**TEO-library**

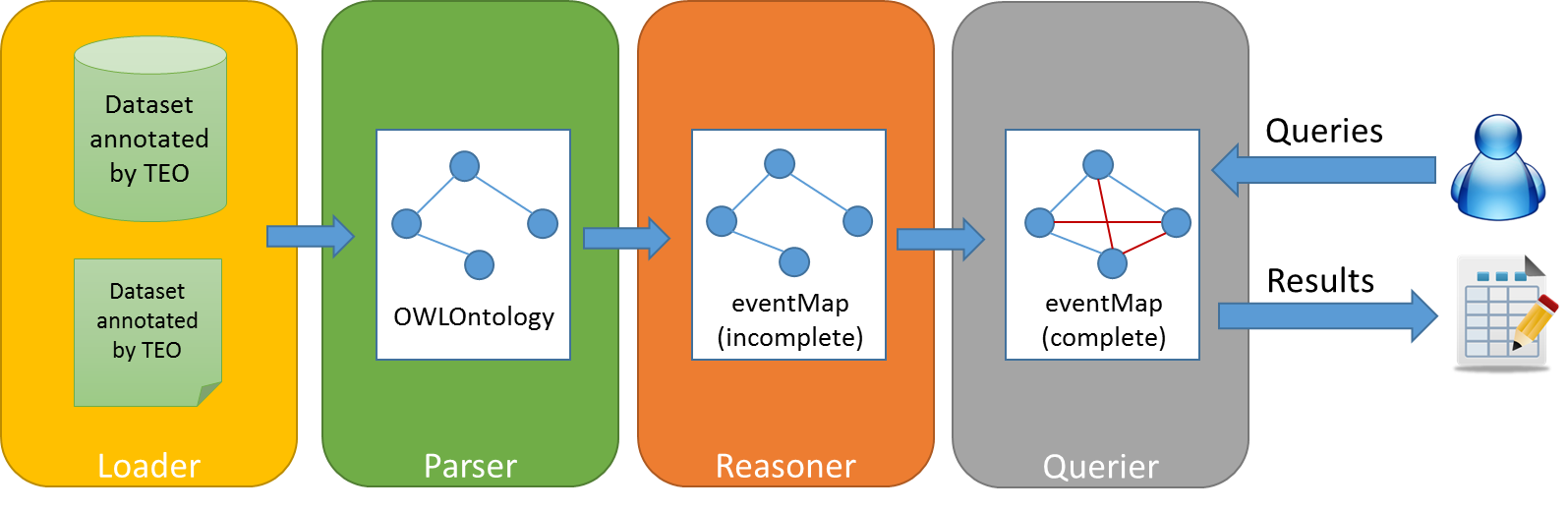
@Yi Luo

In practice, people often want to explore temporal information about events or entities in both free text and knowledge base. This demand can be formalized into two major tasks: temporal information extraction and temporal information reasoning. In this package, we focus on temporal information reasoning. We implement a Java library to load, parse, reason and query upon dataset annotated by the Time Event Ontology (TEO), which becomes an essential building block of the TIMER reasoner.

Resources we imported:

1. OWLAPI [1];
2. Pellet [2];
3. Jena [3];
4. PrettyTime NLP time parser [4];
5. Allen’s Interval Reasoning package [5].

**1. Package Architecture**



The loader utilizes OWLAPI to load the ontology files into memory in the form of OWLOntology. Both TBox (classes, properties and axioms) and ABox (individuals and assertions) are loaded.

The parser detects temporal events, their attributes and asserted temporal relations from the OWLOntology and builds up a standard HashMap as the output to represent the graph for future reasoning and querying. This process is supported by OWLAPI as well. In addition, the parser is responsible to normalize timestamps and detect corresponding granularities/units.

