

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: s.batsakis@hud.ac.uk

^c*Huddersfield University, UK*

E-mail: s.batsakis@hud.ac.uk

Abstract. This paper presents a novel framework to validate the Time Ontology (a.k.a. OWL Time, <https://www.w3.org/TR/owl-time>), which is currently a W3C candidate recommendation draft for representing temporal data in the Semantic Web. The framework is based on SHACL shapes and SHACL-SPARQL rules. These are used together to invalidate knowledge graphs that OWL is unable to identify as such due to its lack of expressivity, specifically its lack of operators to compare and work with temporal data. Besides providing a useful tool to process temporal data encoded in RDF within applications, our research work also sheds some light on how using SHACL shapes and SHACL-SPARQL rules *together*, in order to capture the proper interplay between validation and inference on knowledge graphs. The SHACL shapes and the SHACL-SPARQL rules that define the proposed framework are freely available on the GitHub repository <https://github.com/liviorobaldo/TimeOntologyInSHACL>, together with Java programs and clear instructions to process 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, processing of historical data, diagnostic systems, compliance checking, real-time systems, computer aided engineering, and any other application that requires to model actions, changes, or behaviours. Representing time is in particular pivotal for *accountability purposes*, namely to timestamp the events in the state of affairs fit to enable post-analyses of the data.

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, 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.

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

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

4 The Semantic Web also requires a corresponding representation of time [8]. Semantic Web standards are based on
 5 Resource Description Framework (RDF) [9] for representing interconnected data on the Web and on Web Ontology
 6 Language (OWL) [10] for reasoning over data encoded in RDF. OWL is based on the theoretical work done in
 7 Description Logics [11]; currently, the mostly used version of OWL is perhaps OWL-2 [12]. Although the theoretical
 8 properties of extending Description Logics with the expressiveness of Temporal Logics has been investigated [13],
 9 [14] such extensions have no been added to the family of Semantic Web standards.

10 Concerning instead the representation of temporal data in the Semantic Web, while reviewing the relevant liter-
 11 ature, briefly illustrated below in section 2, we realized that proposed approaches mainly focus on defining vocabu-
 12 laries of RDF resources for *representing* temporal knowledge while instead neglecting the *inferences* that can be
 13 drawn from the represented knowledge. Indeed, the same can be claimed about the Semantic Web in general, not
 14 only about the representation of temporal knowledge (cf. [8]). In light of this, in our view many more efforts should
 15 be invested by the Semantic Web community for researching automated reasoning on knowledge graphs.

16 Some proposals for reasoning on temporal data with OWL have been made, e.g., [13] and [14], which proposed
 17 Temporal Description Logics as a formal solution to represent time in Description Logic. Similarly, [15] propose an
 18 extension of OWL2-QL for ontology-based data access. Nevertheless, these and other similar initiatives have been
 19 only developed at the theoretical level, mainly with the aim of addressing computational complexity and decidability
 20 issues, while they lack implementations.

21 The starting point of this paper is instead that OWL is intrinsically *inadequate* for representing temporal data, for
 22 the mere simple reason that its vocabulary does not include constructs to compare and process quantitative temporal
 23 entities such as dates, hours, minutes, etc. In our view, the lack of these constructs prevents in particular the use of
 24 OWL for the accountability purposes mentioned at the beginning of the introduction.

25 This paper investigates in particular the Time Ontology¹, which is currently a W3C candidate recommendation
 26 draft and is considered as a sort of “de facto” standard for representing temporal knowledge in the Semantic Web.
 27 Many ontologies, in different domains, import the Time Ontology to represent temporal entities.

28 The vocabulary of the Time Ontology is rather rich as it includes RDF resources to represent, both quantitatively
 29 and qualitatively, instants and intervals of time in several reference systems such as the standard Gregorian calendar
 30 as well as non-Gregorian calendars, Unix-time, and geologic time [16]. Notably, the Time Ontology vocabulary
 31 includes RDF properties that implement the well-known Allen’s interval algebra [17], widely used as a basis for
 32 reasoning about temporal descriptions of events. However, not all RDF resources of the Time Ontology are relevant
 33 for this paper, therefore only a *fragment* of the Time Ontology, presented below in section 3, will be used.

34 The Time Ontology is currently implemented in OWL-2 DL. Therefore, as explained above, it only enables
 35 inferences that are irrelevant from the perspective of temporal management. As a consequence, the vocabulary of
 36 the Time Ontology even allows for the encoding of absurd temporal data such as intervals that end before they start
 37 or pairs of intervals that are disjoint but, at the same time, they overlap. OWL inferences are unable to flag these
 38 intervals as inconsistent or, more specifically, as *invalid*, thus significantly diminishing the usefulness of the Time
 39 Ontology within existing applications.

40 It must be however observed that OWL has been designed for *reasoning* with RDF knowledge graphs while for
 41 *validating* RDF knowledge graphs another standard has been releases through an official W3C recommendation:
 42 the Shapes Constraint Language (SHACL).

43 As part of the research activity presented in this paper, we developed a set of SHACL shapes to validate the
 44 fragment of the Time Ontology presented below in section 3. Contrary to OWL, SHACL incorporates SPARQL
 45

46
 47
 48
 49
 50
 51 ¹<https://www.w3.org/TR/owl-time>

v1.1, whose official² vocabulary includes operators to compare values of the `xsd:dateTime` datatype³ as well as functions to work with these values.

Nevertheless, it was immediately clear to us that SHACL shapes do not suffice to identify all possible invalid temporal data that can be encoded through the RDF resources in the Time Ontology. In other words, SHACL shapes are not enough expressive to identify some patterns of invalid RDF triples (examples will be provided below). In order to identify these patterns, it is instead necessary to *infer* further triples from the explicitly asserted ones; then, it is possible to write SHACL shapes that identify the invalid patterns *within the inferred knowledge graph*.

In order to do so, since we could not use OWL, which does not allow for inferences on temporal data, as explained above, we used SHACL shapes in combination with *SHACL rules*, defined in the W3C Working Group Note 08 June 2017⁴. Note that SHACL rules have not been (yet?) released as part of an official W3C recommendation.

In particular, we used SHACL-SPARQL rules⁵, which are also based on SPARQL, as the name suggests, and so they are again capable of processing `xsd:dateTime` values. This is not the first paper that does so; several other approaches from recent literature, e.g., [18], [19], [20], [21], [22], and [23], combine SHACL shapes with SHACL-SPARQL rules, thus acknowledging the intimate interplay between validation and inference.

SHACL-SPARQL rules are SPARQL queries in the form `CONSTRUCT-WHERE` that create new triples in the knowledge graph on the basis of the triples already asserted therein. Nevertheless, while the aforementioned Working Group Note prescribes that SHACL-SPARQL rules must be applied only to valid RDF triples, i.e., *after* the asserted triples have been validated through the SHACL shapes, the present paper provides evidence that it should be instead *the other way round*: first the inferred knowledge graph must be generated through the SHACL-SPARQL rules, then the inferred knowledge graph must be validated through the SHACL shapes.

In other words, SHACL rules compensate the lack of expressivity of SHACL shapes, i.e., they can add to the knowledge graph inferred triples that “help” the SHACL shapes identify the invalid patterns. On the other hand, from the point of view of the validation the order of execution does not matter: SPARQL queries in the form `CONSTRUCT-WHERE` only *add* triples to the initial knowledge graph; therefore, in cases where the latter already includes invalid triples, these will be also included in the inferred knowledge graph.

In addition, similarly to what is done in standard OWL reasoners, e.g., Hermit [24], which re-executes the OWL axioms until no further new triple is inferred, the present paper provides evidence that also SHACL-SPARQL rules should be re-executed until no further triple is inferred, prior to validation. This is not prescribed in the mentioned Working Group Note either. Nor the available libraries that process SHACL shapes and rules, e.g., the TopBraid SHACL Java library v.1.3.2⁶, which we used in our implementation, execute the SHACL rules multiple times, but only once. Therefore, the Java programs available on the GitHub repository associated with this paper first re-execute the SHACL rules *programmatically* until no further triple is inferred, then they validate the inferred knowledge graph on the SHACL shapes.

In light of this, it should be now clear to the reader that this paper does not only present an extension of the Time Ontology that identifies invalid temporal entities. This paper also sheds some light on how the interplay between validation and inference should be carried out *in general*, i.e., not only with respect to the Time Ontology. Specifically, we believe that what our implementation on GitHub “forcibly” does programmatically should be instead officialized by W3C. In other words, in our view W3C should release a new recommendation of SHACL that encompasses both shapes and rules and that stipulates what has been explained above: first the inferred knowledge graph is generated through the iterative execution of the SHACL rules, then the inferred knowledge graph is validated through the SHACL shapes. Available libraries executing SHACL shapes and rules should be updated accordingly, in order to carry out these steps through a single procedure.

²Actually, most available SPARQL implementations “unofficially” extends the coverage of the SPARQL v1.1 operators and functions to other datatypes, e.g., `xsd:date` or `xsd:duration`. However, the official W3C recommendation only consider `xsd:dateTime` (see <https://www.w3.org/TR/sparql11-query/#OperatorMapping>). Perhaps, future versions of the standard should also consider other datatypes.

³<https://www.w3.org/TR/xmlschema-2/#dateTime>

⁴<https://www.w3.org/TR/shacl-af>, retrieved June 1, 2024

⁵<https://www.w3.org/TR/shacl-af/#SPARQLRule>

⁶<https://repo1.maven.org/maven2/org/topbraid/shacl/1.3.2>

The rest of the paper is organized as follows. The next section will quickly review current literature on representing time in the Semantic Web. As already mentioned earlier, the Time Ontology is considered as a kind of standard “de facto” on this topic, as it defines and categorizes basic units of time such as instants, intervals, dates, etc. and so it is often imported in contemporary ontologies that require a representation of temporal data. Section 3 will then focus on describing the Time Ontology and, in particular the fragment of the Time Ontology that will be considered in the proposed formalization. The vocabulary of the Time Ontology is indeed rather rich and it allows to represent time in several reference systems, different from the standard Gregorian calendar; however, the objective of this paper is not the one to exhaustively formalize the whole Time Ontology, so that it will only focus on the minimal fragment needed to achieve its aims.

Sections 4, 5, 6, and 7 contain then the core of the research presented here: the formalization in SHACL shapes and SHACL-SPARQL rules of the fragment of the Time Ontology presented in Section 3, fit to validate temporal data encoded with its vocabulary. Indeed, these four sections could be merged into a single one; however, since their content is rather long, we chose to split it in four parts in order to increase readability.

Section 8 concludes the paper and points out some future works. As already explained above, the research findings of this paper are twofold: the paper does not only provide a novel formalization of temporal data in the Semantic Web that, in our view, is much more useful than the current version of the Time Ontology, from an application perspective. Our work also allowed us to find out how SHACL shapes and SHACL-SPARQL rules should be applied *in general*, i.e., not only for the Time Ontology, in order to capture the proper interplay between validation and inference on knowledge graphs. The advocated “modus operandi” is *not* stipulated by the W3C Working Group Note 08 June 2017 (retrieved on June 1, 2024), which contains the current specifications of the SHACL-SPARQL rules. We therefore advocate a new revised version of these specifications that encompasses the findings of the present paper.

2. Background: representing temporal data in the Semantic Web

There are two main components for representing time in a machine-readable format, as per the use of applications:

- 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 their relations are defined using temporal ontologies while the usage of these properties in specific application domains is a dimension orthogonal to the construction of temporal ontologies. Thus temporal ontologies contain the definitions of temporal intervals and temporal points among others and these definitions can be used for representing a temporal property in various different ways using, e.g., 4D-fluents or reification. In the following these two aspects of temporal representation i.e., definitions of temporal concepts in temporal ontologies and usage of these definitions in domain specific use cases are presented.

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 object are dynamic. Thus, many temporal ontologies have been proposed in the literature [1]. Among these, the Time Ontology in OWL, also know as OWL-Time⁷, is the most widely used⁸. 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

⁷<https://www.w3.org/TR/owl-time/>

⁸Example applications are listed in: https://www.w3.org/2015/spatial/wiki/OWL_Time_Ontology_adoption

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 [25], TimeML [26] that offer a translation to DAML-OIL, the predecessor of OWL, GFO-Time [27] which is part of the upper ontology GFO, TOWL [28] which extends OWL with temporal constructs but is also not compliant with standard Semantic Web tools and TL-OWL [29] which also extends Semantic Web standards with temporal concepts, similarly to OWL-Met [30]. In PSI-ULO [31] temporal concepts are part of an upper ontology defined in PSI-Time [32], and in TimeLine ontology [33] definitions for temporal concepts for digital music are provided. Temporal RDF [34] proposes extending RDF with temporal annotations while SWRL-Time [35], CNTRO [36] and SOWL [37] define reasoning mechanisms based on SWRL which is not a widely supported W3C recommendation for rule definition.

Therefore, when definitions of temporal concepts are needed many 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.

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 temporal concepts involved (e.g. the temporal instant that an event occurs or the temporal interval that a dynamic property holds) are provided by temporal ontologies but their usage in specific applications is not straightforward since fluent properties are not binary (e.g. they involve a subject, an object and a temporal instant or interval), thus not represented directly as an object property or datatype property of a class. This results to many different approaches using temporal concepts in practice.

The Semantic Web standards for represent data are RDF and RDFs [38]. The latter offers a vocabulary and a semantics related to taxonomic relations between classes and properties and domains and ranges of properties represented using RDF. RDFs allows for limited reasoning over Semantic Web data. To achieve more advanced forms of reasoning, OWL was proposed [12]. OWL allows for more complex definitions of classes and properties with the current version (OWL 2) being based on the expressive SROIQ description logic [39]. OWL offers increased expressiveness, while retaining decidability. Nevertheless, direct temporal semantics such as those offered by Temporal Descriptions Logics [13], [14] are not integrated into OWL, thus representation of temporal properties over RDF data must be achieved within the RDF/RDFs/OWL framework and newer standards such as SHACL. In addition, large scale reasoning using OWL, which is crucial when working on large knowledge graphs, is not efficient when dealing with full OWL 2 expressiveness. Large scale reasoning over Semantic Data is achieved when applying a trade-off between expressiveness and efficiency and this is surveyed in detail in [40]. Thus, reasoning over large knowledge graphs with temporal information cannot be achieved using OWL reasoners.

