:Statement
CONSTRUCT{$this :oneOf time:intervalIn,time:intervalStarts,
           time:intervalFinishes,time:intervalDuring. ?thisi :oneOf
           time:intervalContains,time:intervalStartedBy,time:intervalFinishedBy}
WHERE{$this rdf:subject ?pi1; rdf:object ?pi2; rdf:predicate :hasPossibleATR;
       :source time:intervalIn. ?thisi rdf:subject ?pi2; rdf:object ?pi1;
       rdf:predicate :hasPossibleATR; :sourcei time:intervalIn}
```

The SHACL-SPARQL rule in (39) creates the reifications of the property `hasPossibleATR` between two proper intervals connected by a property denoting one of Allen's temporal relations, in both directions. Note that the two

reifications are created only in cases where they do not already exist, as enforced by the `NOT EXISTS` clause in (39); since the rules are re-executed until no new triple is inferred, this clause prevents infinite loops.

Note also that the rule in (39) introduces other three properties (`source`, `sourcei`, and `includes`) and another class (`ContainsCyclingPath`).

`source` and `sourcei` associate the two reifications with the property denoting the Allen's temporal relations from which they are created, e.g., `intervalIn` in Figure 25. The difference between `source` and `sourcei` is that the former marks the reification in the same direction of the Allen's temporal relation while the latter marks the reification in opposite direction ("i" stands for "inverse"). `source` and `sourcei` are used in the rules that associate the reifications with the possible properties through the `oneOf` property, e.g., rule (40).

`includes` associates each reification with the proper intervals belonging to the path denoted by the reification itself (e.g., only the two initial proper intervals for reifications created from properties denoting Allen's temporal relations through the rule in (39)) while `ContainsCyclingPath` is a special class that flags all paths that contain a cycle of `hasPossibleATR` properties between (some of) the proper intervals included therein. With respect to the rule in (39), it is possible to create such a cycle only if the property denoting one of Allen's temporal relation connects a proper interval with itself, as enforced by the `BIND` clause in (39).

`includes` and `ContainsCyclingPath` are not relevant for the content of this section, but they will be crucial for the content of the the next one, i.e., for validating paths of properties denoting Allen's temporal relations.

Once the initial reifications are created through the rule in (39), further rules are needed to populate the set of possible properties that can hold between the two involved intervals. We defined one of such rules for each of the fifteen Allen's temporal relations. (40) shows the one associated with `intervalIn`; this rule associates, through the property `oneOf`, the reification in the same direction with the property itself but also with its sub-properties `intervalStarts`, `intervalDuring`, and `intervalFinishes`. Conversely, the reification in the opposite direction is associated, through the property `oneOf`, with `intervalStartedBy`, `intervalContains`, and `intervalFinishedBy`. Thus, the rules in (39) and (40) infer the properties shown in blue in Figure 25.

Now that the sets of *possible* properties denoting Allen's temporal relations have been inferred, out of one of these properties that it is *actually* asserted between two proper intervals, we must check that the two proper intervals are not connected by any *other* property that denote one of Allen's temporal relations and that is not listed among the set of *possible* properties that may hold between them. This is done by the SHACL shape in (41), which invalidates any property that denotes one of Allen's temporal relation and that does not comply with this constraint.

(41)
```
sh:targetClass rdf:Statement;
SELECT $this ?pi1 ?atr ?pi2
WHERE{$this rdf:subject ?pi1; rdf:object ?pi2. ?pi1 ?atr ?pi2.
      FILTER(strStarts(str(?atr), "http://www.w3.org/2006/time#interval"))
      NOT EXISTS{$this :oneOf ?atr}}
sh:message "Invalid triple '{?pi1} ?atr ?pi2': {?pi1} and {?pi2} cannot
            be connected through {?atr}, given the other Allen's temporal
            relations connecting them."
```

Let us now show two examples of knowledge graphs invalidated by the SHACL shape in (41): the two ones shown in Figure 26. The knowledge graph on the left is invalid because `pi1` is connected to `pi2` both with the property `intervalDuring` and the property `intervalMeets`; however, these two properties are disjoint, as explained in (38.a) above: they cannot hold between the same pair of proper intervals. The knowledge graph on the right is invalid because the property `intervalBefore` connects `pi1` and `pi2` in both directions; however, the property is asymmetric, as explained in (38.b) above.

The SHACL-SPARQL rule in (39) creates two reifications for the property `intervalDuring` occurring in the knowledge graph on the left and for the property `intervalBefore` connecting `pi1` to `pi2` in the knowledge graph on the right. Indeed, the rule creates two more reifications for the property `intervalMeets` occurring in the knowledge graph on the left and two more for the property `intervalBefore` connecting `pi2` to `pi1` in the knowledge graph on the right; however, these are not shown in Figure 26, in order to enhance readability.

The two reifications are then populated with the set of all possible properties that denote one of Allen's temporal relations and that might be asserted between `pi1` and `pi2`. This is done by the SHACL-SPARQL rules in (42) and

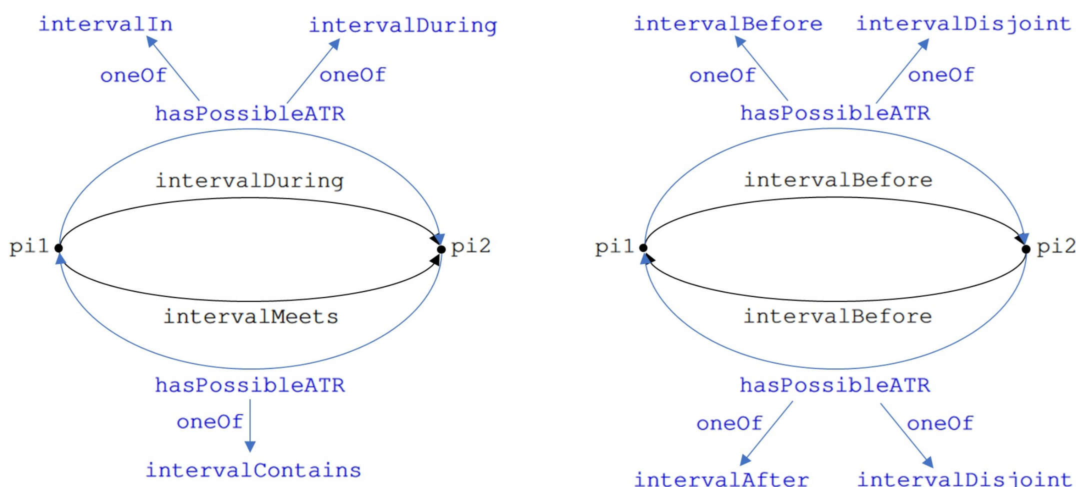

Fig. 26. Inferring *possible* properties among proper intervals. The properties in blue inferred.

(43), which respectively concern the properties `intervalDuring` and `intervalBefore` and parallel the rule in (40), which concerns the property `intervalIn`. As said above, the GitHub includes one of such rules for each of the fifteen Allen's temporal relations, although the paper only shows the three ones in (40), (42), and (43).

(42)
```
sh:targetClass rdf:Statement
CONSTRUCT{$this :oneOf time:intervalDuring,time:intervalIn.
          ?thisi :oneOf time:intervalContains}
WHERE{$this rdf:subject ?pi1; rdf:object ?pi2; rdf:predicate :hasPossibleATR;
      :source time:intervalDuring. ?thisi rdf:subject ?pi2; rdf:object ?pi1;
      rdf:predicate :hasPossibleATR; :sourcei time:intervalDuring}
```

(43)
```
sh:targetClass rdf:Statement
CONSTRUCT{$this :oneOf time:intervalBefore,time:intervalDisjoint.
          ?thisi :oneOf time:intervalAfter, time:intervalDisjoint}
WHERE{$this rdf:subject ?pi1; rdf:object ?pi2; rdf:predicate :hasPossibleATR;
      :source time:intervalBefore. ?thisi rdf:subject ?pi2; rdf:object ?pi1;
      rdf:predicate :hasPossibleATR; :sourcei time:intervalBefore}
```

The two rules in (42) and (43) add the `oneOf` properties shown in blue in Figure 26 and, finally, the SHACL shape in (41) reports the two inferred knowledge graphs as invalid because they each connects two proper intervals through a property denoting one of Allen's temporal relations but this property is not listed among the possible properties of the reification in the same direction.