The reasoner is made up of the Pellet reasoner and our own implementation. We use Pellet to deal with basic and general inference such as for super/sub properties, transitive properties and inverse properties. Then we have two major components in our own reasoner. The first one “reasonValidTime()” is used to reason absolute timestamps for temporal\_regions such as the TimeInstant or TimeInterval (its Duration) of an event. This step takes full advantages of timeOffset information from the annotated dataset. After this process, associated events would be populated with detailed information in their validTime slot and pinned onto the timeline with absolute timestamps. The second step is “reasonTemporalRelations()”, which is designed to complete the temporal relation between any pair of events. In this reasoner, we introduce 12 point relations for absolute time reasoning and adopt 13 Allen’s interval relations for relative temporal relation reasoning.

The querier implements a series of APIs to query the graph.

**2. TEO Components**

2.1. TEOParser

In the TEOParser, we process temporal events, we parse its valid time and its temporal relation assertions. For the valid time parsing, we need to detect the event type – TimeInstant Event or TimeInterval Event. If the type information is not available, we may infer it later. When parsing the explicit dates and timestamps, we may encounter several problems caused by temporal granularities. So we first discuss how to handle these problems.

2.1.1. Granularity

For accurate temporal reasoning, granularity is a very important factor. Reasoning between events that have different time granularities contains potential problems. For example, if a TimeInstant event A is annotated with **10pm 8/5/2014** while TimeInstant event B is annotated with **Aug 2013**. What is the duration between them? If the user wants to know the duration at the **year** level, the reasoner can be sure to answer **one year** as the result. But if the user sets the granularity to **minute**, nobody could know the accurate answer so that the reasoner has to make some assumptions and return an uncertain result with friendly messages. In this section, we discuss how we deal with granularity-related problems in different scenarios.

1. *Granularity of TimeInstant: hasGranularity*

For a TimeInstant individual, the parser detects its granularity information from the Data Property “hasGranularity”. If this field is missing from the annotated data, the parser will use the default configuration which is UNKNOWN for the granularity.

1. *Granularity of Duration: hasDurationUnit*

Similarly, the parser investigates the Data Property “hasDurationUnit” for Duration individuals. If this field is missing, the parser uses UNKNOWN as the default unit for durations.

1. *Inference among TimeInstant1, TimeInstant2and Duration12 (|TimeInstant1 – TimeInstant2|)*

In many cases, we need to handle the reasoning among two time instants and the duration between them. For example, we may know the start and end time instants of a TimeInterval event and we need to calculate the duration; or we may know the start time of the source event and its time offset (duration) after the start time of the target event, we then need to calculate the start time of the target event. These tasks could be very intuitive and simple if their temporal information are at the same granularity level. However, it is non-trivial if they have different granularities. For example, if we know an event started yesterday (granularity: day) and lasted 2 hours (granularity: hour), are you able to answer the end time of it? Did it end before 11am? No one could answer it accurately. Therefore, we need special assumptions to handle these problems.

First, we introduce the granularity transformation with the following example. The original date is **2014-06-12 15:59:51**, then we list the same date in different granularities. The timestamp field in the milliseconds displays the time elapse since January 1, 1970 (midnight UTC/GMT), also known as Epoch Timestamp.

Original 2014-06-12 15:59:51 (GMT)

SECOND: 2014-06-12 15:59:51, timestamp = 1402606791000ms

MINUTE: 2014-06-12 15:59:00, timestamp = 1402606740000ms

HOUR: 2014-06-12 15:00:00, timestamp = 1402603200000ms

DAY: 2014-06-12 00:00:00, timestamp = 1402549200000ms

WEEK: 2014-06-08 00:00:00, timestamp = 1402203600000ms (Sunday)

MONTH: 2014-06-01 00:00:00, timestamp = 1401598800000ms

YEAR: 2014-01-01 00:00:00, timestamp = 1388556000000ms

From up to bottom, we can see the original TimeInstant is casted from a finer granularity to a coarser one with both its normalized time and epoch timestamp changed. This is what happens when we call the setGranularity() method to truncate a date with a finer granularity to the coarser granularity. In the contrary, when we reset the TimeInstant which has a coarser granularity to its finer level, we simply assume it happens at the beginning of every finer granularities. For instance, if the given time is **2014**, we assume it happens at **2014-01-01 00:00:00**.

Based on these assumptions, we propose algorithms to 1) calculate the Duration between two TimeInstants; 2) calculate the start (end) TimeInstant given the Duration and the end (start) TimeInstant.