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 [41] (standard RDF is based in triple stores), versioning [42], which is based on creating a new copy of the knowledge base when a property is modified, named graphs [43], the generic method of reification, and the 4D fluents approach proposed in [44]. Various methods are presented and compared in [45] and they have been extended to cover spacial properties of objects in [46]. The work in [45] 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. Because of certain limitations of OWL regarding chains of properties, temporal reasoning in [45] and also when using SWRL-Time [35] and CNTRO [36] is achieved using the Semantic Web Rule Language (SWRL) [47], but this approach was inefficient for large scale reasoning. In [48], the CHRONOS ED tool was proposed offering increased performance compared to the work in [45] but this approach was rather ad-hoc as it was based on specialized reasoning software, compatible with specific ontology instead of using generic semantic reasoners such as Hermit [49] and Pellet [50], thus limiting its overall applicability. Notice that temporal ontologies that provide definitions for temporal instances and intervals can be used in conjunction with the above-mentioned approaches, provided that

1 the ontologies include reasoning capabilities that can be used for inferences and validation tasks over representations 1
2 of fluent properties. Temporal data representation and management is surveyed in [51]. 2

3 Summarizing, although there are various ways to import temporal concepts in domain specific ontologies and 3
4 there are several temporal ontologies defining these concepts with the Temporal Ontology being the most widely 4
5 used since it is a W3C candidate recommendation draft, there are various issues remaining to be addressed because 5
6 of the limited reasoning capabilities currently offered by these ontologies. 6

7 Furthermore, time is not integral in Semantic Web standards and remaining compatible with such standards calls 7
8 for representations that add reasoning rules into existing ontologies instead of using specialized reasoning software. 8
9 SWRL has been used for representing temporal reasoning rules but corresponding approaches were not efficient. 9
10 The focus of the current work is to use SHACL shapes and SHACL-SPARQL rules to this end. 10
11

12 3. The Time Ontology 12

13
14
15
16 The Time Ontology is currently a W3C candidate recommendation draft⁹ publicly available online and download- 16
17 able in Turtle format¹⁰. Its IRI is “<http://www.w3.org/2006/time#>”; in this paper, as well as in the as- 17
18 sociated GitHub repository, we will refer to this IRI with the prefix “time:”. The version of the Time Ontology 18
19 used in this paper is the one retrieved on June 1, 2024. A copy of this version is available on the GitHub repository 19
20 associated with this paper; of course, subsequent versions of the Time Ontology could not be compatible with the 20
21 implementation proposed in this paper. 21

22 While the full formal definitions of the ontology’s resources (classes, individuals, and properties) are avail- 22
23 able online at the Time Ontology’s homepage, this section will focus on the resources that may be processed 23
24 via SHACL, namely the ones associated with the `xsd:dateTime` datatype. As pointed out in the Introduction, 24
25 `xsd:dateTime` is the single temporal datatype for which SPARQL 1.1 defines comparison operators¹¹, therefore 25
26 it is the single one for which it is currently possible to assert SHACL shapes and rules¹². 26

27 Figure 1 shows the classes and properties of the Time Ontology that we consider in this paper, namely the ones 27
28 connected with the `xsd:dateTime` datatype. The top class is `TemporalEntity`, which has two direct sub- 28
29 classes¹³: `Instant` and `Interval`. These represent the two main conceptual entities of the ontology. Individuals 29
30 of `Instant` refer to exact moments in time while individuals of `Interval` refer to spans of infinite and con- 30
31 tiguous instants between a start and an end instant. The latter are respectively identified by the object properties 31
32 `hasBeginning` and `hasEnd`, inherited from the upper class `TemporalEntity`. 32

33 The class `Interval` has a direct subclass called `ProperInterval`, which is defined as an `Interval` “for 33
34 which the value of the beginning and end are different”. This constraint does not hold for the upper class `Interval`, 34
35 whose instances can then have the *same* `Instant` as beginning and end, in which case they are instants themselves. 35
36 For this reason, the two classes have been declared as *disjoint* (`owl:disjointWith`) in the Time Ontology. 36
37

38 This is one of the only two disjunctions currently asserted in the Time Ontology. The other one holds between the 38
39 two properties `intervalEquals` and `intervalIn`, which are related via `owl:propertyDisjointWith`. 39
40 Therefore, the Time Ontology currently permits any assertion but those non-compliant with these two disjunctions. 40

41 In addition, the Time Ontology also includes several further `owl:allValuesFrom`, `owl:hasValue`, and 41
42 `owl:cardinality` restrictions as well as several `owl:FunctionalProperty`, `owl:inverseOf`, etc. 42
43 properties, but most of these are not relevant for this paper’s objectives. 43
44

45 ⁹<https://www.w3.org/TR/owl-time> 45

46 ¹⁰<https://www.w3.org/TR/turtle> 46

47 ¹¹<https://www.w3.org/TR/2013/REC-sparql11-query-20130321/#OperatorMapping> 47

48 ¹²It is worth noticing that most current SPARQL implementations also support comparisons among other temporal datatypes, e.g., `xsd:date` 48
49 and `xsd:dateTimeStamp`. In this paper we will adhere to the official W3C recommendations although some of their limitations have been 49
50 already “de facto” overcome. 50

51 ¹³`TemporalEntity` is actually defined as the *union* of `Instant` and `Interval`; in other words, all instances of `TemporalEntity` 51
are also instances of either `Instant` or `Interval` (or both). 51

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); further discussion about this property can be found in subsection 3.1 below. 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 than `TemporalEntity`; the property `inside` connects intervals with instants occurring therein. The properties `after` and `before` allows for the 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 3.2 below.

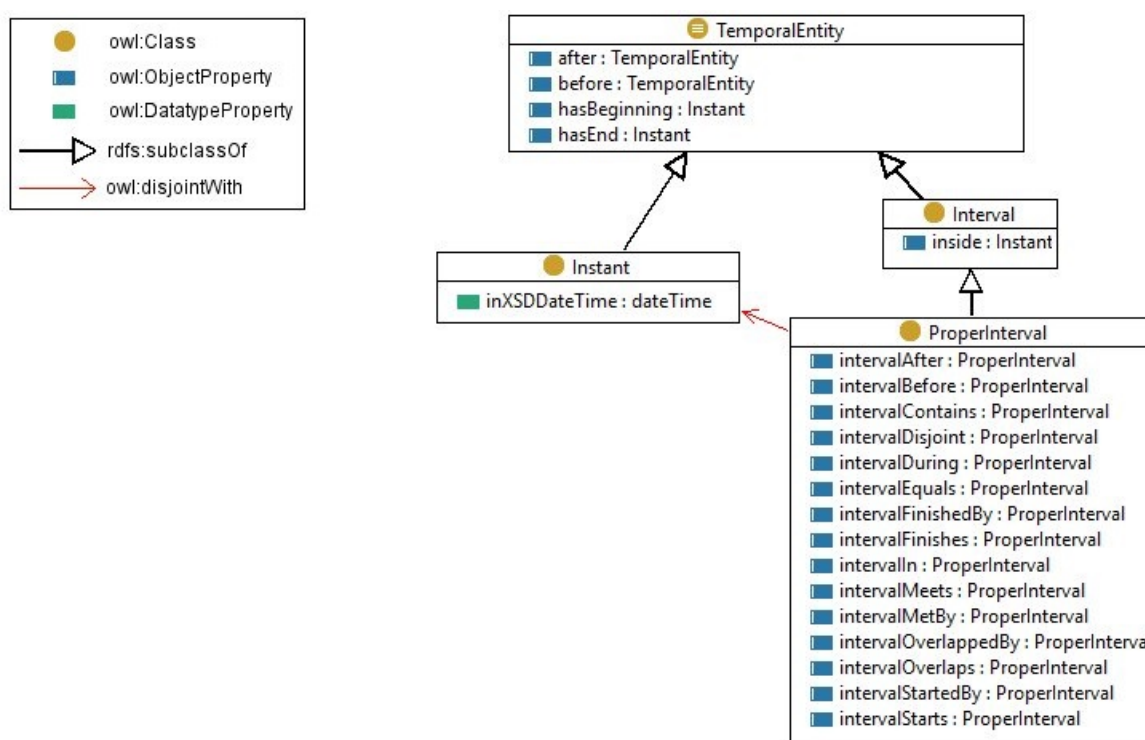


Fig. 1. Classes and properties of the Time Ontology considered in this paper. This figure is a reduced version of the figure at <https://www.w3.org/TR/owl-time/#topology> (retrieved on June 1, 2024)

It is easy to re-implement RDFs and OWL axioms as SHACL shapes or SHACL-SPARQL rules. However, not all RDFs and OWL axioms in the Time Ontology have been re-implemented in the proposed account. Our implementation only includes:

- (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.

These shapes and rules are however omitted from this paper, although the reader can still find them in the GitHub repository. Also the property `owl:propertyDisjointWith` relating `intervalEquals` and `intervalIn`

is encompassed in our implementation; however, we will propose in section 6 below a more general scheme to relate the properties listed in Figure 1 within the box associated with `ProperInterval`, which encompasses several other disjunctions among them, 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 next sections. Before then, however, it is worth spending some further words about the `xsd:dateTime` datatype and the Time Ontology’s properties denoting Allen’s temporal relations. These will be respectively done in the two next subsections.

3.1. The `xsd:dateTime` datatype

As we observed above, Figure 1 contains a single datatype property: `inXSDDateTime`, which associate instances of the class `Instant` to `xsd:dateTime` values. We repeat again that `xsd:dateTime` the single datatype for which the official SPARQL 1.1 recommendation define comparison operators.

`xsd:dateTime` is defined¹⁴ as a string with the following pattern, in which the round brackets “(. . .)” mark optional elements:

$$(-) yyyy-mm-ddThh:mm:ss(.s)(zzzzzz)$$

`xsd:dateTime` may start with an optional “-” that mark negative date-time descriptions; in case this is omitted, the date-time description is positive. Then, `yyyy`, `mm`, and `dd` are mandatory fields, separated by “-”, that respectively specify the year, the month, and the day position in the Gregorian calendar. `dd` is followed by a separator “T” and then by the mandatory fields `hh`, `mm`, and `ss`, separated by “:”, that respectively specify the hours, the minutes, and the seconds. Finally, `.s` and `zzzzzz` are optional fields that respectively specify fractional seconds and the timezone. The latter can be equal to `Z`, in which case the timezone is the Coordinated Universal Time (UTC), i.e., the timezone in Greenwich, or a string with the patten “(+|-)hh:mm”, which represents the either positive (+) or negative (-) offset from UTC, expressed in hours and minutes. For instance, “2002-10-10T07:00:00Z” represents 10th October 2002 at 7.00am in Greenwich; this corresponds to “2002-10-10T12:00:00+05:00”, i.e., 10th October 2002 at 12.00am in Maldives islands.

SPARQL 1.1 is able to compare pairs of `xsd:dateTime` values even if they feature different timezones; this is done by converting all timezones to UTC prior the comparison. However, the timezone is *optional* in `xsd:dateTime` and, although the algorithm that compares `xsd:dateTime` values provides an output even in cases where the timezone is missing¹⁵, this output is meaningless from an application perspective.

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 exactly as the `xsd:dateTime` datatype but it compulsorily requires to specify the timezone¹⁶.

From the point of view of the research presented in this paper, this issue does not represent a critical concern. All `xsd:dateTimeStamp` values can be converted into `xsd:dateTime` values through the SPARQL 1.1 function `strdt`¹⁷. As this would be just a mere technicality, for the sake of simplicity and readability, the SHACL shapes and rules presented below will simply use `xsd:dateTime` but we will include a SHACL shape that compulsory requires the `xsd:dateTime` values to specify the timezone, shown below in (4).

¹⁴<https://www.w3.org/TR/xmlschema-2/#dateTime>

¹⁵See paragraph “3.2.7.4 Order relation on `dateTime`” at <https://www.w3.org/TR/2004/REC-xmlschema-2-20041028/#dateTime>

¹⁶<https://www.w3.org/TR/xmlschema11-2/#dateTimeStamp>

¹⁷<https://www.w3.org/TR/sparql11-query/#func-strdt>

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

Allen's temporal relations are thirteen disjoint relations that can hold between two temporal intervals. These relations were originally defined in [17] and they are still considered nowadays as a reference system in the literature on temporal reasoning. Each of these relations is denoted 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.

The other twelve relations are pairwise clustered: six of them, i.e., `Before`, `During`, `Meets`, `Starts`, `Finishes`, and `Overlaps`, respectively denoted by the properties `intervalBefore`, `intervalDuring`, `intervalMeets`, `intervalStarts`, `intervalFinishes`, and `intervalOverlaps` of the Time Ontology, may be thought as the "main" ones, while the other six, i.e., `After`, `Contains`, `MetBy`, `StartedBy`, `FinishedBy`, and `OverlappedBy`, respectively denoted by the properties `intervalAfter`, `intervalContains`, `intervalMetBy`, `intervalStartedBy`, `intervalFinishedBy`, and `intervalOverlappedBy` of the Time Ontology, are their *inverse* properties.

The six "main" properties are defined as follows:

- (2) a. `Before`: if an interval is related to another interval through `Before`, then it temporally occurs before the latter.
- b. `During`: if an interval is related to another interval through `During`, then the latter *properly* contains it.
- c. `Meets`: if an interval is related to another interval through `Meets`, then its end coincides with the beginning of the latter.
- d. `Starts`: if an interval is related to another interval through `Starts`, then the beginnings of the two intervals coincide while the end of the former temporally occurs before the end of the latter.
- e. `Finishes`: if an interval is related to another interval through `Finishes`, then the ends of the two intervals coincide while the beginning of the former temporally occurs after the beginning of the latter.
- f. `Overlaps`: if an interval is related to another interval through `Overlaps`, then its beginning temporally occurs before the beginning of the latter and its end temporally occurs before the end of the latter.

Figure 2 graphically displays (2.a-f). As stated earlier, the other six missing relations are just the inverse of (2.a-f); therefore, for example, if the interval T1 is `Before` the interval T2, then the interval T2 is `After` the interval T1.

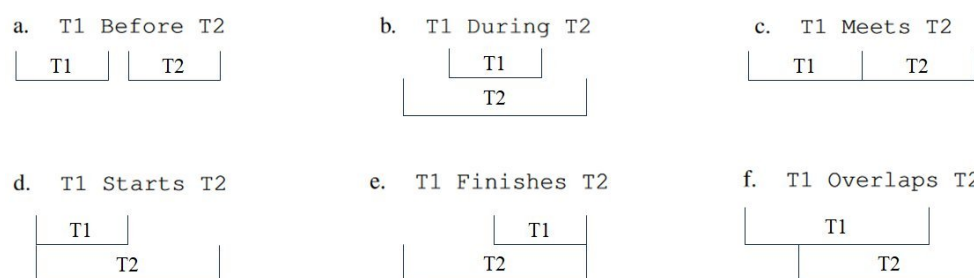


Fig. 2. Graphical representation of (2.a-f)

In addition to the thirteen basic temporal relations, [17] 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:

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

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`. Nevertheless, [17] does not include a corresponding temporal relation.

In addition to the thirteen basic temporal relations, Allen defines a specific *algebra* on them, namely a composition table that, given the temporal relation `r1` between two intervals `T1` and `T2` and the temporal relation `r2` between `T2` and a third interval `T3`, indicates the possible temporal relations between `T1` and `T3`.

For instance, as it is shown in Figure 4, by taking both `r1` and `r2` as `Before`, the temporal relation between `T1` and `T3` is again `Before`.



Fig. 3. Transitivity of `Before`: if `T1` occurs before `T2` and `T2` occurs before `T3`, then `T1` occurs before `T3`.

Of course, the example in Figure 4 is a very easy one, because it is well-known that `Before` is a transitive relation. On the other hand, with other combinations of `r1` and `r2` there could be more options for the temporal relation between `T1` and `T3`, which requires some more careful inspection.

For instance, 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 4, 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 `T1` and `T3`, shown at the bottom of Figure 4: either `T1` occurs `During` `T3`, or it `Starts` at `T3`, or it `Overlaps` with it.

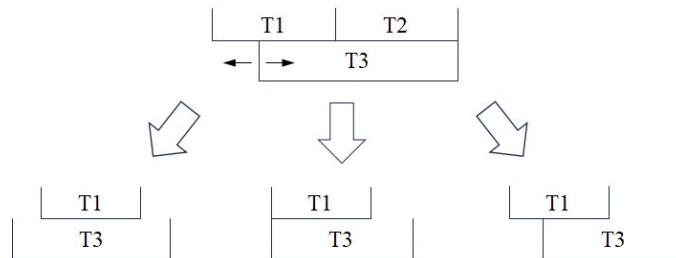


Fig. 4. Transitivity of `Before`: if `T1` occurs before `T2` and `T2` occurs before `T3`, then `T1` occurs before `T3`.

The composition table defining Allen's interval algebra, originally defined in [52], is reported in Table 2. 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 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.