We conclude by introducing the SHACL shape in (44), which invalidates every knowledge graph in which a proper interval is connected to itself through a property denoting one of Allen's temporal relations different from `intervalEquals`. As explained in (38.c) above, all properties denoting Allen's temporal relations but `intervalEquals` are irreflexive. The SHACL-SPARQL rule in (42) creates the reification of the property `hasPossibleATR` that connects the proper interval with itself; furthermore, the rule associates this reification with the property denoting one of Allen's temporal relations from which the reification was created. After that, the SHACL shape in (44) checks that this property was `intervalEquals`.

It is easy to verify that the SHACL shape in (44) properly validates all fifteen properties when they connect a proper interval with itself. The next section will show that this shape also validates *any* arbitrary path of properties denoting Allen's temporal relations that connects a proper interval with itself.

(44)
```
sh:targetClass rdf:Statement;
SELECT $this ?pi
WHERE{$this rdf:subject ?pi; rdf:predicate :hasPossibleATR; rdf:object ?pi.
      NOT EXISTS{$this :oneOf time:intervalEquals}}
sh:message "Invalid ProperInterval {?pi}: it is connected to itself
            through a cycling path of Allen's temporal relations but these
            do not allow for time:intervalEquals between {?pi} and itself."
```

6.1. Evaluation

The SHACL shapes and SHACL-SPARQL rules presented in this section, and even more so those that will be introduced in the next one, mark a fundamental distinction from the shapes and rules discussed in previous sections. Moreover, they provide further evidence that not only SPARQL property path operators alone are insufficient for properly validating the Time Ontology, as already demonstrated in Figure 5 above, but also that other rule-based languages commonly used for inference in the Semantic Web, e.g., SWRL, are inadequate.

To validate properties denoting Allen's temporal relations, we need to represent that, between two specific proper intervals, only a particular *set* of Allen's temporal relations, computed using Allen's compositional table, can hold. This cannot be represented in RDF: the basic representational unit in RDF is the triple, which only allows connecting *individual* RDF resources, whereas here the second element of the triple must be a set, not a single individual.

This representational issue is not new in the Semantic Web. In fact, it is the same issue that led to the definition of RDF*[23]. RDF* extends RDF by allowing triples to be used as subjects or objects in other triples, thereby enabling a more direct representation of metadata related to properties holding between two RDF resources.

However, RDF* does not increase the expressive power of RDF; it simply provides a more compact syntax for implementing RDF reification. Therefore, we chose not to use RDF* in our formalization but to adhere to the RDF standard, especially since only two rules in our framework create anonymous individuals that are reifications of properties: the rule shown in (39) above and the rule shown in (46) in the next section.

Still, while we could have used RDF* to formalize our framework, it is clear that we could not have used SWRL, OWL, or any other rule-based language (not necessarily for the Semantic Web) that does not allow the creation of new anonymous individuals. To the best of our knowledge, only SPARQL supports this capability, therefore it seems the most suitable choice for formalizing the rules needed to validate the sub-algebra of Allen's temporal algebra enabled by the Time Ontology vocabulary.

Concerning the evaluation of the specific SHACL shapes and SHACL-SPARQL rules introduced in this section, since they involve only two intervals, we chose to carry out a simpler generate & test evaluation: we implemented a small script in Java that (1) generates all possible configurations involving (two) properties holding between the same pair of proper intervals, (2) executes (first) the rules and (then) the shape introduced above in this section, and (3) prints the results of the validation into an output file. The script is available in the GitHub repository associated with this paper together with instructions to re-execute it, for the reader to double-check the results.

When the two properties connect the two proper intervals in the same direction, there are 15!/((15-13)!2!)=105 possible configurations to consider. On the other hand, when they connect them in opposite directions there are 105+15=120 configurations to consider, because we must add the 15 configurations in which the property is the same, e.g., the one shown in Figure 26 on the right. Therefore, there are 105+120=225 configurations in total.

We manually checked all configurations one by one and, in line with the properties' definitions, we verified that only 18 of them are valid. Specifically, when the two properties connect the pair of proper intervals in the same direction, only 5 configurations are valid, i.e., when it holds that:

[23]https://www.w3.org/2022/08/rdf-star-wg-charter

a. One of the two properties is either `intervalBefore` or `intervalAfter` while the other one is `intervalDisjoint`.

b. One of the two properties is either `intervalDuring`, `intervalStarts` or `intervalFinishes` while the other one is `intervalIn`.

Conversely, when the two properties connect the pair of proper intervals in opposite directions, only 13 configurations are valid, i.e., when it holds that:

a. One of the two properties is the inverse of the other one.

b. Both properties are either `intervalEquals` or `intervalDisjoint`.

c. One of the two properties is `intervalDisjoint` while the other one is either `intervalBefore` or `intervalAfter`.

d. One of the two properties is `intervalIn` while the other one is either `intervalContains`, `intervalStartedBy`, or `intervalFinishedBy`.

All other configurations are invalidated by the SHACL-SPARQL rules and the SHACL shape introduced above: the latter identifies that one of the two properties does not belong to the set of properties connected through `oneOf` to the reification of same direction's `hasPossibleATR` property.

As already mentioned earlier, the script that we used to generate & test all 225 configurations is available on the GitHub repository. We invite the reader to locally re-execute it and double-check the results.

7. Adding SHACL-SPARQL rules to the Time Ontology: implementing Allen's composition table

This section implements the composition table shown in Table 1, which is necessary for validating multiple *paths* of properties denoting Allen's temporal relations between the same pair of proper intervals. Figure 27 illustrates an example of an invalid knowledge graph that can only be identified as such through the composition table. By applying the composition table reproduced in Table 1, it can be deduced that `pi2` cannot be connected to `pi1` *both* through the chain of properties `intervalOverlaps`, `intervalFinishes`, and `intervalMetBy`, *and* the property `intervalContains`: the three properties in the first path allow only certain Allen's temporal relations between `pi1` and `pi2`, but none of them is `intervalDuring`, i.e., the inverse of `intervalContains`.

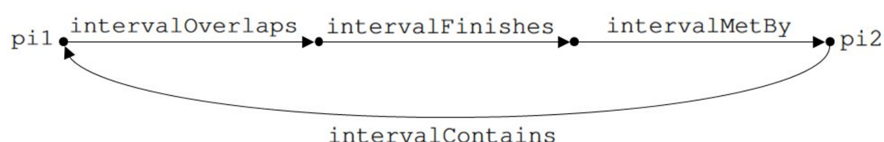

Fig. 27. Invalid knowledge graph involving properties denoting Allen's temporal relations.

More generally, whenever two proper intervals `pi1` and `pi2` are connected through a path of properties denoting Allen's temporal relations, the composition table determines two sets of possible properties between `pi1` and `pi2`, one for each direction. As explained in the previous section, in our solution each set is represented by the reification of the property `hasPossibleATR`, linked to its members via the property `oneOf`. Consequently, as illustrated in Figure 28, when *two* paths of properties denoting Allen's temporal relations exist between `pi1` and `pi2` we calculate *four* sets of possible properties, two for each direction.
It is clear that the knowledge graph is invalid if and only if the two possible sets in one direction share *no* elements. This indicates that no property representing one of Allen's temporal relations can simultaneously satisfy both paths already asserted in the graph. In other words, none of the properties derived from the composition table along one path appear in the set of properties derived from the composition table along the other path.

The SHACL shape in (45) enforces this validation check: in case there are *two* reifications of `hasPossibleATR` between the *same* pair of proper intervals but there is *not* at least one property connected to both reification through

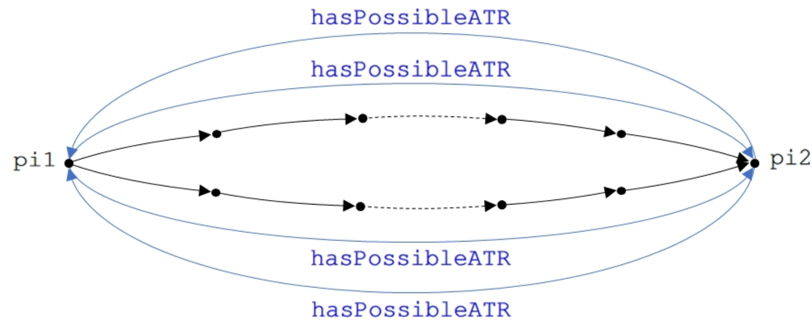

Fig. 28. Pattern of knowledge graph that must be validated. pi1 and pi2 represent proper intervals while the black lines represent paths of properties denoting Allen's temporal relations. The knowledge graph is invalid if the two hasPossibleATR properties in one direction, each of which reifies a set of possible properties between the two intervals in that direction, do not have at least one property in common.