1. The pseudo-code to calculate the duration given the start and end time, where the gran is the desired granularity:

getDuration(TimeInstant startTimeInstant, TimeInstant endTimeInstant, Granularity gran) {

**long** delta = *getDeltaMilliseconds*(startTimeInstant, endTimeInstant);

Granularity coarserGran = *getCoarserGranularity*(startTimeInstant, endTimeInstant);

**if** coarserGran > gran: // if the desired granularity is finer

print out message: the result is not accurate

Duration duration = **new** Duration(delta); // an inferred duration

**return** duration;

}

1. The pseudo-code to calculate the end time given the duration and the start time, where the gran is the desired granularity:

getEndTimeInstant(TimeInstant startTimeInstant, Duration duration, Granularity gran) {

**if** (startTimeInstant’s granularity > duration’s granularity):

print out message: the result is not accurate

Granularity finerGran = *getFinerGranularity*(startTimeInstant, duration);

**if** (finerGran > gran): // if the desired granularity is finer

print out message: the result is not accurate

Calendar endCal = Calendar.*getInstance*();

endCal.setTimeInMillis(startTimeInstant.getNormalizedTime());

endCal.add(duration); // end = start + duration

Date endDate = DateUtils.setGranularity(endCal.getTime(), gran);

TimeInstant endTimeInstant = **new** TimeInstant(endDate);// an inferred timeinstant

**return** endTimeInstant;

}

1. *Consistency checking of two TimeInstants and their Duration*

Sometimes, the start time, end time and the duration of a TimeInterval event are all provided by the source dataset, the reasoner has to check the consistency before it initializes a valid TimeInterval event. However, the problem is how to check the consistency if given TimeInstants and the Duration are at different granularities. The pseudo-code is as follows.

isValidTimeInterval(TimeInstant startTime, TimeInstant endTime, Duration duration) {

**if** (startTime <= endTime) { // is start <= end ?

Granularity maxGran = getMaxGranularity(startTime, endTime, duration);

Duration computedDur = TimeUtils.*getDurationFrom*(startTime, endTime, maxGran);

Duration givenDur = DurationUtils.*changeToUnit*(duration, maxGran);

**if** (computedDur == givenDur at maxGran):

**return** **true**;

}

**return** **false**;

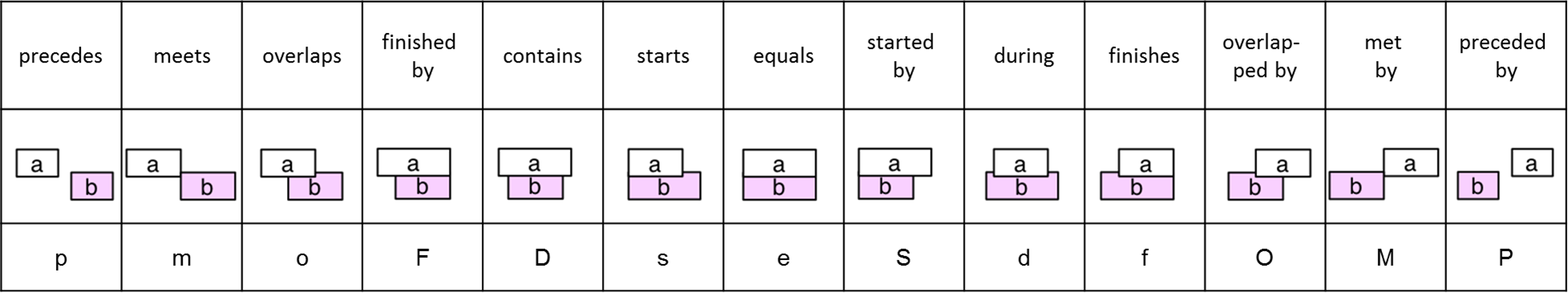
}

In addition, if the result is valid, we then select two fields that have finer granularities to initialize the TimeInterval instance. For instance, suppose we have the start time **Sept 1 2008** (granularity: day) and the end time **July 1 2012** (granularity: day) and the duration **4 year** (granularity: year). This interval passes the validation check and we select both the start and end TimeInstants to create a new TimeInterval object.