Below in section 7, Table 2 will be implemented in terms of SHACL-SPARQL rules. Before then, however, the next two sections will introduce SHACL shapes and SHACL-SPARQL rules to respectively validate the temporal entities and the Allen's temporal relations with respect to the instants that occur therein.

| T1 \ r1 T2 | T2 r1 T3 | < | > | 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 mid f | > | > | > oi mid f | > | > oi mid f | > | > | > oi mid f | > | > |
| During (d) | < | > | d | any | < o m d s | > oi mid f | < | > | d | > oi mid 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 | 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 | 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 | 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 | di |
| Finishes (f) | < | > | d | > oi mi di si | o d s | > oi mi | m | > | d | > oi mi | f | f fi = | 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 | fi |

Table 1

The composition table for the twelve Allen's temporal relation; Equal (=) has been omitted due to space constraints.

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.

In Figure 1, only instances of the class `Instant` can be associated with `xsd:dateTime` values, 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:property[sh:datatype xsd:dateTime;
                  sh:message "Invalid datatype: xsd:dateTime is required"]].
```

Furthermore, since, as mentioned at the beginning of subsection 3.1 above, the *timezone* is *optional* for the `xsd:dateTime` datatype, for practical reasons we also added the SHACL shape in (4), which compulsory requires the objects of `inXSDDateTime` to specify the *timezone*. The SHACL shape in (4) converts the `xsd:dateTime` values as objects of `inXSDDateTime` into strings via the SPARQL 1.1 function `str`¹⁸, then it checks whether the *timezone* is specified or not by using the SPARQL 1.1 function `regex`¹⁹.

```
(4) [rdf:type sh:NodeShape;
      sh:targetObjectsOf time:inXSDDateTime;
      sh:sparql[sh:prefixes ... ;
                sh:select """SELECT $this
                           WHERE{FILTER(!regex(str($this), "(Z|(\+|-)[0-9]2:[0-9]2)$"))}""";
                sh:message "Invalid '{$this}': it does not specify the timezone."].
```

The third SHACL shape that we defined on the property `inXSDDateTime` is shown in (5): if the same instant is associated with two or more `xsd:dateTime` values via `inXSDDateTime`, these values must be the same.

```
(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}."].
```

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

¹⁸<https://www.w3.org/TR/sparql11-query/#func-str>

¹⁹<https://www.w3.org/TR/sparql11-query/#func-regex>

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.

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 (6) is invalid because it begins at two instants associated each with a *different* `xsd:dateTime` value.

```
(6) :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 (7), are introduced. The first shape correctly infers the triples in (6) as invalid. The second shape in (7) 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.

```
(7) [rdf:type sh:NodeShape;
      sh:targetSubjectsOf time:hasBeginning;
      sh:sparql[sh:prefixes ... ;
        sh:select """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}."]];

[rdf:type sh:NodeShape;
  sh:targetSubjectsOf time:hasEnd;
  sh:sparql[sh:prefixes ... ;
    sh:select """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}."]];

[rdf:type sh:NodeShape;
  sh:targetSubjectsOf time:hasBeginning;
  sh:sparql[sh:prefixes ... ;
    sh:select """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 5 shows an example of such a pattern.

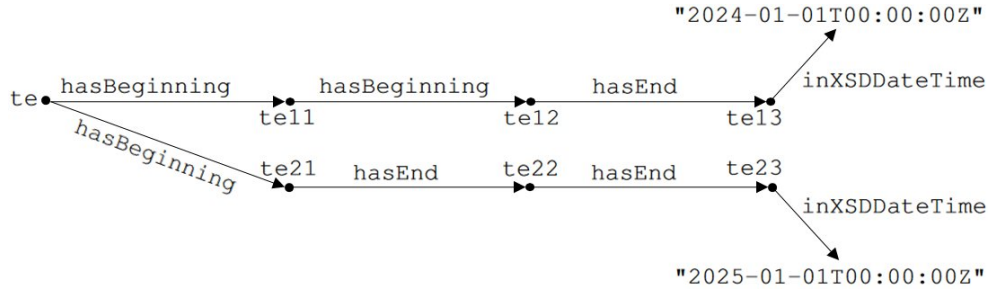


Fig. 5. Example of invalid triples on the properties `hasBeginning`, `hasEnd`, and `inXSDDateTime`.

In Figure 5, 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 5 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 (7) 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 (8). 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 (8) does the same for chains starting with the property `hasEnd`.

```
(8) [rdf:type sh:NodeShape;
    sh:targetSubjectsOf time:hasBeginning;
    sh:rule[rdf:type sh:SPARQLRule; sh:prefixes ... ;
    sh:construct """CONSTRUCT{$this time:hasBeginning ?i}
    WHERE{$this time:hasBeginning/(time:hasBeginning|time:hasEnd)+ ?i}"""].

[rdf:type sh:NodeShape;
    sh:targetSubjectsOf time:hasEnd;
    sh:rule[rdf:type sh:SPARQLRule; sh:prefixes ... ;
    sh:construct """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 (8) and *then* the SHACL shapes in (7), the triples in Figure 5 are properly inferred as invalid: the first rule in (8) infers and adds to the knowledge graph the `hasBeginning` and `hasEnd` properties shown in blue in Figure 6. Then, the first shape in (7) identifies this knowledge graph as invalid thanks to the triples that now *directly* connect `te` with `te13` and `te23`.

Alternatively, rather than introducing new SHACL-SPARQL rules that compute the “transitive closure” of the `hasBeginning` and `hasEnd` properties, as the rules in (8) basically do, we could simply evolve the SHACL shapes in (7) by using the SPARQL property path operators²⁰ fit to identify invalid knowledge graphs such as the one in Figure 5. Specifically, in order to do so, the `WHERE` clauses of the SHACL shapes in (7) could simply match the property paths matched within the `WHERE` clauses of the SHACL-SPARQL rules in (8).

²⁰<https://www.w3.org/TR/sparql11-query/#propertypaths>

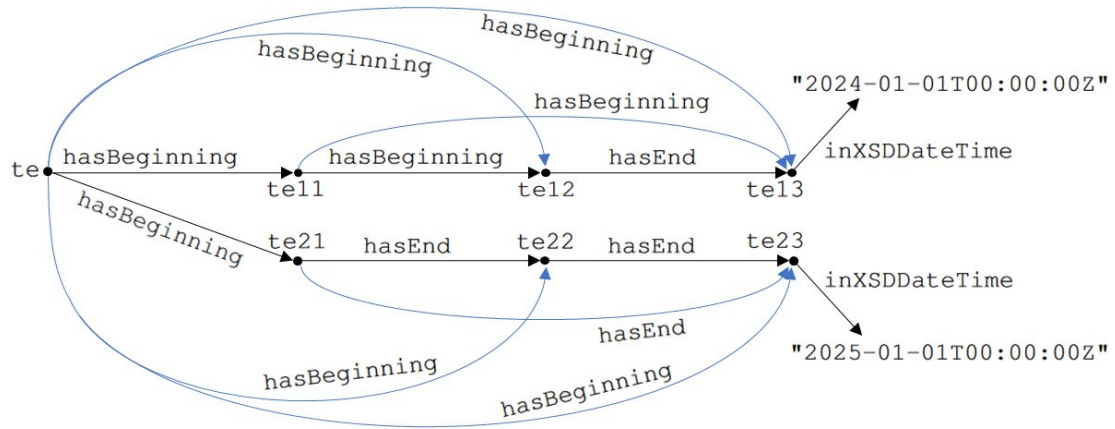


Fig. 6. The knowledge graph inferred from the one in Figure 5 through the rules in (8). The inferred properties are shown in blue.

Although this solution would indeed work for the knowledge graph shown in Figure 5, the main finding of this paper is that it does *not* always work: the SPARQL property path operators are not enough expressive 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 7:

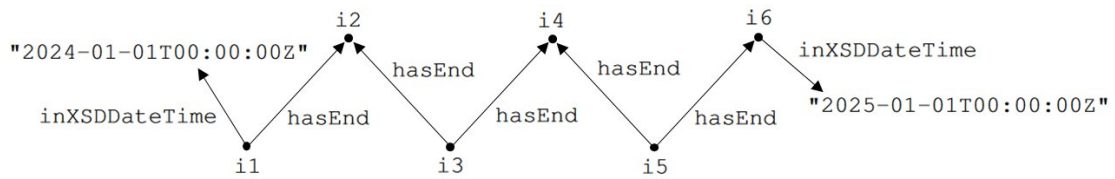


Fig. 7. Example of invalid triples on the properties `hasEnd` and `inXSDDateTime`.

From Figure 7, 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 (7) with the SPARQL property path operators fit to recognize “zigzag” patterns such as the one in Figure 7, 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 7 as invalid, the SHACL-SPARQL rule in (9) 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 (9) basically parallels the ones in (8) but in backward direction and only for instances of the class `Instant`.

```
(9) [rdf:type sh:NodeShape;
     sh:targetClass time:Instant;
     sh:rule[rdf:type sh:SPARQLRule; sh:prefixes ... ;
           sh:construct ""CONSTRUCT{?i time:hasBeginning $this. ?i time:hasEnd $this}
                       WHERE{$this (time:hasBeginning|time:hasEnd)+ ?i}""].
```

In addition, in order to recognize as invalid knowledge graphs such as the one exemplified in Figure 7, the SHACL-SPARQL rule in (9) *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 (8) and (9) on the knowledge graph shown in Figure 7, 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 (7) 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 (8) and (9), until no further triple is inferred, on the knowledge graph in Figure 7, 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 7 are the same*, because `te` cannot begin at two different instants. The rules in (8) and (9) are unable to infer so; therefore, the following rules are added:

```
(10) [rdf:type sh:NodeShape;
      sh:targetSubjectsOf time:hasBeginning;
      sh:rule[rdf:type sh:SPARQLRule; sh:prefixes ... ;
            sh:construct """CONSTRUCT{?i1 time:hasBeginning ?i2}
            WHERE{$this time:hasBeginning ?i1. $this time:hasBeginning ?i2}"""].

[rdf:type sh:NodeShape;
  sh:targetSubjectsOf time:hasBeginning;
  sh:rule[rdf:type sh:SPARQLRule; sh:prefixes ... ;
        sh:construct """CONSTRUCT{?i1 time:hasEnd ?i2}
        WHERE{$this time:hasEnd ?i1. $this time:hasEnd ?i2}"""].
```

We conclude this section with the SHACL-SPARQL rule in (11), which is the dual of the one in (9). The rule in (11) 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.

```
(11) [rdf:type sh:NodeShape;
      sh:targetSubjectsOf time:hasBeginning;
      sh:rule[rdf:type sh:SPARQLRule; sh:prefixes ... ;
            sh:construct """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 for the property `after`.

The first SHACL shape proposed in this section is shown in (12). 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 8. The `xsd:dateTime` values of the two instants must of course comply with the temporal order denoted by the property `before`.

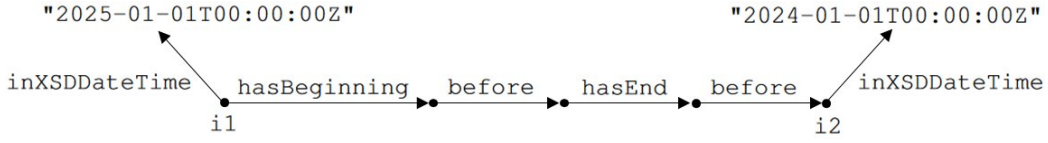


Fig. 8. Invalid knowledge graph involving the properties before, hasBeginning and hasEnd.

```

(12) [rdf:type sh:NodeShape;
      sh:targetSubjectsOf time:before;
      sh:sparql[sh:prefixes ... ;
               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. As partial orders describe direct acyclic graphs, it is necessary to define a SHACL shape that invalidates knowledge graphs containing *cycling chains* of *before* properties (and, again, optionally, *hasBeginning* and *hasEnd*). This SHACL shape is shown in (13):

```

(13) [rdf:type sh:NodeShape;
      sh:targetSubjectsOf time:before;
      sh:sparql[sh:prefixes ... ;
               sh:select """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 (7), namely a shape that invalidates temporal entities that end *before* they begin. This shape is shown in (14); the difference between (14) and the third shape in (7) is that the former *qualitatively* checks that a temporal entity does not end before it begins, while the latter does so *quantitatively*.

```

(14) [rdf:type sh:NodeShape;
      sh:targetSubjectsOf time:before;
      sh:sparql[sh:prefixes ... ;
               sh:select """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 subsection 4.4.2, the three SHACL shapes in (13), (12), and (14) 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 9, 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.

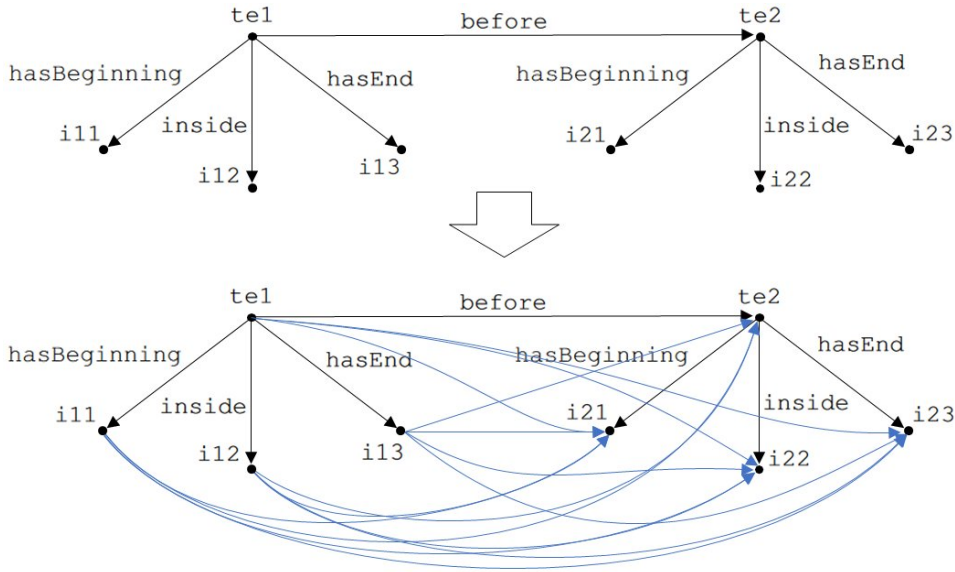


Fig. 9. Inferences on the property *before*; the inferred properties *before* are shown in blue.

Thanks to the SPARQL operators BIND and UNION, the inferences shown in Figure 9 can be carried out via a single SHACL-SPARQL rule, shown in (15). This rule infers that the property *before* holds for the Cartesian product $\{te1, i11, i12, i13\} \times \{te2, i21, i22, i23\}$.

```
(15) [rdf:type sh:NodeShape;
      sh:targetSubjectsOf time:before;
      sh:rule[rdf:type sh:SPARQLRule; sh:prefixes ... ;
            sh:construct """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 10, 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.



Fig. 10. Inferences on the property *before*; the inferred properties are shown in blue.

The inferences depicted in Figure 10 are carried out by the following SHACL-SPARQL rules:

```

1 (16) [rdf:type sh:NodeShape;
2     sh:targetSubjectsOf time:before;
3     sh:rule[rdf:type sh:SPARQLRule; sh:prefixes ... ;
4     sh:construct """CONSTRUCT{?te1 time:before ?te}
5         WHERE{$this time:before ?te. $this ^time:hasEnd ?te1}"""].
6
7     [rdf:type sh:NodeShape;
8     sh:targetSubjectsOf time:before;
9     sh:rule[rdf:type sh:SPARQLRule; sh:prefixes ... ;
10    sh:construct """CONSTRUCT{$this time:before ?te2}
11        WHERE{$this time:before ?te. ?te ^time:hasBeginning ?te2}"""].
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26

```

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 deduced 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 shape should be asserted:

```

27 (17) [rdf:type sh:NodeShape;
28     sh:targetSubjectsOf time:inside;
29     sh:sparql[sh:prefixes ... ;
30     sh:select """SELECT $this WHERE{$this a time:Instant}""";
31     sh:message "Invalid Interval { $this }: the Interval as subject of the
32         property 'inside' is also an Instant. Nevertheless, instants
33         cannot (properly) contain other instants."]].
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51

```

Apart from (17), 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 the neither 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 (18). 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.