oneOf, the knowledge graph is reported as invalid. Note that this shape generalizes the one in (41): the latter covers the specific case in which one of the two paths consists of a single property.

(45)
```
sh:targetClass rdf:Statement;
SELECT $this ?pi1 ?pi2
WHERE{$this rdf:subject ?pi1; rdf:predicate :hasPossibleATR; rdf:object ?pi2.
      ?hpatr rdf:subject ?pi1; rdf:predicate :hasPossibleATR; rdf:object ?pi2.
      NOT EXISTS{$this :oneOf ?atr. ?hpatr :oneOf ?atr}}
sh:message "Invalid knowledge graph: no Allen's temporal relation can exist
            between {?pi1} and {?pi2}, given the Allen's temporal relations
            already asserted in the graph."
```

The remainder of this section presents SHACL-SPARQL rules that implement the composition table in Table 1. It also shows that when these rules are applied to the sequence of three properties connecting pi1 to pi2 in Figure 27, they infer that intervalContains does *not* belong to the set of possible properties that could connect pi2 to pi1. Consequently, the SHACL shape in (45) invalidates the knowledge graph.

(46) presents the first SHACL-SPARQL rule used to implement Allen's composition table. This rule generates a reification representing a path of properties denoting Allen's temporal relations by combining the reifications of two *contiguous* paths. The newly created reification is then linked to the original ones via the properties source1 and source2. To prevent infinite loops, the rule is triggered only if the reification it creates does not already exist, i.e., only if the knowledge graph does not already contain a reification with the same sources.

Furthermore, the rule does not use as sources reifications that contain cyclic paths. This is because the SHACL shape in (44) already invalidates proper intervals connected to themselves through a sequence of Allen's temporal properties that do not permit the property intervalEquals. Conversely, as it will be shown in the next subsection, in cases where two intervals are equal and validated by the SHACL shape in (44), there is no need to check paths that include cycles among these intervals: only the segments of those paths that exclude the cycles need to be considered. Finally, the path denoted by the reification created by this rule may itself contain a cycle. This can be detected by verifying that the two reifications referenced by source1 and source2 do not share a common proper interval other than ?pi2, i.e., the interval that connects them. If such a cycle is detected, the reification created in the CONSTRUCT clause will again be asserted as an instance of the special class ContainsCyclingPath.

(46)
```
sh:targetClass rdf:Statement; sh:order 1;
CONSTRUCT{[rdf:type rdf:Statement,?cp; rdf:subject ?pi1; rdf:object ?pi3;
          rdf:predicate :hasPossibleATR; :source1 $this; :source2 ?r2]}
WHERE{$this rdf:subject ?pi1; rdf:predicate :hasPossibleATR; rdf:object ?pi2.
      ?r2 rdf:subject ?pi2; rdf:predicate :hasPossibleATR; rdf:object ?pi3.
      NOT EXISTS{?r rdf:type rdf:Statement; :source1 $this; :source2 ?r2}
      NOT EXISTS{$this rdf:type :ContainsCyclingPath}
      NOT EXISTS{?r2 rdf:type :ContainsCyclingPath}
      BIND(IF(EXISTS{$this :includes ?p. ?r2 :includes ?p. FILTER(?p!=?pi2)},
          :ContainsCyclingPath, rdf:Statement) AS ?cp)}
```

Note that the `BIND` clause in (46) checks the property `includes`, namely the proper intervals that are included within the two source reifications. As explained in the previous section, the SHACL-SPARQL rule in (39) connects the initial reifications to the two involved proper intervals through the property `includes`. Another SHACL-SPARQL rule is therefore needed to include in the reification created by (46) any proper interval included in either of its sources, i.e., to build the *union* of the sets of proper intervals connected through `includes` to either `source1` or `source2`. Furthermore, it must be guaranteed that this rule is always executed *before* the one in (46); conversely, if the latter is executed before the proper intervals included within the source reifications have been computed, the rule in (46) will not detect cycling paths.

The SHACL-SPARQL rule that creates the union set of the proper intervals included in the two source reifications and includes them within the reification denoting the conjoined path is shown in (47). Note that the latter specifies "`sh:order 0`" while the rule in (46) specifies "`sh:order 1`"; `sh:order` is a special SHACL property to specify the execution order of the rules.[24] The specified values for the property `sh:order` guarantee that the rule in (47) is always executed before the one in (46).

(47)
```
sh:targetClass rdf:Statement; sh:order 0;
CONSTRUCT{$this :includes ?pi}
WHERE{{$this :source1 ?s}UNION{$this :source2 ?s}. ?s :includes ?pi}
```

Rules (47) and (46) demonstrate that SPARQL alone is insufficient to fully validate the portion of the Time Ontology shown in Figure 1, as it lacks support for rule prioritization. Many other use cases also require rule prioritization, e.g., to implement defeasibility, which necessitates the use of SHACL-SPARQL rules, as exemplified in [49] and [20]. Conversely, when prioritization is not required, the reasoning process can be simplified by repeatedly executing SPARQL queries in the form of `CONSTRUCT-WHERE` until no further triples are inferred; an example of this approach is provided in [50].

From the knowledge graph in Figure 27, the SHACL-SPARQL rules in (46) and (47), together with the rules shown in the previous section, infer the reifications and the properties shown in blue in Figure 29.
In order to enhance readability, Figure 29 only shows the reifications built in opposite directions, which are the ones with which the property `intervalContains` will be invalidated. We also omit the occurrences of the property `includes` from the figure, as they are only needed to detect cycles fit to avoid the rule in (46) to loop infinitely.

The three initial reifications in opposite directions are created by the rule in (39) out of the three asserted properties `intervalOverlaps`, `intervalFinishes`, and `intervalMetBy`; the latter are connected to these three reifications through the property `sourcei`. Of course, the rule in (39) also creates reifications from the property `intervalContains`; however, since no other property directly connects two proper intervals that are also connected through a path of properties including `intervalContains`, the reifications created from the latter have no effect and so they are also omitted from the figure. After reifications have been created, the rules that parallel (40), (42), and (43) for the properties `intervalOverlaps`, `intervalFinishes`, and `intervalMetBy` associate the *inverse* properties (`intervalOverlappedBy`, `intervalFinishedBy`, and `intervalMeets`) with the three initial reifications under discussion, through the property `oneOf`. In parallel, the rule in (46) first creates the reification from `pi2` to the subject of `intervalFinishes` by conjoining the two initial reifications

[24]https://www.w3.org/TR/shacl-af/#rules-order

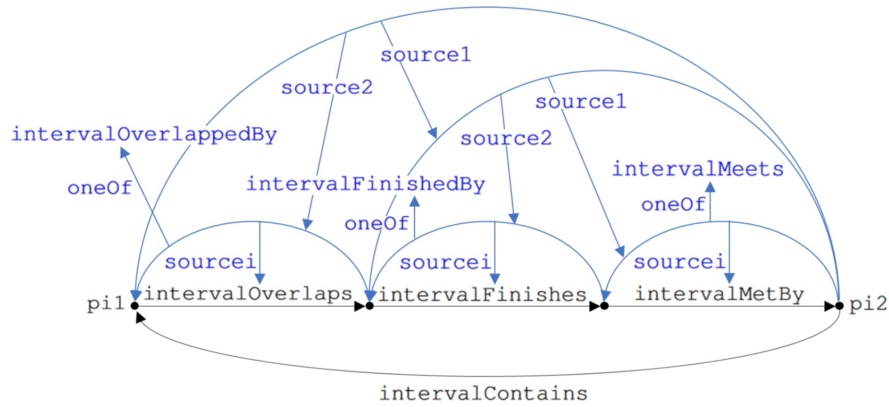

Fig. 29. Invalid knowledge graph involving properties denoting Allen's temporal relations.

on the right; then, it creates the reification from `pi2` to `pi1` by conjoining the latter with the third initial reification. The rule in (46) also connects the reifications referring to the two conjoined paths with the reifications of their two respective sub-paths that are conjoined through the properties `source1` and `source2`.

Next, we need SHACL-SPARQL rules that use the property `oneOf` to populate the reifications of the conjoined paths with the set of possible properties representing Allen's temporal relations. This is determined based on the properties connected via `oneOf` to the two reifications in `source1` and `source2`. These rules serve to implement the composition table in Table 1. This paper presents only the composition rules relevant to the example in Figure 29. However, the complete set of rules implementing Table 1 is available in the GitHub repository.