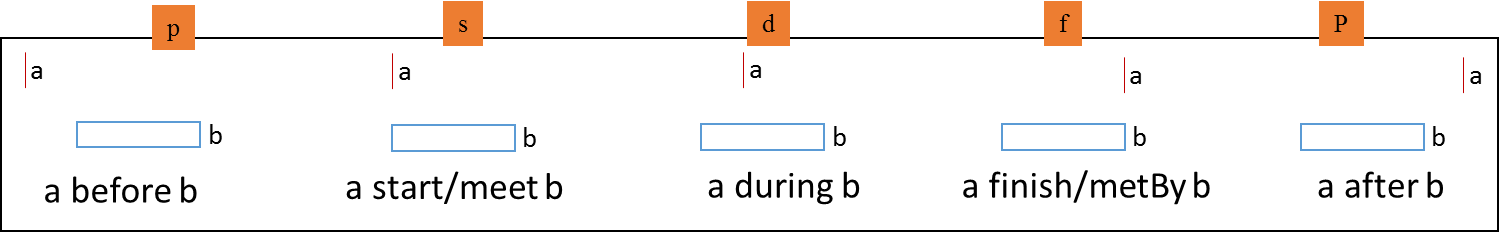
After parsing the valid time information, we then process temporal relations for each event. In this step, we need to handle two categories of relations: interval relations and pointer relations.

2.1.2. Interval Relations

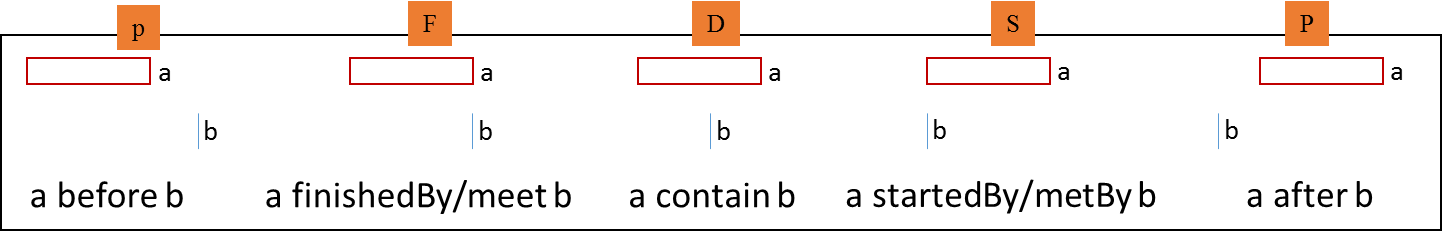
There are 13 interval relations proposed by Allen [6], which is also known as Allen’s Interval Algebra. The following table shows these 13 relations. It assumes both source and target events should be TimeInterval events. However, some relations are still compatible with TimeInstant events. Basically, if the given ***subject*** event is a TimeInstant event, its relations cannot be from the subset of [overlap, overlappedBy, startedBy, finishedBy, contain]; if the given ***object*** event is a TimeInstant event, its relations cannot be from the subset of [overlap, overlappedBy, start, finish, during].



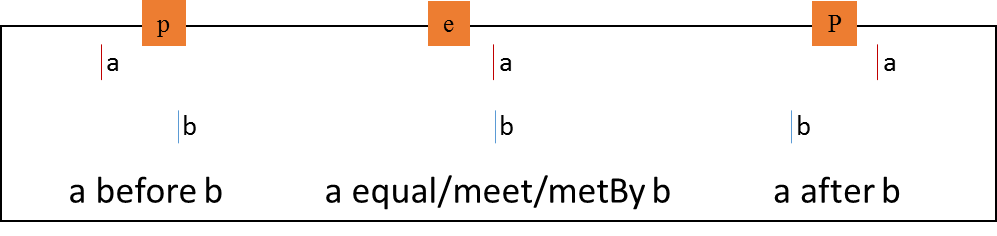
The following 3 figures exhibit 3 different situations: 1) the subject event (source event ***a***) is a TimeInstant event; 2) the object event (target event ***b***) is a TimeInstant event; 3) both events are TimeInstant events. We can see that only partial Interval Relations can be compatible with them. Then during the parsing, we need to check the consistency between the explicit event type and its temporal relations. Also, if the event type is not given explicitly when paring the valid time, we can infer its type here for some situations.