```

47 (18) [rdf:type sh:NodeShape;
48     sh:targetSubjectsOf time:inside;
49     sh:rule[rdf:type sh:SPARQLRule; sh:prefixes ... ;
50     sh:construct """CONSTRUCT{?b time:before ?i}
51         WHERE{$this time:hasBeginning ?b. $this time:inside ?i}"""].

```

```

1  [rdf:type sh:NodeShape;
2  sh:targetSubjectsOf time:inside;
3  sh:rule[rdf:type sh:SPARQLRule; sh:prefixes ... ;
4  sh:construct """CONSTRUCT{?i time:before ?e}
5  WHERE{$this time:hasEnd ?e. $this time:inside ?i}"""];
6
7
8
9

```

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 more abstract classes `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 11, 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 (8) and (9) 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 11: (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 (7) invalidate the pattern in (1) while no shape invalidates the one in (2).

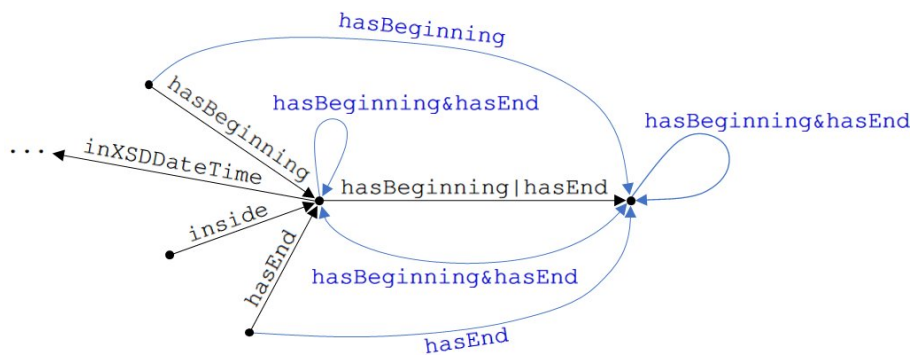


Fig. 11. Inferences on the property `hasBeginning` or `hasEnd`; the inferred properties are shown in blue.

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 `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 already discussed in the next subsections, 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 12 on the left, when both properties are `hasEnd`, as shown in Figure 12 in the middle, and when one of the two properties is `hasBeginning` and the other one is `hasEnd`, as shown in Figure 12 on the right.

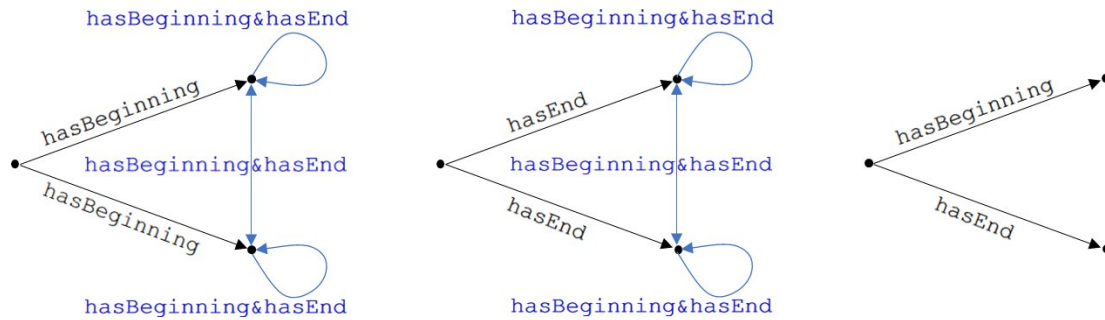


Fig. 12. Inferences on the property `hasBeginning` or `hasEnd`; the inferred properties are shown in blue.

In Figure 12 on the left and in the middle, the SHACL-SPARQL rules in (10) 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 (7) or the one in (14) will report them as invalid.

Finally, in Figure 12 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 (7) and by the shape in (14).

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 (12), 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 (13), which *qualitatively* checks that the knowledge graph does not contain *cycling* chains of properties `before` (and, optionally, `hasBeginning` and `hasEnd`).
- The shape in (14), 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 (12), (13), and (14), 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:

- (19) 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 (19.a), as shown in Figure 13 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 (12), (13), and (14) 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 (15) 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 (12) detects the invalid pattern. Another example is shown in Figure 13 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 (15) 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 (13).

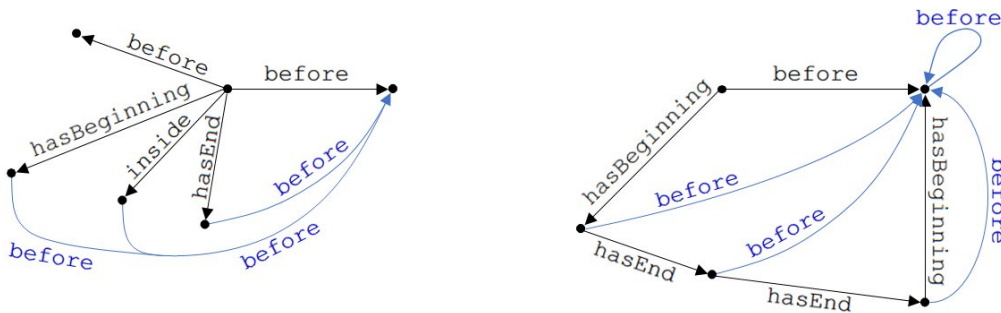


Fig. 13. Sample knowledge graphs involving `before`, `hasBeginning`, `hasEnd`, and `inside`; the properties in blue are inferred.

Let us now consider the case in (19.a). When the property `before` comes after another property `before` or a property `hasBeginning`, as shown in Figure 14 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 (12) and (13) 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 14 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 (12) and (13): if a property `inside` occur 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.

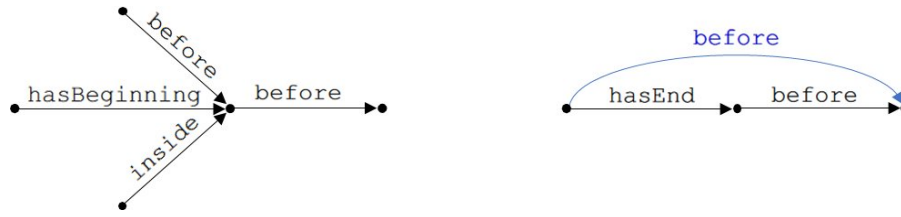


Fig. 14. Sample knowledge graphs involving before, hasBeginning, hasEnd, and inside; the property in blue is inferred.

The case in which the property before comes after the property hasEnd, as shown in Figure 14 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 (16) above infers the property before shown in blue in Figure 14 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 (16), several invalid knowledge graphs would not be invalidated. An example of these invalid knowledge graphs is shown in Figure 15. 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 15, 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 (12), (13), and (14) 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 (15) 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 (12) detects the invalid knowledge graph.

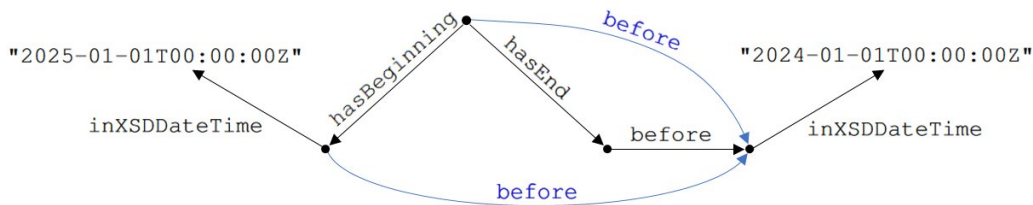


Fig. 15. Invalid knowledge graph involving before, hasBeginning, hasEnd, and inXSDDateTime; the properties in blue are inferred.

The case in (19.c) is the simplest one to validate. As for the case in (19.b) and as shown in Figure 16, if the object of the property before occurs as subject of other properties before, hasBeginning, or hasEnd, the SHACL shapes in (12), (13), and (14) 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 (15) 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.

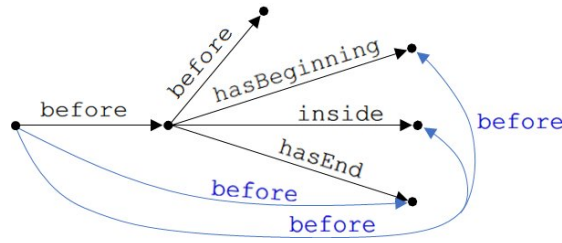


Fig. 16. Invalid knowledge graph involving `before`, `hasBeginning`, `hasEnd`, and `inXSDDateTime`; the properties in blue are inferred.

Finally, we consider the case in (19.d), in which the object of the property `before` also occurs as object of other properties. As shown in Figure 17 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 (16), 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 15, associated with the *first* rule shown above in (16).

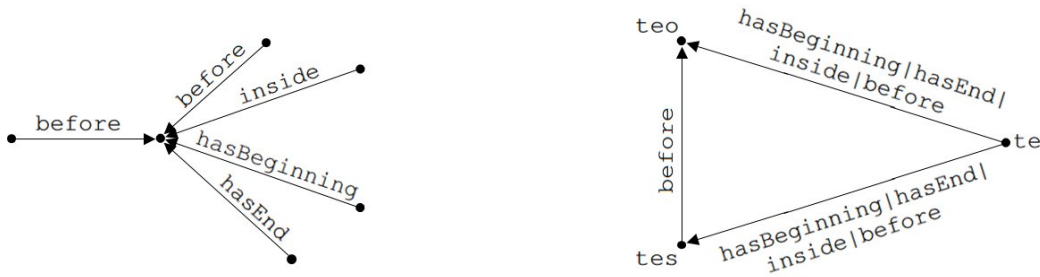


Fig. 17. Knowledge graph involving `before`, `hasBeginning`, `hasEnd`, and `inside`.

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 17, on the right. Five sub-cases need to be distinguished, depending on the specific properties involved:

- (20) 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 (15) infers that also `teo` is connected to `tes` through the property `before` and so the shape in (13) 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 (10) infer that the subject and the object of `before` are the same instant, the rule in (15) infers that `tes` or `teo` occur before themselves, and the shape in (13) 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 patterns 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 (14) 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 (18) infer the opposite properties `before`, which denote the right temporal order, so that the shape in (13) detects the cycling paths and invalidates the knowledge graph.

4.4.3. `inside`

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

The SHACL shape shown above in (17) 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 (17) 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 (17), 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 (18). As shown in Figure 18, 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 (12) *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 (20.e), the shape in (13) *qualitatively* invalidates knowledge graphs that, after new inferred properties `before` have been added therein, contain a cycling path.



Fig. 18. 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 section introduce SHACL shapes and SHACL-SPARQL rules for validating the Time Ontology's properties that denote Allen's temporal relations. In particular, this section will investigate how these properties interplay with the properties `hasBeginning`, `hasEnd`, and `before`, seen in the previous section, while the next

section will propose SHACL-SPARQL rules that implement Allen’s interval algebra (see Table 2 above) as well as SHACL shapes to invalidate knowledge graphs that do not comply with this algebra.

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 (21), which specifies that every instant beginning a proper interval occurs before the instant that ends it:

```
(21) [rdf:type sh:NodeShape;
      sh:targetClass time:ProperInterval;
      sh:rule[rdf:type sh:SPARQLRule; sh:prefixes ... ;
            sh:construct """CONSTRUCT{?b time:before ?e}
                          WHERE{$this time:hasBeginning ?b. $this time:hasEnd ?e}"""].
```

The next subsections will discuss the SHACL shapes and the SHACL-SPARQL rules to validate the fifteen properties denoting Allen’s temporal relations, shown in Figure 1 within the box associated with `ProperInterval`. In particular, in line with what has been done above for the properties `after` and `inside`, we mostly introduce SHACL-SPARQL rules that infer properties `hasBeginning`, `hasEnd`, and `before` between the two involved proper intervals. After these rules are iteratively re-executed until no further triple is inferred, the SHACL shapes defined in the previous section will validate the (inferred) knowledge graphs.

However, as it will be explained below, for two properties denoting Allen’s temporal relations it is also necessary to introduce some SHACL shapes: `intervalIn` and `intervalDisjoint`. We remind that these two properties are the only two properties that do not denote 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:

```
(22) [rdf:type sh:NodeShape;
      sh:targetSubjectsOf time:before;
      sh:rule[rdf:type sh:SPARQLRule; sh:prefixes ... ;
            sh:construct """CONSTRUCT{$this time:intervalBefore ?pi2}
                          WHERE{$this time:before ?pi2; rdf:type time:ProperInterval.
                                ?pi2 rdf:type 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 (23): 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.

```

1 (23) [rdf:type sh:NodeShape;
2     sh:targetSubjectsOf time:intervalStarts;
3     sh:rule[rdf:type sh:SPARQLRule; sh:prefixes ... ;
4     sh:construct """CONSTRUCT{?b1 time:hasBeginning ?b2. ?b2 time:hasBeginning ?b1.
5         ?b1 time:hasEnd ?b2. ?b2 time:hasEnd ?b1}
6     WHERE{$this time:intervalStarts ?te2.
7         $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 (24) implements this inference. Then, the rule shown above in (15) will in turn infer that also any instant within that interval occurs before the end of `intervalStarts`'s object.

```

13 (24) [rdf:type sh:NodeShape;
14     sh:targetSubjectsOf time:intervalStarts;
15     sh:rule[rdf:type sh:SPARQLRule; sh:prefixes ... ;
16     sh:construct """CONSTRUCT{$this time:before ?e2}
17     WHERE{$this time:intervalStarts/time:hasEnd ?e2}"""].

```

Finally, a further pattern involving the property `intervalStart` must be considered. Suppose, as depicted in Figure 19, that the object of `intervalStarts` also occurs as subject of the property `hasBeginning`, while its subject does not. The SHACL-SPARQL rule in (23) 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 (25):

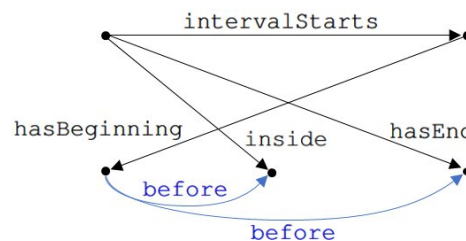


Fig. 19. Inference on the properties `intervalStarts`, `hasBeginning`, `inside`, and `hasEnd`. The inferred properties are shown in blue.