The first rule that we need for the example in Figure 29 is the one that composes `intervalMeets` with `intervalFinishedBy`. This rule, shown in (48), corresponds to the cell at the row "`Meets (m)`" and the column "`fi`" in Table 1; that cell specifies that by composing `intervalMeets` with `intervalFinishedBy` the only possible Allen's temporal relation is `intervalBefore`. Note that (48) also asserts `intervalDisjoint`, super-property of `intervalBefore`, among the possible properties on the conjoined path.

(48)
```
sh:targetSubjectsOf :source1;
CONSTRUCT{$this :oneOf time:intervalBefore,time:intervalDisjoint}
WHERE{$this :source1 ?r1; :source2 ?r2. ?r1 :oneOf time:intervalMeets.
      ?r2 :oneOf time:intervalFinishedBy}
```

The second rule that we need for the example in Figure 29, shown in (49), combines `intervalBefore` with `intervalOverlappedBy`; as specified in the corresponding cell of Table 1, the possible properties on the conjoined path are "`< o m d s`", i.e., `intervalBefore`, `intervalOverlaps`, `intervalMeets`, `intervalDuring`, and `intervalStarts`, to which the rule in (49) adds `intervalDisjoint` and `intervalIn`, super-properties of `intervalBefore`, `intervalDuring`, and `intervalStarts`.

(49)
```
sh:targetSubjectsOf :source1;
CONSTRUCT{$this :oneOf time:intervalBefore,time:intervalDisjoint,
          time:intervalOverlaps,time:intervalMeets,time:intervalDuring,
          time:intervalStarts,time:intervalIn}
WHERE{$this :source1 ?r1; :source2 ?r2. ?r1 :oneOf time:intervalBefore.
      ?r2 :oneOf time:intervalOverlappedBy}
```

By applying the rules in (48) and (49) to the knowledge graph in Figure 29, the one in Figure 30 is obtained. Figure 30 omits `sourcei`, `source1`, and `source2` in order to enhance readability. From the figure, it is easy to see that `intervalContains` does not belong to the set of possible properties leading from `pi2` to `pi1`, given the asserted properties `intervalOverlaps`, `intervalFinishes`, and `intervalMetBy`. For that reason, the knowledge graph is invalidated by the SHACL shape in (45).

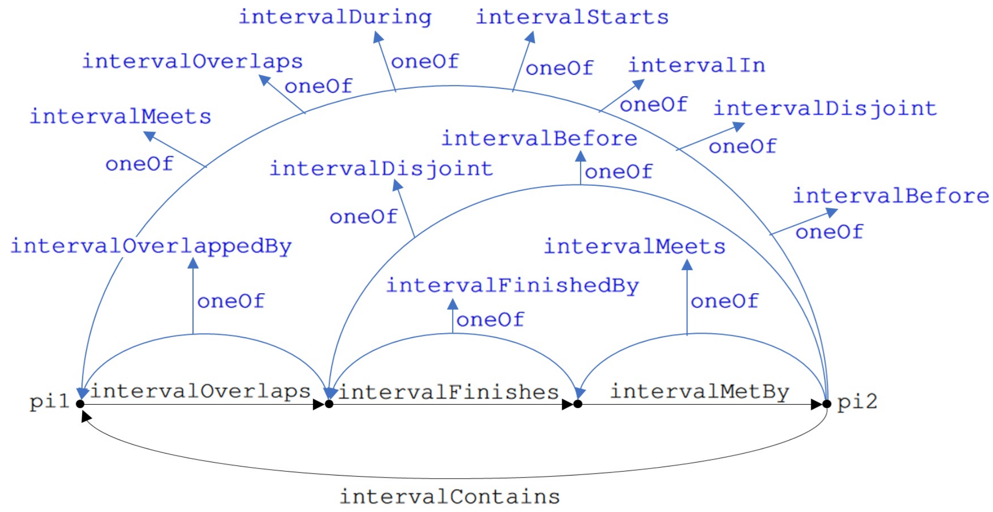

Fig. 30. Properties inferred by the SHACL-SPARQL rules in (48) and (49) from the knowledge graph in Figure 29.

As a second example, consider the knowledge graph in Figure 31, which describes a valid cycling path of properties denoting Allen's temporal relations. Clearly, the only possible relations between a proper interval and itself is intervalEquals, as checked by the SHACL shape in (44). This is indeed the case, as shown in the figure with respect to the proper interval pi1: according to Table 1, by composing intervalOverlappedBy with intervalStartedBy, it is inferred that the only possible properties that may hold from pi1 to pi2 are intervalAfter, intervalDisjoint, intervalOverlappedBy, and intervalMetBy. From pi2 to pi1, intervalDisjoint is asserted, therefore the possible properties that may hold between the two proper intervals in that directions are intervalAfter and intervalBefore, besides intervalDisjoint itself. Finally, by composing intervalAfter with intervalBefore, any property is possible, including intervalEquals, as shown in the figure. For this reason, the knowledge graph is *not* detected as invalid. The reader can verify that intervalEquals is also linked through oneOf to the cycling paths that connect each of the other two intervals to themselves.

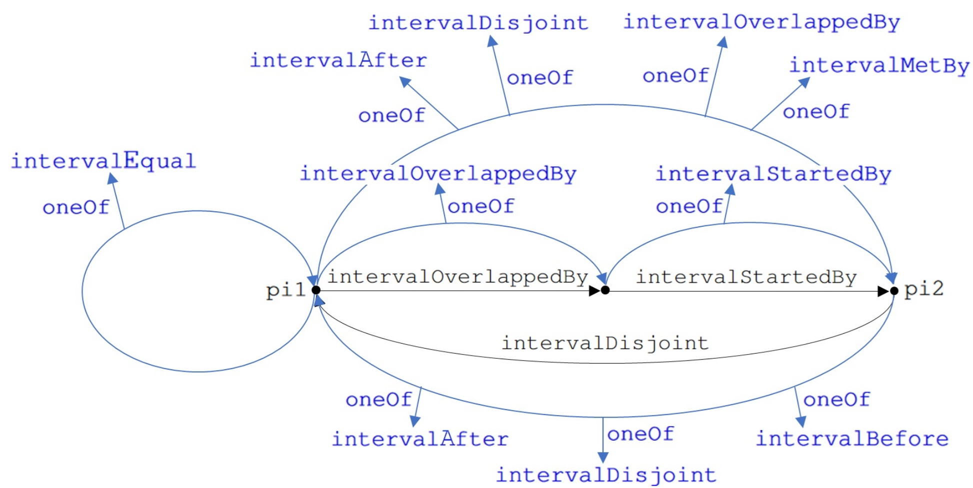

Fig. 31. Valid knowledge graph featuring a cycling path of properties denoting Allen's temporal relations. Inferred triples are shown in blue.

7.1. Evaluation

This section introduced SHACL shapes and SHACL-SPARQL rules for validating paths of properties denoting Allen's temporal relations. The composition table in Table 1 is used to compute two sets of possible properties, denoting Allen's temporal relations, that can be asserted between the two ends of a path, one for each direction. Then, the SHACL shape in (45) ensures that, in cases where two such sets are derived (each from a *different* path but connecting the *same* pair of intervals in the *same* direction), at least one element is shared between them.

The composition table necessitates 144 SHACL-SPARQL rules, one for each cell in the table, along with an additional 26 rules for `intervalEquals`, which were omitted from Table 1 as they are trivial.

To evaluate that these rules are sound and complete with respect to Table 1, we performed a simple generate-and-test evaluation, similar to the approach used in the previous section. We implemented a small script in Java to generate the composition table and manually verified each rule's inferences, ensuring that each rule outputs the same results as those reported in the corresponding table cell. This script is also available in the GitHub repository associated with this paper, along with instructions for re-executing it. The reader is invited to run the script locally and double-check the inferences.

Nevertheless, we remind the reader that the solution presented in this section does not implement the full version of Allen's temporal algebra, but only the sub-algebra that can be represented using the current vocabulary of the Time Ontology. In particular, the full version of Allen's temporal algebra allows for the specification of vectors of *multiple* basic temporal relations between two proper intervals, while the vocabulary of the Time Ontology only allows for the specification of *single* basic temporal properties between two proper intervals, i.e., unary vectors.

As demonstrated in [17] and [48], validating vectors of temporal relations requires considering pairs, triples, etc., of vectors simultaneously, leading to algorithms with NP-hard complexity. In addition, as discussed in the next section, devoted to conclusions and future work, the expressivity of SHACL-SPARQL is likely insufficient to implement these checks, for which more expressive formalisms would be required.

The validation is instead much simpler when only single properties, i.e., unary vectors, may be asserted between pairs of intervals. As illustrated in Figure 32, for each path of properties connecting a sequence of proper intervals, the SHACL-SPARQL rules outlined above create two reifications of the property `hasPossibleATR` for each pair of proper intervals involved, one for each direction. The composition table always associates a non-empty set of possible properties with each of these reifications: no cell in the composition table is empty, meaning that each of the 144+26 combinations always produces at least one possible property. It therefore follows that if all pairs of proper intervals in a knowledge graph are connected by *at most a single* path of properties denoting Allen's temporal relations, the knowledge graph will never be invalid: the composition table will never result in a knowledge graph where the reification of at least one `hasPossibleATR` is not associated, through `oneOf`, with any property, which would imply that it is *impossible* to establish a temporal relation between the two proper intervals.

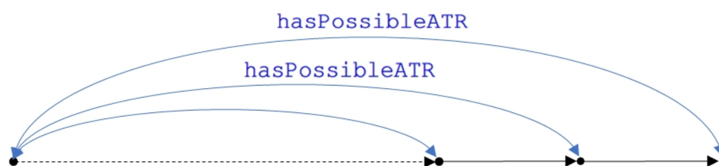

Fig. 32. Path of properties denoting Allen's temporal relations (in black): the composition table always associate a set of possible property to the reification of *each* `hasPossibleATR` property among the pairs of involved intervals.

An invalid pattern can instead occur when there are *two or more* paths between the same pair of proper intervals, as illustrated in Figure 28 above. In such cases, if the two paths do not share any possible property, it indicates that it is *impossible* to establish a property between the two intervals that is consistent with the composition table applied along either of the two paths. The SHACL shape in (45) specifically detects such patterns, where no such a shared property exists (see clause `NOT EXISTS{$this :oneOf ?atr. ?hpatr :oneOf ?atr}`).

There is only one remaining case to consider in Figure 28 and Figure 32: when the two proper intervals at the endpoints of the path(s) are the same, meaning that the path is a *cycling* path.

As illustrated in Figure 33, a cycling path occurs when the last proper interval of one path (shown in red in the figure), which is concatenated with another path (shown in green in the figure), also belongs to the latter. The SHACL-SPARQL rule shown above in (46), the only one that conjoins paths of possible properties denoting Allen's temporal relations, identifies such patterns by querying the values of the property `include`. If a cycling path is detected, the rule marks the newly conjoined path as an instance of the class `ContainsCyclingPath`.

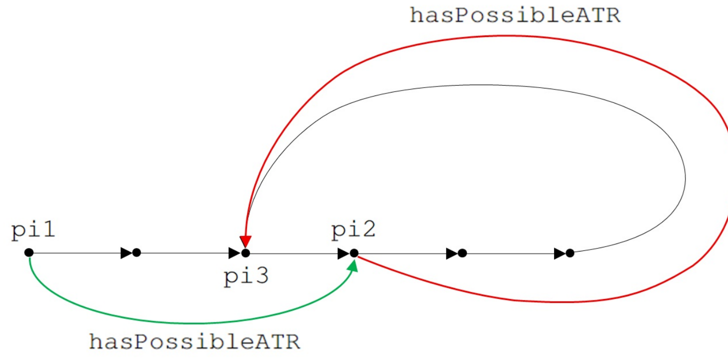

Fig. 33. Reifications of `hasPossibleATR` properties associated to a path of Allen's temporal relations that includes a cycle.

We observe now that the set of possible properties computed through the composition table for the conjunction of the green and the red paths in Figure 33 coincides with the set of possible properties computed through the composition table for the conjunction of the orange and the purple paths in Figure 34, being the purple path a cycling path between `pi3` and itself.

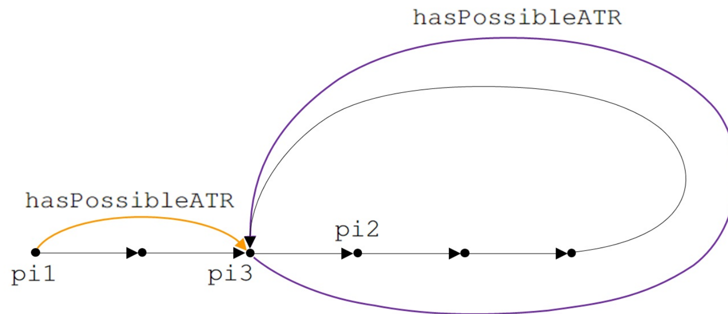

Fig. 34. Reifications of `hasPossibleATR` properties associated to a path of Allen's temporal relations that includes a cycle.

The set of possible properties associated with the purple path in Figure 34 must include the property `intervalEquals`, otherwise the SHACL shape in (44) would invalidate the knowledge graph.

If one of the possible properties associated with the purple path is `intervalEquals`, then the set of possible properties associated with the conjunction of the orange and purple paths is a *superset* of the set of possible properties associated with the orange path alone. As explained earlier, composing any property with `intervalEquals` results in the property itself. Consequently, if a property denoting one of Allen's temporal relations is asserted between `pi1` and `pi3`, and this property belongs to the set of possible properties associated with the orange path, it will also belong to the set of properties associated with the conjunction of the orange and purple paths. In other words, this property remains valid both for the orange path alone and for its conjunction with the purple path.

In light of these considerations, we conclude that validating properties denoting Allen's temporal relations with respect to paths that include cycling sub-paths is equivalent to validating these properties with respect to the sub-paths that do not include the cycling paths, along with ensuring that the cycling paths are associated with the property `intervalEquals`, as verified by the SHACL shape in (44). In other words, cycling paths are irrelevant for the validation, except for the requirement that they must include `intervalEquals` among their possible properties.

7.1.1. Termination and computational complexity

The last two sections differ from the previous ones because, in order to validate properties denoting Allen's temporal relations, it was necessary to introduce new individuals in the `CONSTRUCT` clauses of the SHACL-SPARQL rules.

As explained towards the end of the Introduction, similar to standard OWL reasoners such as HermiT [22], which re-execute OWL axioms until no further triples are inferred, the Java procedures available in the GitHub repository programmatically re-execute the SHACL-SPARQL rules until no further triples are inferred.

This iterative re-execution does not pose any issue for the validation of the Time Ontology without the properties denoting Allen's temporal relations. If the knowledge graph contains a set of resources R, and a subset P⊆R of properties, the maximum number of triples that can be established in the knowledge graph is |R|∗|R|∗|P|. Therefore, the SHACL-SPARQL rules introduced in Section 4 and Section 5 above have a complexity of O(|R|∗|R|∗|P|): these rules are re-executed at most until every resource in the knowledge graph is connected to all other resources (including itself) through each of the properties. After that, no more triples can be added, and the Java procedures terminate the re-execution of the rules.

The SHACL shapes and SHACL-SPARQL rules described in the previous section are also not problematic from this perspective because they merely check for disjointness, symmetry, and reflexivity of the properties denoting Allen's temporal relations. As a result, these rules are only executed once for each property denoting one of Allen's temporal relations.

Conversely, as explained in [51], if the SHACL-SPARQL rules *create* new individuals, their iterative execution may result in an infinite loop, as these rules *expand* the set of resources R. To prevent this, the rules must incorporate special mechanisms to avoid the generation of an *infinite* set of resources.

In our formalization, the mechanism to prevent the rules presented in this section from looping infinitely is the class `ContainsCyclingPath`. The SHACL-SPARQL rule in (46), which creates the reification referring to a conjoined path from the reifications of the two sub-paths, contains the following clauses:

```
NOT EXISTS{$this rdf:type :ContainsCyclingPath}
NOT EXISTS{?r2 rdf:type :ContainsCyclingPath}
```

where `$this` and `?r2` are the reifications referring to the two sub-paths. Thanks to these two clauses, the rule in (46) does not trigger in cases where there are cycles in the source sub-paths.

Therefore, the rule in (46) is executed at most N times, where N is the number of occurrences of properties denoting Allen's temporal relations. In the extreme case, all these occurrences will be connected in a path, after which the only way to extend the path is by concatenating a node that has already been visited, thus forming a cycle.

The only other rule that creates new individuals in the knowledge graph is (39). This rule creates the reifications of the initial paths, based on the properties denoting Allen's temporal relations explicitly asserted. However, the `WHERE` part of this rule includes a `NOT EXISTS` clause to ensure that the initial (unary) paths have not already been created. Therefore, this rule is executed only once for each property denoting one of Allen's temporal relations.