```

36 (25) [rdf:type sh:NodeShape;
37     sh:targetSubjectsOf time:intervalStarts;
38     sh:rule[rdf:type sh:SPARQLRule; sh:prefixes ... ;
39     sh:construct """CONSTRUCT{?b1 time:before ?i2}
40     WHERE{$this time:intervalStarts/(time:hasEnd|time:inside) ?i2.
41     $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 (26). In light of the same observations made above for the rule in (24), note that the first of the rules in (26) 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.

```
(26) [rdf:type sh:NodeShape;
      sh:targetSubjectsOf time:intervalOverlaps;
      sh:rule[rdf:type sh:SPARQLRule; sh:prefixes ... ;
            sh:construct """CONSTRUCT{?b1 time:before ?te2}
                        WHERE{$this time:intervalOverlaps ?te2.
                          $this time:hasBeginning ?b1}"""].
```

```
[rdf:type sh:NodeShape;
  sh:targetSubjectsOf time:intervalOverlaps;
  sh:rule[rdf:type sh:SPARQLRule; sh:prefixes ... ;
        sh:construct """CONSTRUCT{?b2 time:before ?e1}
                    WHERE{$this time:intervalOverlaps/time:hasBeginning ?b2.
                      $this time:hasEnd ?e1}"""].
```

```
[rdf:type sh:NodeShape;
  sh:targetSubjectsOf time:intervalOverlaps;
  sh:rule[rdf:type sh:SPARQLRule; sh:prefixes ... ;
        sh:construct """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:

```
(27) [rdf:type sh:NodeShape;
      sh:targetSubjectsOf time:intervalMeets;
      sh:rule[rdf:type sh:SPARQLRule; sh:prefixes ... ;
            sh:construct """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 20, 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 (28).

```
(28) [rdf:type sh:NodeShape;
      sh:targetSubjectsOf time:intervalMeets;
      sh:rule[rdf:type sh:SPARQLRule; sh:prefixes ... ;
            sh:construct """CONSTRUCT{?i time:before ?te2}
                        WHERE{$this time:intervalMeets ?te2.
                          $this time:hasBeginning|time:inside ?i}"""].
```

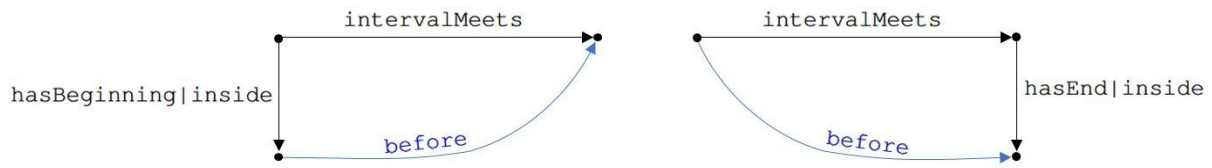


Fig. 20. Inference on the properties `intervalMeets`, `hasBeginning`, `inside`, and `hasEnd`. The inferred properties are shown in blue.

```
[rdf:type sh:NodeShape;
 sh:targetSubjectsOf time:intervalMeets;
 sh:rule[rdf:type sh:SPARQLRule; sh:prefixes ... ;
 sh:construct """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 (23), (24), and (25) above.

```
(29) [rdf:type sh:NodeShape;
 sh:targetSubjectsOf time:intervalFinishes;
 sh:rule[rdf:type sh:SPARQLRule; sh:prefixes ... ;
 sh:construct """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}"""].

[rdf:type sh:NodeShape;
 sh:targetSubjectsOf time:intervalFinishes;
 sh:rule[rdf:type sh:SPARQLRule; sh:prefixes ... ;
 sh:construct """CONSTRUCT{?b2 time:before $this}
 WHERE{$this time:intervalFinishes/time:hasBeginning ?b2}"""].

[rdf:type sh:NodeShape;
 sh:targetSubjectsOf time:intervalFinishes;
 sh:rule[rdf:type sh:SPARQLRule; sh:prefixes ... ;
 sh:construct """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 (24), (26), and (28).

```
(30) [rdf:type sh:NodeShape;
      sh:targetSubjectsOf time:intervalDuring;
      sh:rule[rdf:type sh:SPARQLRule; sh:prefixes ... ;
            sh:construct """CONSTRUCT{?b2 time:before $this}
            WHERE{$this time:intervalDuring/time:hasBeginning ?b2}"""];

[rdf:type sh:NodeShape;
  sh:targetSubjectsOf time:intervalDuring;
  sh:rule[rdf:type sh:SPARQLRule; sh:prefixes ... ;
        sh:construct """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:

```
(31) [rdf:type sh:NodeShape;
      sh:targetSubjectsOf time:intervalEquals;
      sh:rule[rdf:type sh:SPARQLRule; sh:prefixes ... ;
            sh:construct """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}"""];

[rdf:type sh:NodeShape;
  sh:targetSubjectsOf time:intervalEquals;
  sh:rule[rdf:type sh:SPARQLRule; sh:prefixes ... ;
        sh:construct """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:

```
(32) [rdf:type sh:NodeShape;
      sh:targetSubjectsOf time:intervalEquals;
      sh:rule[rdf:type sh:SPARQLRule; sh:prefixes ... ;
            sh:construct """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 (33), 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`.

```
(33) [rdf:type sh:NodeShape;
      sh:targetSubjectsOf time:intervalIn;
      sh:sparql[sh:prefixes ... ;
        sh:select """SELECT $this ?pi
          WHERE{$this time:intervalIn ?pi.
            $this time:hasBeginning ?b1. ?pi time:hasBeginning ?b2.
            {?b1 time:inXSDDateTime ?dt1. ?b2 time:inXSDDateTime ?dt2.
              FILTER(?dt1<?dt2)}UNION{?b1 time:before ?b2}""";
        sh:message "Invalid triple `{$this} time:intervalIn {?pi}`:
          {$this} begins before {?pi}."]];

[rdf:type sh:NodeShape;
  sh:targetSubjectsOf time:intervalIn;
  sh:sparql[sh:prefixes ... ;
    sh:select """SELECT $this ?pi
      WHERE{$this time:intervalIn ?pi.
        $this time:hasEnd ?e1. ?pi time:hasEnd ?e2.
        {?e1 time:inXSDDateTime ?dt1. ?e2 time:inXSDDateTime ?dt2.
          FILTER(?dt1>?dt2)}UNION{?e2 time:before ?e1}""";
    sh:message "Invalid triple `{$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 (34).

```
(34) [rdf:type sh:NodeShape;
      sh:targetSubjectsOf time:intervalIn;
      sh:rule[rdf:type sh:SPARQLRule; sh:prefixes ... ;
        sh:construct """CONSTRUCT{$this time:intervalStarts ?te2}
          WHERE{$this time:intervalIn ?te2.
            $this time:hasBeginning ?b1. ?te2 time:hasBeginning ?b2.
            {?b1 time:inXSDDateTime ?dt1. ?b2 time:inXSDDateTime ?dt2.
              FILTER(?dt2=?dt1)}UNION{?b1 time:hasBeginning ?b2}""";
```

```

1
2 [rdf:type sh:NodeShape;
3   sh:targetSubjectsOf time:intervalIn;
4   sh:rule[rdf:type sh:SPARQLRule; sh:prefixes ... ;
5     sh:construct ""CONSTRUCT{$this time:intervalFinishes ?te2}
6     WHERE{$this time:intervalIn ?te2.
7       $this time:hasEnd ?e1. ?te2 time:hasEnd ?e2.
8       {?e1 time:inXSDDateTime ?dt1. ?e2 time:inXSDDateTime ?dt2.
9       FILTER(?dt1=?dt2)}UNION{?e1 time:hasBeginning ?e2}}""].
10
11 [rdf:type sh:NodeShape;
12   sh:targetSubjectsOf time:intervalIn;
13   sh:rule[rdf:type sh:SPARQLRule; sh:prefixes ... ;
14     sh:construct ""CONSTRUCT{$this time:intervalDuring ?te2}
15     WHERE{$this time:intervalIn ?te2.
16       $this time:hasBeginning ?b1. ?te2 time:hasBeginning ?b2.
17       $this time:hasEnd ?e1. ?te2 time:hasEnd ?e2.
18       {?b1 time:inXSDDateTime ?dtb1. ?b2 time:inXSDDateTime ?dtb2.
19       FILTER(?dtb2<?dtb1)}UNION{?b2 time:before ?b1}
20       {?e1 time:inXSDDateTime ?dte1. ?e2 time:inXSDDateTime ?dte2.
21       FILTER(?dte1<?dte2)}UNION{?e2 time:before ?e1}}""].

```

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 (35), 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.

```

32 (35) [rdf:type sh:NodeShape;
33   sh:targetSubjectsOf time:intervalDisjoint;
34   sh:sparql[sh:prefixes ... ;
35     sh:select ""SELECT $this ?pi ?i
36     WHERE{$this time:intervalDisjoint ?pi.
37       $this (time:hasBeginning|time:hasEnd|time:inside)+ ?i.
38       ?pi (time:hasBeginning|time:hasEnd|time:inside)+ ?i}"";
39     sh:message "Invalid triple `${$this} time:intervalDisjoint {?pi}`:
40       the instant {?i} belongs to both {$this} and {?pi}."].
41
42 [rdf:type sh:NodeShape;
43   sh:targetSubjectsOf time:intervalDisjoint;
44   sh:sparql[sh:prefixes ... ;
45     sh:select ""SELECT $this ?pi ?i1 ?i2
46     WHERE{$this time:intervalDisjoint ?pi.
47       $this (time:hasBeginning|time:hasEnd|time:inside)+ ?i1.
48       ?pi (time:hasBeginning|time:hasEnd|time:inside)+ ?i2.
49       ?i1 time:inXSDDateTime ?dt1. ?i2 time:inXSDDateTime ?dt2.
50       FILTER(?dt1=?dt2)}"";
51     sh:message "Invalid triple `${$this} time:intervalDisjoint {?pi}`:
52       the instants {?i1} and {?i2}, each of which belongs to one
53       of the two intervals, denote the same xsd:dateTime value."].

```

Then, we also introduce the SHACL-SPARQL rules in (36) 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.

```
(36) [rdf:type sh:NodeShape;
      sh:targetSubjectsOf time:intervalDisjoint;
      sh:rule[rdf:type sh:SPARQLRule; sh:prefixes ... ;
            sh:construct """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:inXSDDateTime ?dt1. ?i2 time:inXSDDateTime ?dt2.
                  FILTER(?dt1<?dt2)}UNION{?i1 time:before ?i2}}"""];

[rdf:type sh:NodeShape;
  sh:targetSubjectsOf time:intervalDisjoint;
  sh:rule[rdf:type sh:SPARQLRule; sh:prefixes ... ;
        sh:construct """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.
              {?i1 time:inXSDDateTime ?dt1. ?i2 time:inXSDDateTime ?dt2.
              FILTER(?dt1>?dt2)}UNION{?i2 time:before ?i1}}"""];
```

5.10. Evaluation

In order to evaluate whether the SHACL-SPARQL rules proposed in the previous subsections, as well as the four shapes proposed for `intervalIn` and `intervalDisjoint`, cover all knowledge graphs that we may build through the properties denoting Allen’s temporal relations, we must consider how the involved intervals relate of one another when they are connected through one of these properties. This will (visually) make evident which `before` relations must be inferred and, consequently, which SHACL-SPARQL rules must be asserted.

For example, Figure 21 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 (26) are then respectively asserted to infer each of the three occurrences of `before` in Figure 21. As already explained above in (26), 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.

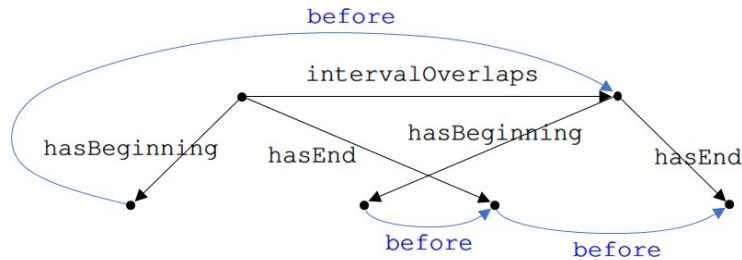


Fig. 21. Inferences on the property `intervalOverlaps`.

Another example is shown in Figure 22, which depicts two proper intervals connected through the property `intervalFinishes`. Again, it is easy to double-check that the SHACL-SPARQL rules in (29) above infer the occurrences of `before` shown in the figure.

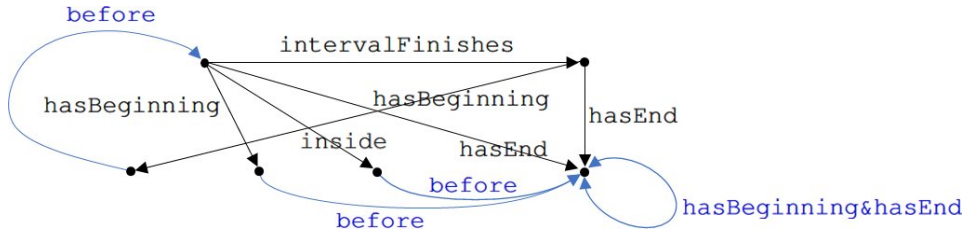


Fig. 22. Inferences on the property `intervalFinishes`. The inferred properties are shown in blue.

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 23. The two SHACL shapes in (33) 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 23).

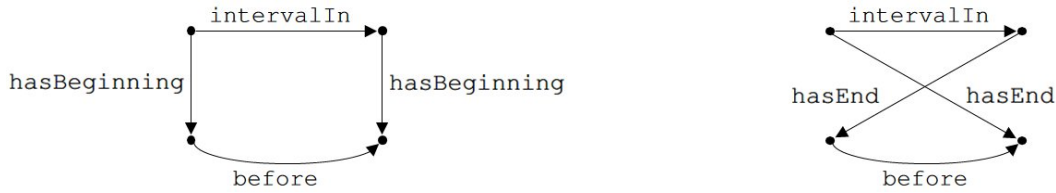


Fig. 23. Invalid patterns involving the property `intervalIn`. The subject of the property cannot begin before or end after its object.

Similarly, it must be checked that two intervals connected through the property `intervalDisjoint` do not share any instant, as depicted in Figure 24. The two SHACL shapes in (35) do invalidate patterns as such, both when there are two different instants associated with the same `xsd:dateTime` value (as in Figure 24) and when there is a single instant connected to both intervals through the properties `hasBeginning`, `hasEnd`, or `inside`.

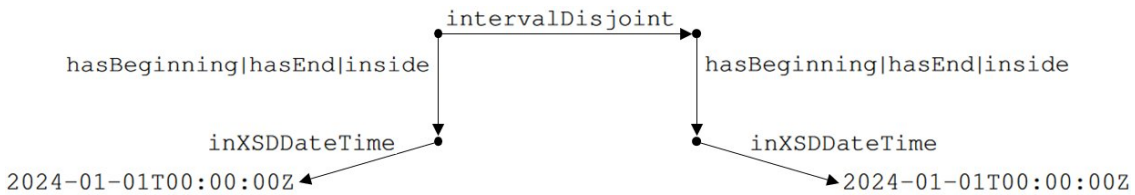


Fig. 24. Invalid patterns involving the property `intervalDisjoint`. The two involved intervals cannot share instants.

Furthermore, for knowledge graphs that comply with the shapes in (33) and (35), 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 25 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 (34) 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 (24) and (29) 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.

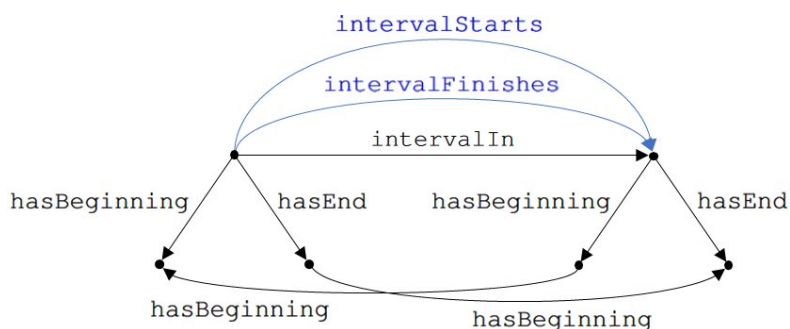


Fig. 25. Invalid patterns involving the property `intervalIn`. The two object properties in blue are inferred.

Similar considerations hold for the property `intervalDisjoint`, as exemplified in Figure 26. In the knowledge graph shown in the figure, `intervalDisjoint`'s object begins before the beginning of its subject; therefore, the SHACL-SPARQL rules in (36) 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 (36) 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 (13) above detects as invalid.

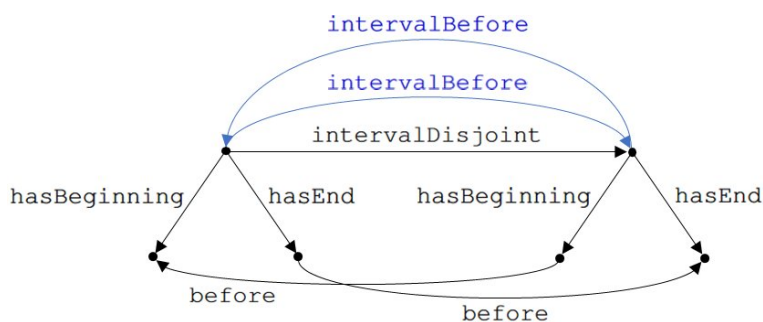


Fig. 26. Invalid patterns involving the property `intervalDisjoint`. The two object properties in blue are inferred.

6. Adding SHACL shapes and SHACL-SPARQL rules to the Time Ontology: disjoint, symmetric, and reflexive 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:

- (37) 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 (37.a-c) through OWL axioms that involve the properties `owl:propertyDisjointWith`, `owl:AsymmetricProperty`, and `owl:IrreflexiveProperty`.

In our formalization, which is based on SHACL rather than OWL, we introduce special SHACL shapes and SHACL-SPARQL rules to encode (37.a-c).

Indeed, it would be rather easy to encode SHACL shape that directly parallel the advocated OWL axioms involving `owl:propertyDisjointWith`, `owl:AsymmetricProperty`, and `owl:IrreflexiveProperty`. However, this paper proposes an alternative solution that, besides encoding (37.a-b), enforces Allen's interval algebra, as it will be explained in the next section.

The proposed solution makes use of RDF reification²¹ in order to compute the set of Allen's temporal relations that can *possibly* connect two proper intervals. Then, in cases where an explicitly asserted property is *not* one of these possible properties, the knowledge graph is reported as invalid.

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 27 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`.

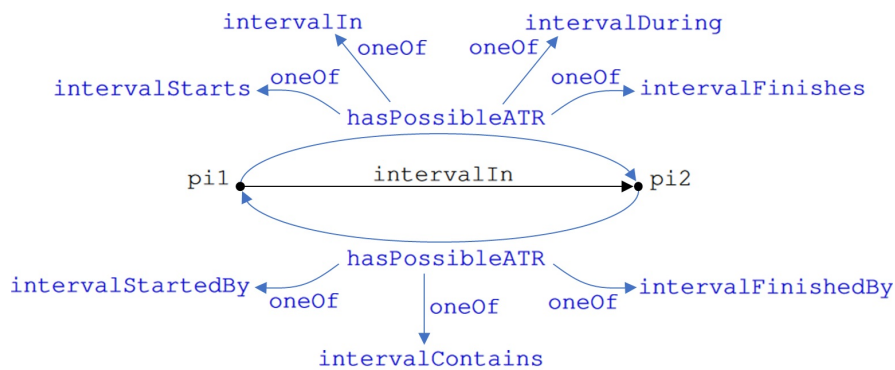


Fig. 27. Inferring *possible* properties among proper intervals. The properties in blue are those that must be inferred from the one in black.

In order to infer the properties `hasPossibleATR` and `oneOf` as explained in Figure 27, the SHACL-SPARQL rules in (38) and (39) are introduced.

²¹<https://www.w3.org/TR/rdf-primer/#reification>

```

1 (38) [rdf:type sh:NodeShape;                                1
2   sh:targetClass time:ProperInterval;                      2
3   sh:rule[rdf:type sh:SPARQLRule; sh:prefixes ... ;       3
4     sh:construct ""CONSTRUCT{                             4
5       [rdf:type rdf:Statement,?cp; rdf:subject $this; rdf:object ?pi2; 5
6         rdf:predicate :hasPossibleATR; :includes $this,?pi2; :source ?atr]. 6
7       [rdf:type rdf:Statement,?cp; rdf:subject ?pi2; rdf:object $this; 7
8         rdf:predicate :hasPossibleATR; :includes $this,?pi2; :source ?atr]] 8
9     WHERE{$this ?atr ?pi2.                                  9
10       FILTER(strStarts(str(?atr),"http://www.w3.org/2006/time#interval")) 10
11       BIND(IF($this=?pi2, :ContainsCyclingPath, rdf:Statement) AS ?cp) 11
12       NOT EXISTS{?r rdf:type rdf:Statement; rdf:subject $this; 12
13         rdf:object ?pi2; rdf:predicate :hasPossibleATR; :source ?atr}}""]]. 13
14
15 (39) [rdf:type sh:NodeShape;                                15
16   sh:targetClass rdf:Statement                              16
17   sh:rule[rdf:type sh:SPARQLRule; sh:prefixes ... ;       17
18     sh:construct ""CONSTRUCT{$this :oneOf time:intervalIn,time:intervalStarts, 18
19       time:intervalFinishes,time:intervalDuring. ?thisi :oneOf 19
20       time:intervalContains,time:intervalStartedBy,time:intervalFinishedBy} 20
21     WHERE{$this rdf:subject ?pi1; rdf:object ?pi2; rdf:predicate :hasPossibleATR; 21
22       :source time:intervalIn. ?thisi rdf:subject ?pi2; rdf:object ?pi1; 22
23       rdf:predicate :hasPossibleATR; :source time:intervalIn}""]]. 23

```

The SHACL-SPARQL rule in (38) 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 (38); since the rules are re-executed until no new triple is inferred, this clause prevents infinite loops.

Note also that the rule in (38) 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 27. 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 (39).

`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 (38)) 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 (38), 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 (38).

`includes` and `ContainsCyclingPath` are not relevant for the content of the present section, but they will be crucial for the content of the the next one, specifically for computing Allen’s interval algebra.

Once the initial reifications are created through the rule in (38), 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. (39) 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 (38) and (39) infer the properties shown in blue in Figure 27.

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 (40), which invalidates any property that denotes one of Allen's temporal relation and that does not comply with this constraint.

```
(40) [rdf:type sh:NodeShape;
      sh:targetClass rdf:Statement;
      sh:sparql [sh:prefixes ... ;
                sh:select """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 (40): the two ones shown in Figure 28. 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 (37.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 (37.b) above.

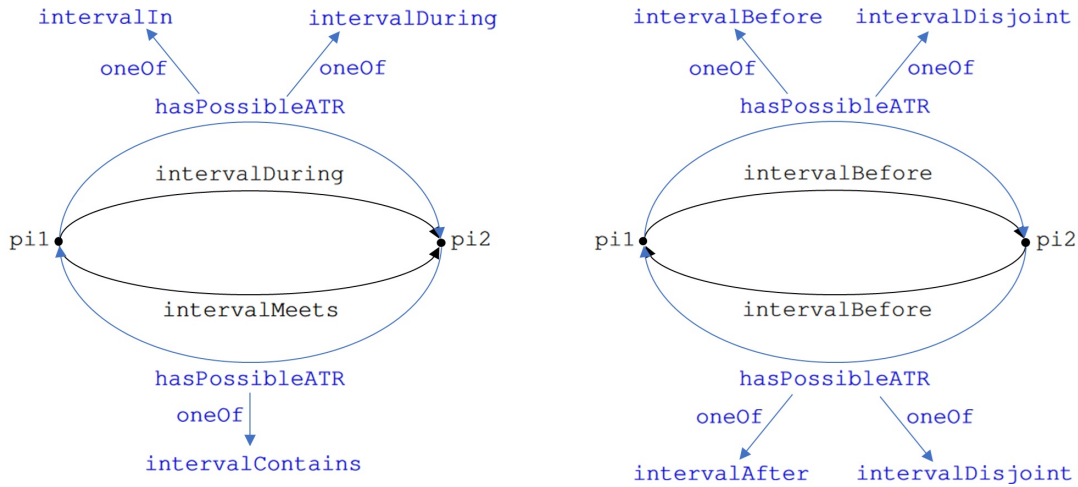


Fig. 28. Inferring *possible* properties among proper intervals. The properties in blue inferred.

The SHACL-SPARQL rule in (38) 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 28, 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 (41) and (42), which respectively concern the properties *intervalDuring* and *intervalBefore* and parallel the rule in (39), 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 (39), (41), and (42).

```

1 (41) [rdf:type sh:NodeShape;
2     sh:targetClass rdf:Statement
3     sh:rule[rdf:type sh:SPARQLRule; sh:prefixes ... ; sh:construct"""
4     CONSTRUCT{$this :oneOf time:intervalDuring,time:intervalIn.
5         ?thisi :oneOf time:intervalContains}
6     WHERE{$this rdf:subject ?pi1; rdf:object ?pi2; rdf:predicate :hasPossibleATR;
7         :source time:intervalDuring. ?thisi rdf:subject ?pi2; rdf:object ?pi1;
8         rdf:predicate :hasPossibleATR; :sourcei time:intervalDuring}"""].

```

```

10 (42) [rdf:type sh:NodeShape;
11     sh:targetClass rdf:Statement
12     sh:rule[rdf:type sh:SPARQLRule; sh:prefixes ... ; sh:construct"""
13     CONSTRUCT{$this :oneOf time:intervalBefore,time:intervalDisjoint.
14         ?thisi :oneOf time:intervalAfter, time:intervalDisjoint}
15     WHERE{$this rdf:subject ?pi1; rdf:object ?pi2; rdf:predicate :hasPossibleATR;
16         :source time:intervalBefore. ?thisi rdf:subject ?pi2; rdf:object ?pi1;
17         rdf:predicate :hasPossibleATR; :sourcei time:intervalBefore}"""].

```

The two rules in (41) and (42) add the `oneOf` properties shown in blue in Figure 28 and, finally, the SHACL shape in (40) 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 (43), 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 (37.c) above, all properties denoting Allen's temporal relations but `intervalEquals` are irreflexive. The SHACL-SPARQL rule in (41) 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 (43) checks that this property was `intervalEquals`.

It is easy to verify that the SHACL shape in (43) properly validates all fifteen properties when they connect a proper interval with itself. The next section will show that the shape also validates *any* arbitrary path of properties denoting Allen's temporal relations that connects a proper interval with itself.

```

33 (43) [rdf:type sh:NodeShape;
34     sh:targetClass rdf:Statement;
35     sh:sparql [sh:prefixes ... ;
36     sh:select """SELECT $this ?pi
37     WHERE{$this rdf:subject ?pi; rdf:predicate :hasPossibleATR; rdf:object ?pi.
38     NOT EXISTS{$this :oneOf time:intervalEquals}""";
39     sh:message "Invalid ProperInterval {?pi}: it is connected to itself
40     through a cycling path of Allen's temporal relations but these
41     do not allow for time:intervalEquals between {?pi} and itself."].

```

6.1. Evaluation

Contrary to the case-based evaluation presented in the two previous sections, for the SHACL shape and the SHACL-SPARQL rules introduced in this section we chose to carry out a simpler generate & test evaluation: we implemented a small Java script 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 $15!/((15-13)!) = 105$ 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 28 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:

- 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: Allen's interval algebra

The present section extends the scheme introduced in the previous one fit to encompass Allen's interval algebra. Figure 29 shows an example of invalid knowledge graph that can be identified as such only through Allen's interval algebra: by applying the composition table reported above in Table 2, it may be deduced that `pi2` cannot be connected to `pi1` through the property `intervalContains`.

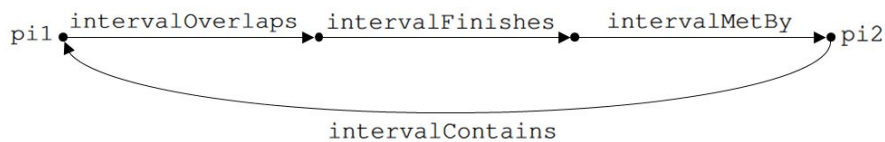


Fig. 29. Invalid knowledge graph involving properties denoting Allen's temporal relations.

This section introduces SHACL-SPARQL rules that implement Table 2. When these rules are applied to the sequence of three properties connecting `pi1` to `pi2` in Figure 29, they infer that `intervalContains` does *not* belong to the set of possible properties denoting one of Allen's temporal relations that may connect `pi2` to `pi1`. Then, the SHACL shape shown in the previous section in (43) invalidates the the property.

The first SHACL-SPARQL rule that we introduce to implement Allen's interval algebra is shown in (44). This rule creates the reification that denotes a path of properties denoting Allen's temporal relations out of the reifications denoting two *contiguous* paths as such. The latter are then connected to the former through the properties `source1`

and `source2`. In order to prevent infinite loops, the rule triggers only if the reification that it creates does not already exist, namely only if the knowledge graph does not already including a reification having the same sources.

Furthermore, the rule does not use as sources reifications that contain cycling paths; the reason is that the SHACL shape in (43) already invalidates proper intervals connected to themselves through a path of Allen's temporal properties that do not allow for the property `intervalEquals`; conversely, in cases two intervals are equal and this is indeed validated by the SHACL shape in (43), there is no need to check paths that include cycles among this interval, but only the parts of those paths that do not include the cycles. Finally, the path denoted by the reification created by this rule might also contain a cycle; this may be detected by checking that the two reifications denoted by `source1` and `source2` do not include a common proper interval different from `?pi2`, i.e., the proper interval that conjoins them. If that is the case, the reification created in the `CONSTRUCT` clause will be again asserted as instance of the special class `ContainsCyclingPath`.

```
(44) [rdf:type sh:NodeShape;
      sh:targetClass rdf:Statement
      sh:rule[rdf:type sh:SPARQLRule; sh:order 1; sh:prefixes ...; sh:construct """
        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 (44) 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 (38) 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 (44) 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 (44); conversely, if the latter is executed before the proper intervals included within the source reifications have been computed, the rule in (44) 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 (45). Note that the latter specifies “`sh:order 0`” while the rule in (44) specifies “`sh:order 1`”; `sh:order` is a special SHACL property to specify the execution order of the rules²². The specified values for the property `sh:order` guarantee that the rule in (45) is always executed before the one in (44).

```
(45) [rdf:type sh:NodeShape;
      sh:targetClass rdf:Statement
      sh:rule[rdf:type sh:SPARQLRule; sh:order 0; sh:prefixes ...;
      sh:construct """CONSTRUCT{$this :includes ?pi}
      WHERE{($this :source1 ?s)UNION{$this :source2 ?s}. ?s :includes ?pi}"""].
```

From the knowledge graph in Figure 29, the SHACL-SPARQL rules in (44) and (45), together with the rules shown in the previous section, infer the reifications and the properties shown in blue in Figure 30.

²²<https://www.w3.org/TR/shacl-af/#rules-order>

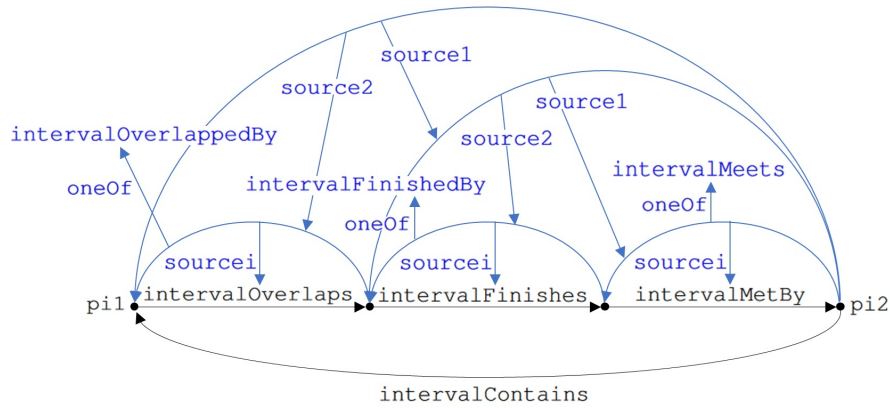


Fig. 30. Invalid knowledge graph involving properties denoting Allen's temporal relations.

In order to enhance readability, Figure 30 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 (44) to loop infinitely.