8. Conclusions and future works

This paper delivers two key contributions to the state of the art. First, it presents a new SHACL-based version of the Time Ontology. Effective time management is crucial for progress in both academia and industry, as it underpins a wide range of real-world applications. The Time Ontology is widely recognized as the "de facto" standard for representing temporal data in the Semantic Web. However, its current version primarily serves as a terminological vocabulary for labeling instants, intervals, and other temporal concepts, while offering very limited inferencing capabilities. As a result, it permits the assertion of inconsistent data, such as intervals that end before they start.

Second, our research journey into identifying and evaluating suitable SHACL shapes and SHACL-SPARQL rules for validating the Time Ontology led to novel insights into the interplay between validation and inference in SHACL. As we have demonstrated, SHACL shapes alone lack the expressiveness needed to fully validate the Time Ontology. To address this limitation, we leveraged SHACL-SPARQL rules to compute *inferred* knowledge graphs, which were then validated using SHACL shapes. However, our analysis revealed, on the one hand, that while SHACL-SPARQL

rules are sufficient for validating the current vocabulary of the Time Ontology, they may not support more advanced inferences; on the other hand, they introduce computational costs that could be unnecessary in other use cases.

The next two subsections further elaborate about the two contributions, while pointing out some future works.

8.1. Contribution #1: a new SHACL-based version of the Time Ontology

Thus far, the Time Ontology has primarily been used as a terminological vocabulary, providing standardized symbols to label instants, intervals, and other temporal concepts. Harmonizing and standardizing these symbols across use cases is crucial to ensuring interoperability and facilitating data sharing, but Semantic Web technologies offer much more than just terminology management.

A fundamental aspect of formal ontologies is their ability to support reasoning through a *proof system*, i.e., a set of formal rules that allow for the derivation of conclusions, including the detection of inconsistencies, from the axiomatization within the ontology. In fact, the Time Ontology currently has a proof system, based on OWL axioms, that can be used for reasoning over data encoded with its vocabulary. However, as discussed earlier, this proof system is insufficient for handling temporal data. OWL's lack of built-in operators for comparing and manipulating temporal data severely limits its ability to validate and reason over temporal relationships effectively.

This paper also demonstrates that even combining SWRL or similar rule-based languages with OWL, a widely used approach in past Semantic Web literature, is insufficient. SWRL lacks the ability to programmatically create new individuals, which is essential for assigning metadata to properties. In our specific use case, this capability is necessary to properly validate the sub-algebra of Allen's temporal algebra supported by the Time Ontology vocabulary. As discussed in Subsection 6.1, this is a well-known representational challenge in RDF that extends beyond the Time Ontology. It requires the use of RDF reification together with SPARQL to programmatically create anonymous individuals that refer to the reifications. Alternatively, RDF* offers a more concise and readable syntax for associating metadata with statements, avoiding the verbosity of standard RDF reification.

In light of these limitations, this paper proposes an alternative approach: using SHACL shapes and SHACL-SPARQL rules specifically tailored to validate and reason over the fragment of the Time Ontology shown in Figure 1. This fragment includes only those resources related to the `xsd:dateTime` datatype, which is currently the only temporal datatype supported by the official SPARQL v1.1 W3C recommendation.

These SHACL rules and shapes constitute a new, more expressive proof system for the ontology. Unlike OWL, which can only detect a limited set of inconsistencies, our SHACL-based proof system enables a broader and more expressive validation process, ensuring that temporal constraints are correctly enforced. By enriching the ontology with our novel SHACL-based proof system, we provide a means to ensure the correctness and logical consistency of knowledge graphs using the Time Ontology, thereby unlocking new opportunities for its application in both temporal data validation and AI-driven reasoning tasks.

The first direction of future work that we will focus on is extending the formalization proposed in this paper to include the Time Ontology's resources not represented in Figure 1. In addition, we aim to propose extensions to the Time Ontology's in order to expand the knowledge that it can represent.

For example, as our current formalization only encompasses instants, intervals, and their interrelations, the first reasonable extension appears to be the validation of *durations* of intervals. The Time Ontology also includes properties to encode temporal entities' durations, e.g., the property `hasXSDDuration`, which associates instances of the class `TemporalEntity` to values of the datatype `xsd:duration`. Therefore, for instance, if a proper interval begins on 1st January 2024, ends on 1st January 2025, and is associated through the property `hasXSDDuration` with the `xsd:duration` value "`P10DT0H0M0`", which represents 10 days, then the proper interval is invalid: between the two dates there is a full year, not only 10 days.

Calculating the exact duration by only using the operators from the official SPARQL v1.1 specification could be quite labor-intensive, as it would require accounting for different month lengths, leap years, leap seconds, and other complexities of the temporal reference system. However, it is worth noting that several SPARQL implementations, e.g., the TopBraid SHACL Java library v.1.3.2, which we used in our implementation, calculate the duration between two `xsd:dateTime` values by simply subtracting one from the other within a SPARQL query. While this approach is not part of the official SPARQL v1.1 specification, it is still a practical solution. Nonetheless, this method may

not be sufficient to accurately calculate durations in temporal reference systems other than the standard Gregorian calendar, which the Time Ontology supports (cf. [16]).

On the other hand, we will consider extending the Time Ontology to fully implement Allen's temporal algebra, particularly the constraint propagation algorithm proposed in [47]. However, since the full version of this algorithm is NP-hard, as it requires scanning over all possible subsets of intervals in the domain, we may instead opt to implement one of the sub-algebras proposed in the literature that can be processed in polynomial time [52] [53]. Still, implementing any of these (sub-)algebras requires a level of expressivity far beyond what SHACL-SPARQL offers, particularly because they involve *cycles* over sets of intervals, which SHACL-SPARQL cannot express.

To easily implement all these validation checks, as well as others not discussed here, a potential solution could be to use SHACL-X[25], a recent proposal that combines SHACL with JavaScript, thereby enhancing SHACL with the expressive power of a standard programming language like JavaScript.

Other future work includes defining inference rules to integrate the Time Ontology with other ontologies, enabling the representation of fluents relevant to specific domains (see Subsection 2.2 above).

For example, [51] recently proposes an ontology to reason with obligations, permissions and other deontic statements. Importing the Time Ontology within the ontology proposed in [51] is seen as a future work, in order, for instance, to infer from (50.a-b) that John violated the prohibition of entering the park, *but only from 4pm to 5pm*.

(50) a. It is prohibited to enter the park from 3pm until 5pm.

b. John was in the park from 4pm until 6pm.

Assuming that the two sentences in (50) respectively refer to a prohibition that held and to a fact that took place on the 1st January 2025, the two overlapping proper intervals mentioned in (50.a-b) can be represented as follows:

(51)
```
:pi1 rdf:type time:ProperInterval. :pi2 rdf:type time:ProperInterval.
:pi1 time:hasBeginning :b1; time:hasEnd :e1.
:pi2 time:hasBeginning :b2; time:hasEnd :e2.
:b1 time:inXSDDateTime "2025-01-01T15:00:00Z".
:e1 time:inXSDDateTime "2025-01-01T17:00:00Z".
:b2 time:inXSDDateTime "2025-01-01T16:00:00Z".
:e2 time:inXSDDateTime "2025-01-01T18:00:00Z".
:pi1 time:intervalOverlaps :pi2.
```

The RDF triples in (51) are valid, as can be easily verified by first executing the SHACL-SPARQL rules and then applying the SHACL shapes presented in the previous sections. Still, none of the SHACL-SPARQL rules can infer the relevant interval needed to assess John's compliance with respect to (50.a). To address this, we need to introduce an additional SHACL-SPARQL rule that, given two proper intervals that overlap, *creates* a new instance of `ProperInterval` representing the interval shared by the two overlapping ones. This rule could be:

(52)
```
sh:targetSubjectsOf time:intervalOverlaps;
CONSTRUCT{[a time:ProperInterval; time:hasBeginning ?b; time:hasEnd ?e;
           time:intervalFinishes $this; time:intervalStarts ?pi2]}
WHERE{$this time:intervalOverlaps ?pi2; time:hasEnd ?e. ?pi2 time:hasBeginning ?b.
    NOT EXISTS{?pi3 a time:ProperInterval time:hasBeginning ?b; time:hasEnd ?e}}
```

The SHACL-SPARQL rule in (52) creates the following anonymous individual, which represents the interval from 4pm to 5pm, i.e., the one in which John violates the prohibition in (50.a). Note that this new interval finishes at `pi1`, i.e., the interval from 3pm to 5pm and starts at `pi2`, i.e., the interval from 4pm to 6pm.

[25] https://github.com/SHACL-X/shacl-x

(53) `[rdf:type time:ProperInterval; time:hasBeginning :b2; time:hasEnd :e1; time:intervalFinishes :pi1; time:intervalStarts :pi2].`

As explained above, the SHACL-SPARQL rule in (52) is specifically needed for the compliance-checking application envisioned in [51], whereas it may be irrelevant for other applications. Consequently, it was not included in the GitHub repository associated with this paper, which is application-neutral and, therefore, only contains rules for validating the portion of the Time Ontology depicted in Figure 1.

In our future work, we will explore and propose new SHACL-SPARQL rules to enable the use of the Time Ontology in external applications and case studies, such as the one described in [51].

8.2. Contribution #2: novel insights about the interplay between validation and inference in SHACL

Although exploring novel and effective methods for modeling and reasoning with temporal data in the Semantic Web is crucial for many applications, as discussed in the previous subsection, our work on the Time Ontology has also provided broader insights into SHACL itself, specifically how the intricate relationship between validation and inference can be handled. In other words, our work on the Time Ontology can also be viewed as a *case study* for SHACL, offering valuable insights into how the W3C standard should be applied.

In our view, (1) the fully inferred knowledge graph must first be computed, meaning the graph obtained by iteratively applying inference rules until no further triples are inferred; then, (2) the knowledge graph from step (1) can be validated using SHACL shapes.

This two-step approach is *not* included in the W3C Working Group Note from 08 June 2017.[26] In fact, the note even appears to suggest[27] the reverse sequence: first, the knowledge graph should be validated through the shapes, then the rules can be applied to the valid RDF triples.

Note that by "inference rules", we do not necessarily mean "SHACL-SPARQL rules". To validate the current vocabulary of the Time Ontology, we needed rules to create new anonymous individuals, referring to the reifications of properties. This was necessary to represent metadata of properties, particularly for validating Allen's temporal relations within the Time Ontology. As discussed in Subsection 6.1, this capability is possible in SPARQL and RDF*, but it is not supported by standard SHACL shapes, which are limited to the use of `SELECT-WHERE` queries, nor in OWL or other standard rule-based systems such as SWRL. However, Section 7, particularly rules (47) and (46), demonstrates that SPARQL alone is still insufficient because it does not support rule prioritization. To prioritize rules, we used the `sh:order` property provided in SHACL-SPARQL.

On the other hand, as discussed in the previous section, more advanced inferences, such as those required to compute interval durations or to fully implement Allen's temporal algebra, may demand expressive power beyond what SHACL-SPARQL offers. To handle such advanced inferences, SHACL-X appears to be a promising candidate.

Of course, different reasoning languages (RDFS, OWL, SWRL, RDF*, SHACL-SPARQL, SHACL-X, and many others) have distinct computational properties: the more expressive they are, the more they come with computational costs or other undesirable properties, e.g., the possibility of looping infinitely during the iterative re-execution of SHACL-SPARQL rules, as discussed in Subsubsection 7.1.1 above.

These computational costs must be taken seriously, particularly when dealing with large-scale datasets or when advanced inferences are unnecessary. We recognize that SHACL-SPARQL rules are not always required, especially in simpler use cases where standard SHACL shapes suffice or where SHACL shapes are applied only after RDFS, OWL, or SWRL inferences. In fact, our implementation shows that when rules are unnecessary, simpler shapes may be sufficient for validation.

Beyond the specifics of the Time Ontology, our work encourages further exploration of how inference and constraint validation can be balanced in SHACL. We believe this can be achieved by defining different dialects or

[26] https://www.w3.org/TR/shacl-af, retrieved March 10, 2025

[27] See https://www.w3.org/TR/shacl-af/#rules-examples, retrieved March 10, 2025; however, this section is non-normative and thus holds no formal implications (personal communication with Holger Knublauch).

profiles of SHACL, similar to how OWL defines subsets like OWL-Lite and OWL-DL. This would allow the community to select an appropriate level of expressivity and complexity based on the use case, ensuring that computational costs remain manageable. Notably, the SHACL official W3C recommendation prescribes support for different *entailment regimes*[28], although only RDFS and the SPARQL 1.1 entailment regimes[29] are mentioned in the recommendation and the latter explicitly states that "SHACL implementations MAY, but are not required to, support entailment regimes".

In light of this, we see our work as a call to action for the Semantic Web community to systematically investigate the representational requirements of different use cases and identify the minimal expressivity necessary for each, while defining a specific entailment regime accordingly. Whether advanced inferences with SHACL-SPARQL are needed to handle defeasibility or to represent metadata of properties, or SHACL-X is required for implementing Allen's temporal algebra, each use case should be able to rely on a targeted level of expressivity that meets its needs without incurring unnecessary complexity. More implementations should then be developed to support these entailment regimes, similar to how the definition of the different OWL dialects in the past led to the development of several OWL reasoners (HermiT, Pellet, Racer, etc.).

This paper, we hope, serves as a step in that direction, contributing to a broader understanding of the interplay between inference and validation in the Semantic Web.

Acknowledgments

We sincerely appreciate Maxime Jakubowski, Jose Emilio Labra Gayo, and an anonymous reviewer of an earlier version of this paper for their insightful feedback and valuable suggestions.

References

[1] V. Ermolayev, S. Batsakis, N. Keberle, O. Tatarintseva and A. Grigoris, Ontologies of Time: Review and Trends., *International Journal of Computer Science & Applications* **11**(3) (2014).
[2] Y. Shoham, Temporal logics in AI: Semantical and ontological considerations, *Artificial intelligence* **33**(1) (1987).
[3] D. Gabbay, I. Hodkinson and M. Reynolds, *Temporal Logic (Vol. 1): Mathematical Foundations and Computational Aspects*, Oxford University Press, Inc., USA, 1994.
[4] D. Gabbay, M. Reynolds and M. Finger, *Temporal Logic (Vol. 2): Mathematical Foundations and Computational Aspects*, Oxford University Press, United Kingdom, 2000.
[5] K. Rozier, Linear temporal logic symbolic model checking, *Computer Science Review* **5**(2) (2011).
[6] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani and A. Tacchella, Nusmv 2: An opensource tool for symbolic model checking, in: *Computer Aided Verification: 14th International Conference, CAV 2002 Copenhagen, Denmark, July 27–31, 2002 Proceedings 14*, Springer, 2002, pp. 359–364.
[7] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri and S. Tonetta, The nuXmv symbolic model checker, in: *Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings 26*, Springer, 2014, pp. 334–342.
[8] G. Antoniou and F. Van Harmelen, *A Semantic Web primer*, MIT press, 2004.
[9] F. Manola, E. Miller, B. McBride et al., RDF primer, *W3C recommendation* **10**(1–107) (2004), 6.
[10] D.L. McGuinness, F. Van Harmelen et al., OWL web ontology language overview, *W3C recommendation* **10**(10) (2004), 2004.
[11] F. Baader, I. Horrocks, C. Lutz and U. Sattler, *Introduction to description logic*, Cambridge University Press, 2017.
[12] P. Hitzler, M. Krötzsch, B. Parsia, P.F. Patel-Schneider, S. Rudolph et al., OWL 2 web ontology language primer, *W3C recommendation* **27**(1) (2009), 123.
[13] C. Lutz, F. Wolter and M. Zakharyaschev, Temporal Description Logics: a survey, in: *15th International Symposium on Temporal Representation and Reasoning*, IEEE, 2008.
[14] A. Artale and E. Franconi, A survey of temporal extensions of description logics, *Annals of Mathematics and Artificial Intelligence* **30** (2000), 171–210.
[15] A. Artale, R. Kontchakov, F. Wolter and M. Zakharyaschev, Temporal Description Logic for Ontology-Based Data Access, in: *Proc. of the 23rd International Joint Conference on Artificial Intelligence (IJCAI)*, 2013.

[28] https://www.w3.org/TR/shacl/#shacl-rdfs
[29] https://www.w3.org/TR/sparql11-entailment

[16] S. Cox, Time ontology extended for non-Gregorian calendar applications, *Semantic Web* **7**(2) (2016).
[17] J. Allen, Towards a General Theory of Action and Time, *Artificial Intelligence* **23**(2) (1984).