The three initial reifications in opposite directions are created by the rule in (38) out of the three asserted properties `intervalOverlaps`, `intervalFinishes`, and `intervalMetBy`; the latter are connected to these three reifications through the property `source1`. Of course, the rule in (38) 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 (39), (41), and (42) 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 (44) first creates the reification from `pi2` to the subject of `intervalFinishes` by conjoining the two initial reifications on the right; then, it creates the reification from `pi2` to `pi1` by conjoining the latter with the third initial reification. The rule in (44) 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`. It is easy to see that the rule in (44) is re-applied until it generates reifications that describe cycling paths connecting proper intervals with themselves; being *cycling* paths, the latter are no longer reused by the rule in (44) to produce new reifications, so that its iterative re-execution terminates.

Now we need SHACL-SPARQL rules that populate the reifications of the conjoined paths with the set of possible properties denoting Allen's temporal relations through the property `oneOf`, on the basis of the properties connected through `oneOf` to the two reifications in its `source1` and `source2`. These are the rules that implement the composition table in Table 2. The paper only reports the composition rules needed for the example in Figure 30; all rules implementing Table 2 are however available on the GitHub repository.

The first rule that we need for the example in Figure 30 is the one that composes `intervalMeets` with `intervalFinishedBy`. This rule, shown in (46), corresponds to the cell at the row "Meets (m)" and the column "fi" in Table 2; that cell specifies that by composing `intervalMeets` with `intervalFinishedBy` the only possible Allen's temporal relation is `intervalBefore`. Note that (46) also asserts `intervalDisjoint`, super-property of `intervalBefore`, among the possible properties on the conjoined path.

```
(46) [rdf:type sh:NodeShape;
      sh:targetSubjectsOf :source1;
      sh:rule[rdf:type sh:SPARQLRule; sh:prefixes ... ; sh:construct """
        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 30, shown in (47), combines `intervalBefore` with `intervalOverlappedBy`; as specified in the corresponding cell of Table 2, 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 (47) adds `intervalDisjoint` and `intervalIn`, super-properties of `intervalBefore`, `intervalDuring`, and `intervalStarts`.

```
(47) [rdf:type sh:NodeShape;
      sh:targetSubjectsOf :source1;
      sh:rule[rdf:type sh:SPARQLRule; sh:prefixes ... ; sh:construct """
        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 (46) and (47) to the knowledge graph in Figure 30, the one in Figure 31 is obtained. Figure 31 omits `source1`, `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 property `intervalContains` is invalidated by the SHACL shape in (40).

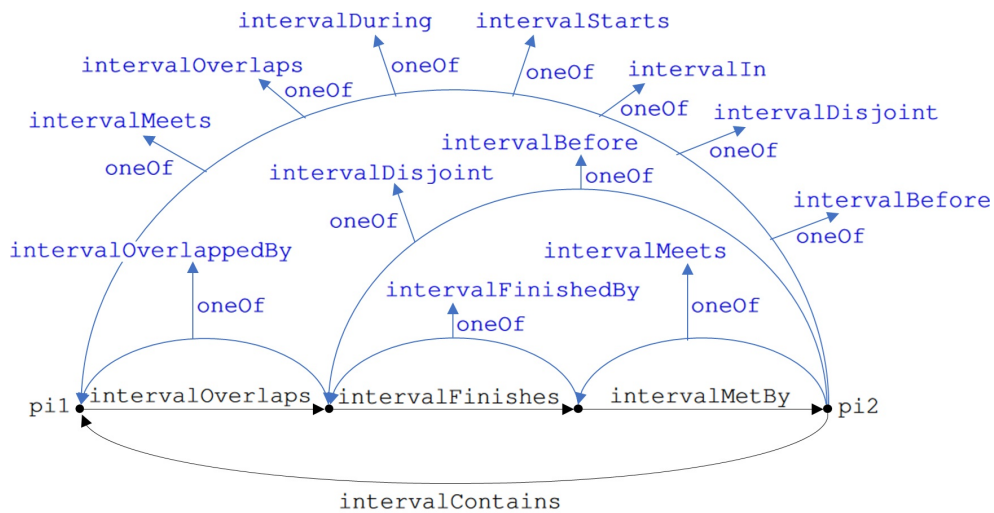


Fig. 31. Properties inferred by the SHACL-SPARQL rules in (46) and (47) from the knowledge graph in Figure 30.

As a second example, consider the knowledge graph in Figure 32, 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 (43). This is indeed the case, as shown in the figure with respect to the proper interval `pi1`: according to Table 2, 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 likewise derived between each of the three proper intervals and itself through all other possible cycling paths of reified `hasPossibleATR` properties that can be built from the asserted properties `intervalOverlappedBy`, `intervalStartedBy`, and `intervalDisjoint`.

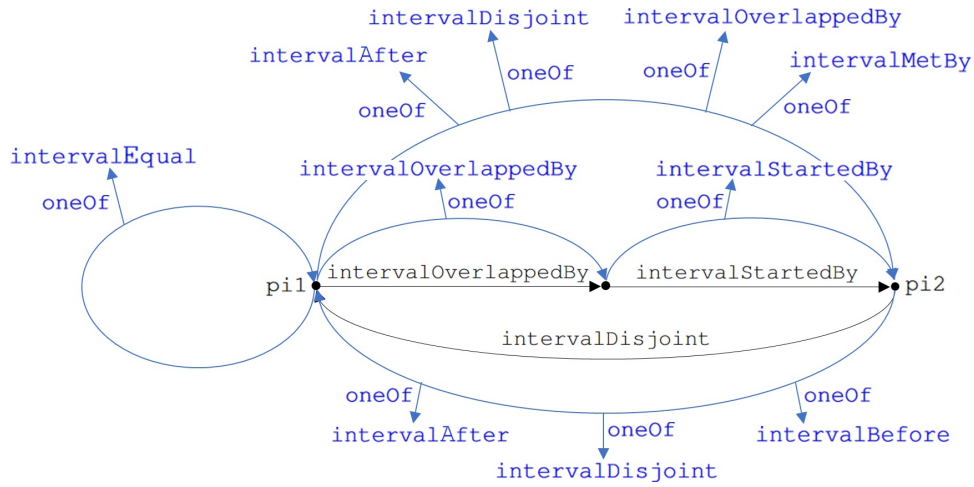


Fig. 32. Valid knowledge graph featuring a cycling path of properties denoting Allen's temporal relations. Inferred triples are shown in blue.

7.1. Evaluation

Evaluating the SHACL-SPARQL rules that implement the composition table in Table 2 is also rather simple because they are intended to parallel 1:1 the entries of the table itself. Therefore, to evaluate these rules, as in the previous section we carried out a simple generate & test evaluation: we implemented a small Java script that generates the composition table and we manually checked each rule's inferences one by one, i.e., that each rule outputs the same results reported in the corresponding table's cell. This script is also available in the GitHub repository associated with this paper, together with instructions to re-execute it; we invite the reader to locally re-execute the script and to double-check the inferences. Furthermore, we added the composition rules that compose `intervalEquals` with the other properties denoting Allen's temporal relations; these have not been included in Table 2 for space constraints and because they are rather simple: when a property denoting one of Allen's temporal relation is composed with `intervalEquals`, the result is simply the property itself.

Concerning the evaluation of the other SHACL-SPARQL rules introduced in this section, it is easy to see that the rules in (38) and (44) creates a reification of `hasPossibleATR` for each pairs of proper intervals connected by a path including one or more properties denoting Allen's temporal relations, in both directions.

It is likewise easy to see that the rule in (44) does not loop infinitely thanks to its NOT EXISTS clause that avoids reusing cycling paths. As exemplified in Figure 33, these are obtained when the reification associated with a path from two proper intervals `pi1` and `pi2` (shown in green in Figure 33) is conjoined with the reification associated with a path from two proper intervals `pi2` and `pi3` (shown in red in Figure 33), *such that the proper interval `pi3` is included in the set of proper intervals belonging to the first path.*

As the composition rules from Table 2 are assumed to be correct, also because they have been widely investigated and used in the last four decades, the set of possible properties calculated by the composition rules for the conjunction of the green and the red paths in Figure 33 coincides with the set of possible properties calculated by the composition rules for the conjunction of the orange and the purple paths in Figure 34, being the purple path a cycling path between `pi3` and itself. The set of possible properties associated with the purple path in Figure 34 must include the property `intervalEquals`, otherwise the SHACL shape in (43) would invalidate the knowledge graph.

It is now easy to see that, in cases where one of the possible properties associated with the purple path is `intervalEquals`, the set of possible properties associated with the conjunction of the orange and the purple paths is a *superset* of the set of properties associated with the orange path alone: as explained above, by composing *any* property with `intervalEquals`, the property itself is obtained. Therefore, if a property denoting one of Allen's temporal relation is explicitly asserted between `pi1` and `pi3`, and this property belongs to the set of possible properties associated with the orange path, then it will also belong to the set of properties associated with the conjunction of the orange and the purple path.

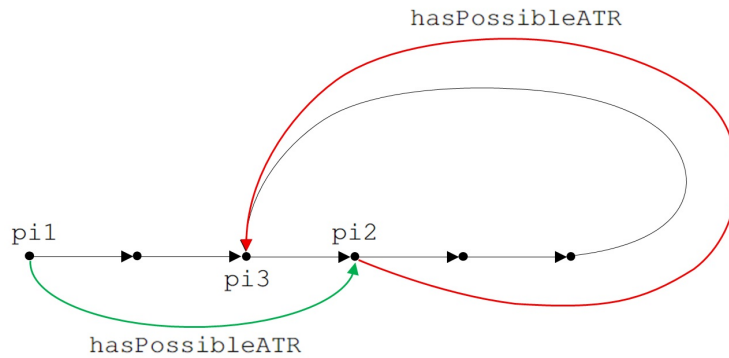


Fig. 33. Reifications of `hasPossibleATR` properties associated to a path of Allen's temporal relations that includes a cycle.

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 the validation of the properties with respect to the other sub-paths (i.e., the parts that do not include the cycling paths) plus the check that the cycling paths are associated with the property `intervalEquals`, done by the SHACL shape in (43).

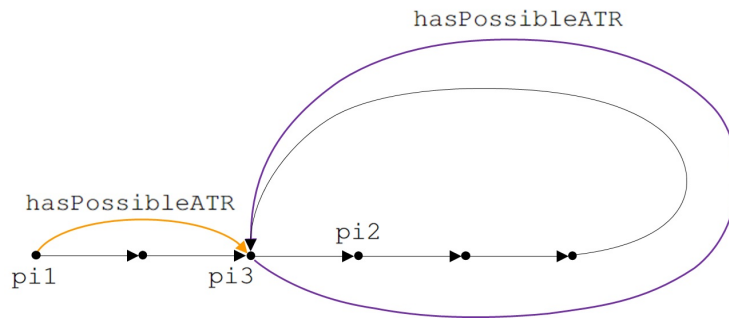


Fig. 34. Reifications of `hasPossibleATR` properties associated to a path of Allen's temporal relations that includes a cycle.

8. Conclusions and future works

This paper delivered two important contributions to the state of the art. First, it proposed a new version of the Time Ontology based on SHACL; this is crucial for further developments in academia as well as in industry, as time management is fundamental for the great majority of real-world applications. Secondly, our work on the Time Ontology highlighted novel insights on how validation and inference should be carried out in SHACL; specifically, the previous sections have shown that, in order to properly validate RDF knowledge graphs, the SHACL shapes must be executed on the transitive closure of the SHACL-SPARQL rules applied to the initial knowledge graphs.

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

Time management is needed to model and reason with evolution over time of actions, events, etc. For this reason, it is pervasively involved in applications intended to monitor or make predictions about real-world processes.

In the Semantic Web, the Time Ontology has become the standard “de facto” to represent temporal data. Many ontologies, in different domains, import the Time Ontology and use its resources to represent temporal entities.

However, the Time Ontology's resources have been merely used as a terminological vocabulary, i.e., to simply name instances, intervals, etc. with the same symbols defined in the ontology. On the other hand, to the best of our

1 knowledge no one has so far proposed methods to check and validate the data encoded through these symbols nor
 2 to reason about them in order to infer new relevant temporal entities. Validation is crucial to enable interoperability
 3 and data sharing, while inference might encourage the development of Artificial Intelligence methods to be used
 4 within the applications that employ the ontology.

5 The main reason why validation and inference with the Time Ontology’s resources have been so far neglected in
 6 the literature is that the ontology is encoded in OWL, whose vocabulary does not include operators to compare and
 7 manipulate temporal data. In light of the above considerations, it sounds quite evident that, without these operators,
 8 OWL is just inadequate to effectively deploy and use the Time Ontology within applications.

9 The present paper proposes instead to use SHACL shapes for validating knowledge graphs encoded through the
 10 Time Ontology’s resources and SHACL-SPARQL rules to make new inferences from these graphs. We then defined
 11 suitable shapes and rules to validate the *portion* of the Time Ontology shown in Figure 1. The Time Ontology’s
 12 resources occurring in Figure 1 are only those related to the `xsd:dateTime` datatype, which is currently the
 13 single temporal datatype supported by the official SPARQL v1.1 W3C recommendation.

14 Therefore, the first future work on the Time Ontology that we will devote our efforts to is how to extend the
 15 formalization proposed in this paper to the ontology’s resources that do *not* occur in Figure 1.

16 In particular, as our current formalization only encompasses instants, intervals, and their interrelations, the first
 17 reasonable extension appears to be the validation of *durations* of intervals. The Time Ontology also includes proper-
 18 ties to encode temporal entities’ durations, e.g., the property `hasXSDDuration`, which associates instances of the
 19 class `TemporalEntity` to values of the datatype `xsd:duration`. Therefore, for instance, if a proper interval
 20 begins on 1st January 2024, ends on 1st January 2025, and is associated, through the property `hasXSDDuration`,
 21 with the `xsd:duration` value “P10DT0H0M0”, which represents 10 days, then the proper interval is invalid:
 22 between the two dates there is a full year, not only 10 days. Calculating the exact duration by only using the oper-
 23 ators of the official SPARQL v1.1 release could be quite labour-intensive, as it would require to take into account
 24 different month durations, leap years/seconds, and other intricacies of the temporal reference systems. To overcome
 25 this problem, it must be however observed that several SPARQL implementations, e.g., the TopBraid SHACL Java
 26 library v.1.3.2 that we used in our implementation, calculates the duration among two `xsd:dateTime` values by
 27 simply subtracting one of them from the other within a SPARQL query, although this is not stipulated by the official
 28 SPARQL v1.1 recommendation. Still, this could not suffice to properly calculate durations in temporal reference
 29 systems different from the standard Gregorian calendar, which the Time Ontology supports (cf. [16]).

30 Similar considerations hold for other resources of the Time Ontology that allow for the encoding of partial and
 31 qualitative temporal data or for the encoding of temporal data in special coordinate systems. An example is the
 32 class `TimePosition`, whose instances specify “either a (nominal) value from an ordinal reference system, or a
 33 (numeric) value in a temporal coordinate system”. While the nominal value from an ordinal reference system is a
 34 string, i.e., a qualitative value that can be hardly be manipulated for comparisons, the numeric value in a temporal
 35 coordinate system, e.g., ISO 19108:2002²³, could be converted into an `xsd:dateTime` in order to reuse the
 36 SHACL shapes and the SHACL-SPARQL rules defined in this paper; however, the conversion appears to be rather
 37 hard to be implemented in terms of SHACL-SPARQL rules, at least with respect to the ISO 19108:2002.

38 Perhaps, in order to (easily) implement all these validation checks, and others that we do not discuss, a solution
 39 could be to use SHACL-X²⁴, a recent proposal to combine SHACL with JavaScript, thus enriching the former with
 40 the expressive power offered by a standard programming language such as JavaScript.