[18] U. Şimşek, K. Angele, E. Kärle, O. Panasiuk and D. Fensel, Domain-Specific Customization of Schema.org Based on SHACL, in: *Proc. of 19th International Semantic Web Conference (ISWC)*, Springer-Verlag, Berlin, Heidelberg, 2020.
[19] P. Pareti and G. Konstantinidis, A review of SHACL: From data validation to schema reasoning for rdf graphs, *Reasoning Web International Summer School* (2021).
[20] J. Anim, L. Robaldo and A. Wyner, A SHACL-Based Approach for Enhancing Automated Compliance Checking with RDF Data, *Information* **15**(12) (2024).
[21] N. Ferranti, J. de Souza, S. Ahmetaj and A. Polleres, Formalizing and Validating Wikidata's Property Constraints using SHACL and SPARQL, *Semantic Web journal* **to appear** (2024).
[22] B. Glimm, I. Horrocks, B. Motik, G. Stoilos and Z. Wang, HermiT: An OWL 2 Reasoner, *Journal of Automated Reasoning* **53**(3) (2014).
[23] D. Wu, H. Wang and A. Tansel, A survey for managing temporal data in RDF, *Information Systems* (2024).
[24] R. Fikes and Q. Zhou, A reusable time ontology, in: *AAAI-2002 Workshop on Ontologies and the Semantic Web*, Citeseer, 2002.
[25] J. Pustejovsky, R. Ingria, R. Sauri, J.M. Castaño, J. Littman, R.J. Gaizauskas, A. Setzer, G. Katz and I. Mani, The Specification Language TimeML., 2005.
[26] R. Baumann, F. Loebe and H. Herre, Ontology of time in GFO, in: *Formal Ontology in Information Systems*, IOS Press, 2012, pp. 293–306.
[27] V. Milea, F. Frasincar and U. Kaymak, tOWL: a temporal web ontology language, *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* **42**(1) (2011), 268–281.
[28] S.-K. Kim, M.-Y. Song, C. Kim, S.-J. Yea, H.C. Jang and K.-C. Lee, Temporal ontology language for representing and reasoning interval-based temporal knowledge, in: *The Semantic Web: 3rd Asian Semantic Web Conference, ASWC 2008, Bangkok, Thailand, December 8-11, 2008. Proceedings. 3*, Springer, 2008, pp. 31–45.
[29] N. Keberle, Y. Litvinenko, Y. Gordeyev and V. Ermolayev, Ontology evolution analysis with OWL-MeT, in: *Proceedings of the International Workshop on Ontology Dynamics (IWOD-07)*, 2007, pp. 1–12.
[30] V. Ermolayev, N. Keberle and W.-E. Matzke, An upper level ontological model for engineering design performance domain, in: *Conceptual Modeling-ER 2008: 27th International Conference on Conceptual Modeling, Barcelona, Spain, October 20-24, 2008. Proceedings 27*, Springer, 2008, pp. 98–113.
[31] V. Ermolayev, N. Keberle and W.-E. Matzke, An ontology of environments, events, and happenings, in: *2008 32nd Annual IEEE International Computer Software and Applications Conference*, IEEE, 2008, pp. 539–546.
[32] Y. Raimond, S.A. Abdallah, M.B. Sandler and F. Giasson, The Music Ontology., in: *ISMIR*, Vol. 2007, Vienna, Austria, 2007, p. 8th.
[33] C. Gutierrez, C. Hurtado and A. Vaisman, Temporal rdf, in: *The Semantic Web: Research and Applications: Second European Semantic Web Conference, ESWC 2005, Heraklion, Crete, Greece, May 29–June 1, 2005. Proceedings 2*, Springer, 2005, pp. 93–107.
[34] M.J. O'Connor and A.K. Das, A method for representing and querying temporal information in OWL, in: *International joint conference on biomedical engineering systems and technologies*, Springer, 2010, pp. 97–110.
[35] C. Tao, W.-Q. Wei, H.R. Solbrig, G. Savova and C.G. Chute, CNTRO: a semantic web ontology for temporal relation inferencing in clinical narratives, in: *AMIA annual symposium proceedings*, Vol. 2010, American Medical Informatics Association, 2010, p. 787.
[36] S. Batsakis and E.G. Petrakis, SOWL: a framework for handling spatio-temporal information in OWL 2.0, in: *Rule-Based Reasoning, Programming, and Applications: 5th International Symposium, RuleML 2011–Europe, Barcelona, Spain, July 19-21, 2011. Proceedings 5*, Springer, 2011, pp. 242–249.
[37] I. Horrocks, P.F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, M. Dean et al., SWRL: A semantic web rule language combining OWL and RuleML, *W3C Member submission* **21**(79) (2004), 1–31.
[38] H.-U. Krieger, A General Methodology for Equipping Ontologies with Time., in: *LREC*, 2010.
[39] M. Klein, D. Fensel, A. Kiryakov and D. Ognyanov, Ontology versioning and change detection on the web, in: *Knowledge Engineering and Knowledge Management: Ontologies and the Semantic Web: 13th International Conference, EKAW 2002 Sigüenza, Spain, October 1–4, 2002 Proceedings 13*, Springer, 2002, pp. 197–212.
[40] J. Tappolet and A. Bernstein, Applied temporal RDF: Efficient temporal querying of RDF data with SPARQL, in: *The Semantic Web: Research and Applications: 6th European Semantic Web Conference, ESWC 2009 Heraklion, Crete, Greece, May 31–June 4, 2009 Proceedings 6*, Springer, 2009, pp. 308–322.
[41] C. Welty, R. Fikes and S. Makarios, A reusable ontology for fluents in OWL, in: *FOIS*, Vol. 150, 2006, pp. 226–236.
[42] S. Batsakis, E.G. Petrakis, I. Tachmazidis and G. Antoniou, Temporal representation and reasoning in OWL 2, *Semantic Web* **8**(6) (2017), 981–1000.
[43] S. Batsakis, I. Tachmazidis and G. Antoniou, Representing time and space for the semantic web, *International Journal on Artificial Intelligence Tools* **26**(03) (2017), 1750015.
[44] A. Preventis, E.G. Petrakis and S. Batsakis, Chronos Ed: A tool for handling temporal ontologies in Protege, *International Journal on Artificial Intelligence Tools* **23**(04) (2014), 1460018.
[45] B. Glimm, I. Horrocks, B. Motik, G. Stoilos and Z. Wang, HermiT: an OWL 2 reasoner, *Journal of automated reasoning* **53** (2014), 245–269.
[46] E. Sirin, B. Parsia, B.C. Grau, A. Kalyanpur and Y. Katz, Pellet: A practical owl-dl reasoner, *Journal of Web Semantics* **5**(2) (2007), 51–53.
[47] J. Allen, Maintaining knowledge about temporal intervals, *Communications of the ACM* **26**(11) (1983).
[48] M. Vilain, H. Kautz and P. van Beek, Constraint Propagation Algorithms for Temporal Reasoning: A Revised Report, in: *Readings in qualitative reasoning about physical systems*, Morgan Kaufmann Publishers Inc., 1990, pp. 373–381.

[49] L. Robaldo, Towards compliance checking in reified I/O logic via SHACL, in: *Proc. of 18th International Conference for Artificial Intelligence and Law (ICAIL 2021)*, J. Maranhão and A.Z. Wyner, eds, ACM, 2021.
[50] L. Robaldo and G. Pozzato, Handling irresolvable conflicts in the Semantic Web: an RDF-based conflict-tolerant version of the Deontic Traditional Scheme, *https://arxiv.org/abs/2411.19918. The paper is currently under review in the Journal of Logic and Computation (https://academic.oup.com/logcom)* (2024).
[51] L. Robaldo, S. Batsakis, R. Calegari, F. Calimeri, M. Fujita, G. Governatori, M. Morelli, F. Pacenza, G. Pisano, K. Satoh, I. Tachmazidis and J. Zangari, Compliance checking on first-order knowledge with conflicting and compensatory norms - a comparison among currently available technologies, *Artificial Intelligence and Law* **32** (2023).
[52] A. Krokhin, P. Jeavons and P. Jonsson, Reasoning about temporal relations: the tractable subalgebras of Allen's interval algebra, *Journal of the ACM* (2003), 591–640.
[53] G. Rosu and S. Bensalem, Allen Linear (Interval) Temporal Logic - Translation to LTL and Monitor Synthesis, in: *Proc. of 18th International Conference Computer Aided Verification CAV*, T. Ball and R. Jones, eds, Lecture Notes in Computer Science, Vol. 4144, Springer, 2006, pp. 263–277.