41 Other future works concern the definition of SHACL-SPARQL rules to use the Time Ontology within other
 42 ontologies. In fact, all SHACL-SPARQL rules defined above in this paper are *functional* to the validation, meaning
 43 that they are only needed to compute inferences that the SHACL shapes are unable to compute, as it will be explained
 44 in further detail in the next subsection. Therefore, simply put, once the knowledge graph has been validated, these
 45 inferred triples could be deleted, because the reason why they were computed (i.e., validation) has been achieved.

46 On the other hand, in some applications it could be necessary to infer new RDF triples in order to achieve the
 47 objectives of the applications themselves. Of course, applications should only work with valid knowledge graphs,
 48 but validating knowledge graphs could be hardly seen as an objective/service offered by a real-world application.

50 ²³<https://www.iso.org/standard/26013.html>

51 ²⁴<https://github.com/SHACL-X/shacl-x>

For example, [53] recently proposes an ontology to reason with obligations, permissions and other deontic statements. Importing the Time Ontology within the ontology proposed in [53] is indicated as a future work, in order, for instance, to infer from (48.a-b) that John violated the prohibition of entering the park, *but only from 4pm to 5pm*.

(48) 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 (48) respectively refer to a prohibition that held and to a fact that took place on the 1st January 2024, the two overlapping proper intervals mentioned in (48.a-b) can be represented as follows:

```
(49) :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 "2024-01-01T15:00:00Z".
      :e1 time:inXSDDateTime "2024-01-01T17:00:00Z".
      :b2 time:inXSDDateTime "2024-01-01T16:00:00Z".
      :e2 time:inXSDDateTime "2024-01-01T18:00:00Z".
      :pi1 time:intervalOverlaps :pi2.
```

The RDF triples in (49) are valid, as it can be easily seen by executing first the SHACL-SPARQL rules and then the SHACL shapes shown above. Still, none of the SHACL-SPARQL rule allows to infer the relevant interval that is needed to assess John's compliance with respect to (48.a). To do so, we need to introduce an additional SHACL-SPARQL rule that, taken two proper intervals that overlap, *creates* a new instance of `ProperInterval` that represents the interval shared by the two overlapping ones. This SHACL-SPARQL rule could be:

```
(50) [rdf:type sh:NodeShape;
      sh:targetSubjectsOf time:intervalOverlaps;
      sh:rule[rdf:type sh:SPARQLRule; sh:prefixes ... ;
      sh:construct """CONSTRUCT{[rdf:type 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 rdf:type time:ProperInterval;
      time:hasBeginning ?b; time:hasEnd ?e}}"""].
```

The SHACL-SPARQL rule in (50) creates the following anonymous individual, which represents the interval from 4pm to 5pm, i.e., the one in which John violates the prohibition in (48.a). Note that this new interval finishes `pi1`, i.e., the interval from 3pm to 5pm and starts `pi2`, i.e., the interval from 4pm to 6pm.

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

Nevertheless, the SHACL-SPARQL rule in (50) is only needed by the application for compliance checking envisioned in [53], while for other applications it could be irrelevant. Therefore, it was not included in the GitHub repository associated with this paper; the latter is application-neutral and, as such, it only includes rules to validate the portion of the Time Ontology shown in Figure 1.

In our future works, we will investigate and propose new SHACL-SPARQL rules for using the Time Ontology within external applications and case studies, e.g., the one envisioned in [53].

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

Although investigating novel effective ways to model and reason with temporal data in the Semantic Web is strategic for many applications, as argued in the previous subsection, our work on the Time Ontology even allowed us to realize more general insights about SHACL itself, specifically on how the intimate interplay between validation and inference should be implemented via the W3C standard. In other words, our work on the Time Ontology might be also seen as a *case study* for SHACL, shedding some light on how the format should be used.

This paper showed that the proper validation of temporal data encoded through the classes and properties of the Time Ontology requires two sequential steps: (1) first, we must compute the “fully” inferred knowledge graph, i.e., the knowledge graph obtained by iteratively applying the SHACL-SPARQL rules until no further triple is inferred; (2) then, the knowledge graph from (1) can be validated through the SHACL shapes. Of course, there is no apparent reason to conclude that our findings only hold for the Time Ontology; it rather seems to be a *general* scheme for carrying out validation and inference in SHACL, which must be followed for validating *any* knowledge graph.

The sequential procedure in (1) and (2) is *not* encompassed in the (current) definition of the SHACL rules from the W3C Working Group Note 08 June 2017²⁵. Indeed, the latter stipulates the reverse sequence: first the knowledge graph is validated through the shapes, then the rules are applied to the valid RDF triples. We therefore hope our work could lead to a new version of the W3C Working Group Note and, possibly, to future W3C recommendations.

A simple knowledge graph that exemplifies and supports our claims is the one shown in Figure 7. It is not possible to write a SHACL shape that validates zig-zag patterns such as the one shown in Figure 7, involving an arbitrary number of temporal entities: to cross these patterns we must use the SPARQL operators for encoding property paths²⁶; however, these operators are not enough expressive to impose further conditions on the RDF resources in the path, e.g., checking that they are instances of the class `Instant`, as in the example in Figure 7. Conversely, by first executing the SHACL-SPARQL rule in (9) we can (iteratively) connect the pairs of instants in the path, so that a simpler SHACL shape may check whether these are not associated with different `xsd:dateTime` values.

The need for the sequential procedure in (1) and (2) is even more evident when validating the Time Ontology’s properties that denote Allen’s temporal relations. It is in fact evident that SHACL shapes are not expressive enough to implement Allen’s interval algebra, i.e., the full composition table reported in Table 2. In order to do so, in our proposed solution we even had to introduce additional special properties, i.e., `hasPossibleATR` and `oneOf`, fit to represent sets of *possible* properties holding between two instances of `ProperInterval`. Triples asserted on these properties, as already explained in the previous subsection, are only functional to the SHACL shapes validating Allen’s interval algebra, i.e., they are intended to be deleted after validation.

Furthermore, even when the use of SHACL-SPARQL rules is *not* mandatory, because it is indeed possible to write SHACL shapes capable of validating the target patterns, often this is still practical and convenient, thus desirable. An example shown above was the introduction of SHACL-SPARQL rules to implement the inverse relations, e.g., the rule to derive triples asserted on the propriety *before* from the (inverse) triples asserted on the property *after*. Without this rule, it would be necessary to *double* the SHACL shapes defined on the property *before*, i.e., to introduce further parallel shapes for the property *after*. Introducing a single rule rather than several shapes leads to a less verbose formalization. These considerations of course hold in general: rather than writing complex SHACL shapes involving intricate property paths difficult to understand and to debug, it is often much more convenient to decouple the complexity of the task in two sequential phases, namely to introduce some SHACL-SPARQL rules that compute preliminary results whose validation requires much simpler SHACL shapes.

In light of the above discussion, this paper advocates the definition of a new W3C standard format for the Semantic Web that allows for the encoding of both (validation) shapes and (inference) rules and that stipulates to first execute the rules until no further RDF triple is inferred (exactly as it is done by standard OWL reasoner, e.g., `Hermit` [24]), and then to validate the inferred knowledge graph so obtained through the shapes. In other words, we hope the present work will trigger further research towards the definition of a single *unified* W3C standard format for both validation and inference that, in our view, should be carried out *together*.

²⁵<https://www.w3.org/TR/shacl-af>, retrieved June 1, 2024

²⁶<https://www.w3.org/TR/sparql11-property-paths>

Finally, it is worth concluding this paper by pointing out a crucial difference between OWL axioms and SHACL-SPARQL rules: while the iterative execution of OWL axioms always ends in a finite number of steps, the iterative execution of SHACL-SPARQL rules can loop infinitely. The reason is that the latter might also *create* new anonymous individuals, on which the rules might recursively (and infinitely) apply.

This is also done in the formalization proposed in this paper. For instance, section 6 above shows a SHACL-SPARQL rule that creates anonymous individuals reifying the `hasPossibleATR` properties. Another example is the SHACL-SPARQL rule shown in the previous subsection in (50), which creates a new proper interval out of two overlapping ones. In order to avoid these rules to loop infinitely, it was necessary to add `NOT EXISTS` clauses that create the new anonymous individuals only in cases where these individuals do not already exist.

Therefore, SHACL-SPARQL rules require more attention and care by the knowledge engineer than OWL axioms. This in turn demands for the creation of suitable editors and debuggers to help and assist their encoding. On the other hand, in our view the possibility to generate infinite loops should not be seen as a substantial pros of OWL with respect to SHACL-SPARQL. Main programming languages currently used, e.g., JavaScript, also allow to create infinite loops, e.g., “`while(true){}`”, therefore these considerations also apply to the SHACL-X proposal mentioned in the previous subsection. Still, this does not seem to be a good reason to restrict the expressivity of the programming language, as this expressivity is indeed required to properly represent many existing case studies. Conversely, as already argued above, we should rather create editors and debuggers that help programmers to quickly detect infinite loops, in cases where they occur.

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 open source 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. Zakharyashev, 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. Zakharyashev, Temporal Description Logic for Ontology-Based Data Access, in: *Proc. of the 23rd International Joint Conference on Artificial Intelligence (IJCAI) year = 2013*.
- [16] S. Cox, Time ontology extended for non-Gregorian calendar applications, *Semantic Web* **7**(2) (2016).
- [17] J.F. Allen, Towards a General Theory of Action and Time, *Artificial Intelligence* **23**(2) (1984).
- [18] P. Pareti, G. Konstantinidis, T. Norman and M. Sensoy, SHACL Constraints with Inference Rules, in: *The 18th International Semantic Web Conference (ISWC)*, Lecture Notes in Computer Science, Vol. 11778, Springer, 2019.
- [19] 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.
- [20] P. Pareti and G. Konstantinidis, A review of SHACL: From data validation to schema reasoning for rdf graphs, *Reasoning Web International Summer School* (2021).

- [21] L. Robaldo, F. Pacenza, J. Zangari, R. Calegari, F. Calimeri and G. Siragusa, Efficient compliance checking of RDF data, *Journal of Logic and Computation* **to appear** (2023).
- [22] J. Anim, L. Robaldo and A. Wyner, Compliance checking in the energy domain via W3C standards, in: *LNAI post-proceedings of JSAI-isAI: JSAI International Symposium on Artificial Intelligence. To appear, 2024*.
- [23] 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).
- [24] B. Glimm, I. Horrocks, B. Motik, G. Stoilos and Z. Wang, HermiT: An OWL 2 Reasoner, *Journal of Automated Reasoning* **53**(3) (2014).
- [25] R. Fikes and Q. Zhou, A reusable time ontology, in: *AAAI-2002 Workshop on Ontologies and the Semantic Web*, Citeseer, 2002.
- [26] 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.
- [27] R. Baumann, F. Loebe and H. Herre, Ontology of time in GFO, in: *Formal Ontology in Information Systems*, IOS Press, 2012, pp. 293–306.
- [28] V. Milea, F. Frasinca and U. Kaymak, tOWL: a temporal web ontology language, *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* **42**(1) (2011), 268–281.
- [29] 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*, Springer, 2008, pp. 31–45.
- [30] 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.
- [31] 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.
- [32] 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.
- [33] Y. Raimond, S.A. Abdallah, M.B. Sandler and F. Giasson, The Music Ontology., in: *ISMIR*, Vol. 2007, Vienna, Austria, 2007, p. 8th.
- [34] 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.
- [35] 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.
- [36] C. Tao, W.-Q. Wei, H.R. Solbrig, G. Savova and C.G. Chute, CNTR0: 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.
- [37] 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.
- [38] B. McBride, The resource description framework (RDF) and its vocabulary description language RDFS, in: *Handbook on ontologies*, Springer, 2004, pp. 51–65.
- [39] I. Horrocks, O. Kutz and U. Sattler, The Even More Irresistible SROIQ., *Kr* **6** (2006), 57–67.
- [40] G. Antoniou, S. Batsakis, R. Mutharaju, J.Z. Pan, G. Qi, I. Tachmazidis, J. Urbani and Z. Zhou, A survey of large-scale reasoning on the web of data, *The Knowledge Engineering Review* **33** (2018), e21.
- [41] H.-U. Krieger, A General Methodology for Equipping Ontologies with Time., in: *LREC*, 2010.
- [42] 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.
- [43] 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.
- [44] C. Welty, R. Fikes and S. Makarios, A reusable ontology for fluents in OWL, in: *FOIS*, Vol. 150, 2006, pp. 226–236.
- [45] S. Batsakis, E.G. Petrakis, I. Tachmazidis and G. Antoniou, Temporal representation and reasoning in OWL 2, *Semantic Web* **8**(6) (2017), 981–1000.
- [46] 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.
- [47] I. Horrocks, P.F. Patel-Schneider, H. Boley, S. Tabet, B. Groszof, M. Dean et al., SWRL: A semantic web rule language combining OWL and RuleML, *W3C Member submission* **21**(79) (2004), 1–31.
- [48] 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.
- [49] B. Glimm, I. Horrocks, B. Motik, G. Stoilos and Z. Wang, HermiT: an OWL 2 reasoner, *Journal of automated reasoning* **53** (2014), 245–269.
- [50] 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.
- [51] D. Wu, H.-T. Wang and A.U. Tansel, A survey for managing temporal data in RDF, *Information Systems* (2024), 102368.
- [52] J. Allen, Maintaining knowledge about temporal intervals (1983).

- 1 [53] L. Robaldo and G. Pozzato, Compliance checking in the Semantic Web: an RDF-based conflict-tolerant ver- 1
2 sion of the Deontic Traditional Scheme, *Semantic Web*, under review at [https://semantic-web-journal.net/content/](https://semantic-web-journal.net/content/compliance-checking-semantic-web-rdf-based-conflict-tolerant-version-deontic-traditional) 2
3 *compliance-checking-semantic-web-rdf-based-conflict-tolerant-version-deontic-traditional* (2024). 3
4 [54] S. Ahmetaj, R. David, M. Ortiz, A. Polleres, B. Shehu and M. Šimkus, Reasoning about Explanations for Non-validation in SHACL, in: 4
5 *Proc. of 18th International Conference on Principles of Knowledge Representation and Reasoning*, 2021. 5
6 [55] E. Anagnostopoulos, E. Petrakis and S. Batsakis, CHRONOS: improving the performance of qualitative temporal reasoning in OWL, in: 6
7 *2014 IEEE 26th International Conference on Tools with Artificial Intelligence*, IEEE, 2014, pp. 309–315. 7
8 [56] J. Pérez, M. Arenas and C. Gutierrez, Semantics and Complexity of SPARQL, *ACM Transactions on Database Systems* **34**(3) (2009). 8
9 [57] L. Robaldo, S. Batsakis, R. Calegari, F. Calimeri, M. Fujita, G. Governatori, M. Morelli, F. Pacenza, G. Pisano, K. Satoh, I. Tachmazidis 9
10 and J. Zangari, Compliance checking on first-order knowledge with conflicting and compensatory norms - a comparison among currently 10
11 available technologies, *Artificial Intelligence and Law to appear* (2023). 11
12 [58] X. Sun and L. Robaldo, On the complexity of Input/Output logic., *The Journal of Applied Logic* **25** (2017), 69–88. 12
13 [59] M. Leuschel and M. Butler, ProB: A model checker for B, in: *FME 2003: Formal Methods: International Symposium of Formal Methods* 13
14 *Europe, Pisa, Italy, September 8-14, 2003. Proceedings*, Springer, 2003, pp. 855–874. 14
15 15
16 16
17 17
18 18
19 19
20 20
21 21
22 22
23 23
24 24
25 25
26 26
27 27
28 28
29 29
30 30
31 31
32 32
33 33
34 34
35 35
36 36
37 37
38 38
39 39
40 40
41 41
42 42
43 43
44 44
45 45
46 46
47 47
48 48
49 49
50 50
51 51