The subject event (source event ***a***) is a TimeInstant event



The object event (target event ***b***) is a TimeInstant event

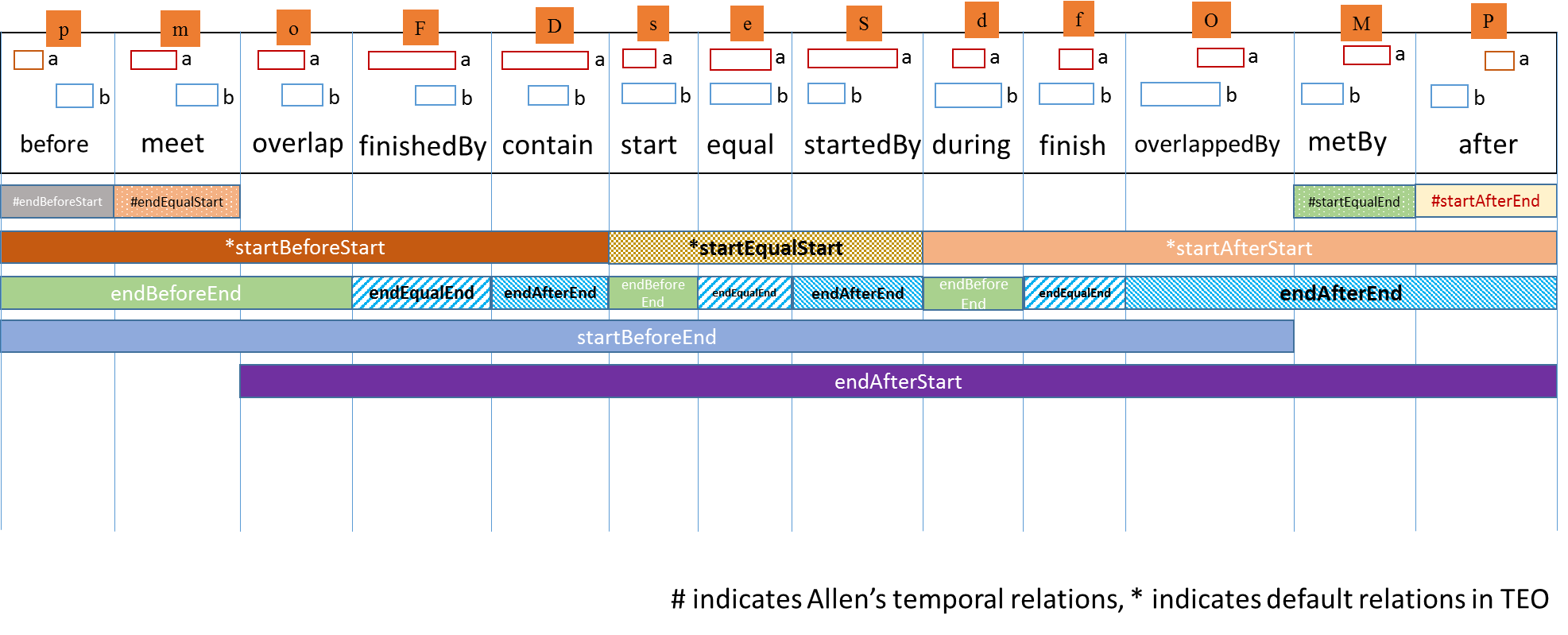


Both events are TimeInstant events

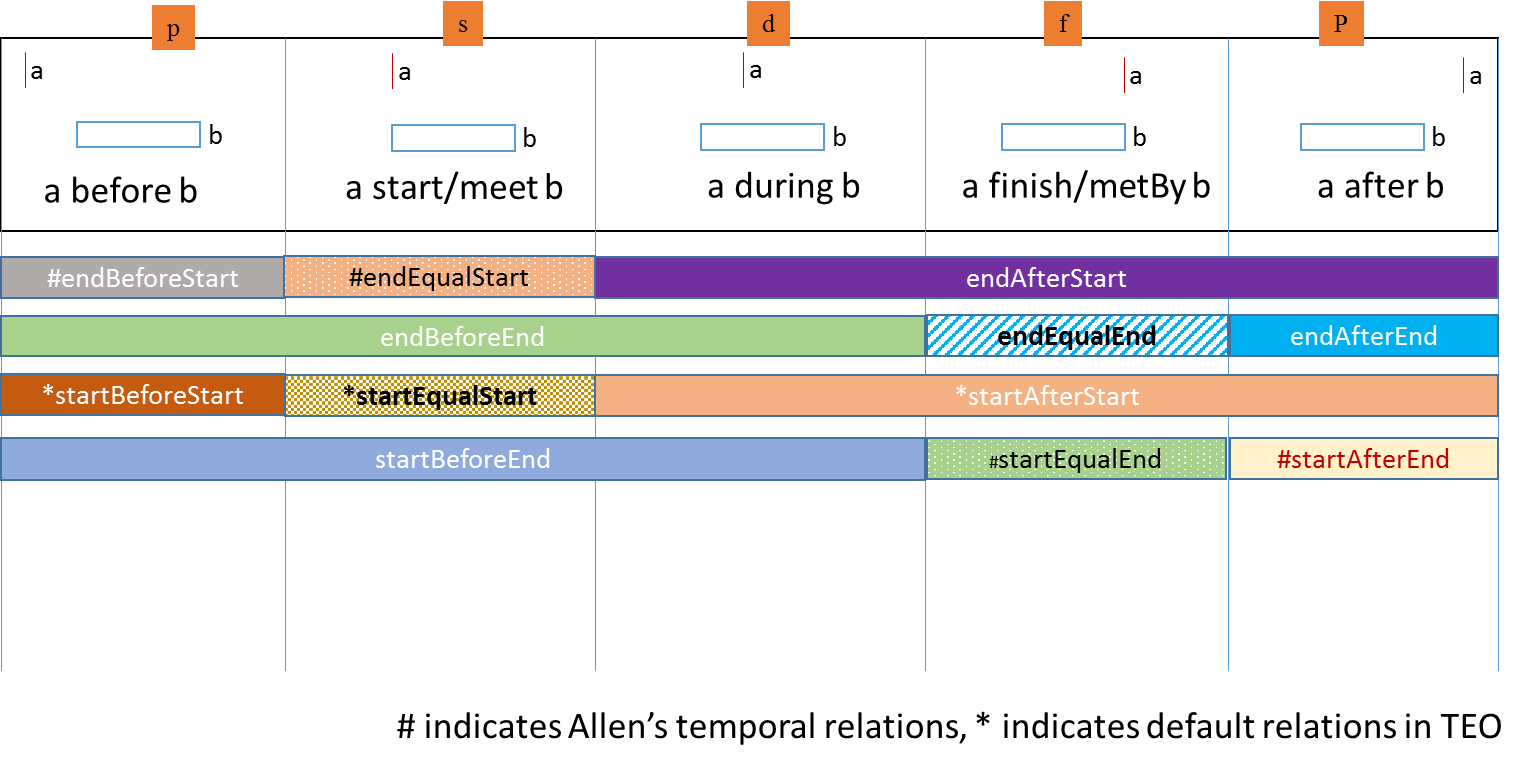
2.1.3. Point Relations

There are 12 point relations which are defined to handle time-offsets between time instants. For example, if the start time of Event\_A is **3 hours** before the start time of Event\_B, we can represent it as “Event\_A startBeforeStart Event\_B with timeOffset 3hours” (we will show the standard W3C annotation syntax later). Interval relations are relative relations which cannot well handle semantics about time-offsets, so the introduction of point relations becomes necessary. To bring in point relations, we assume an event has two time points – start time () and end time () where.

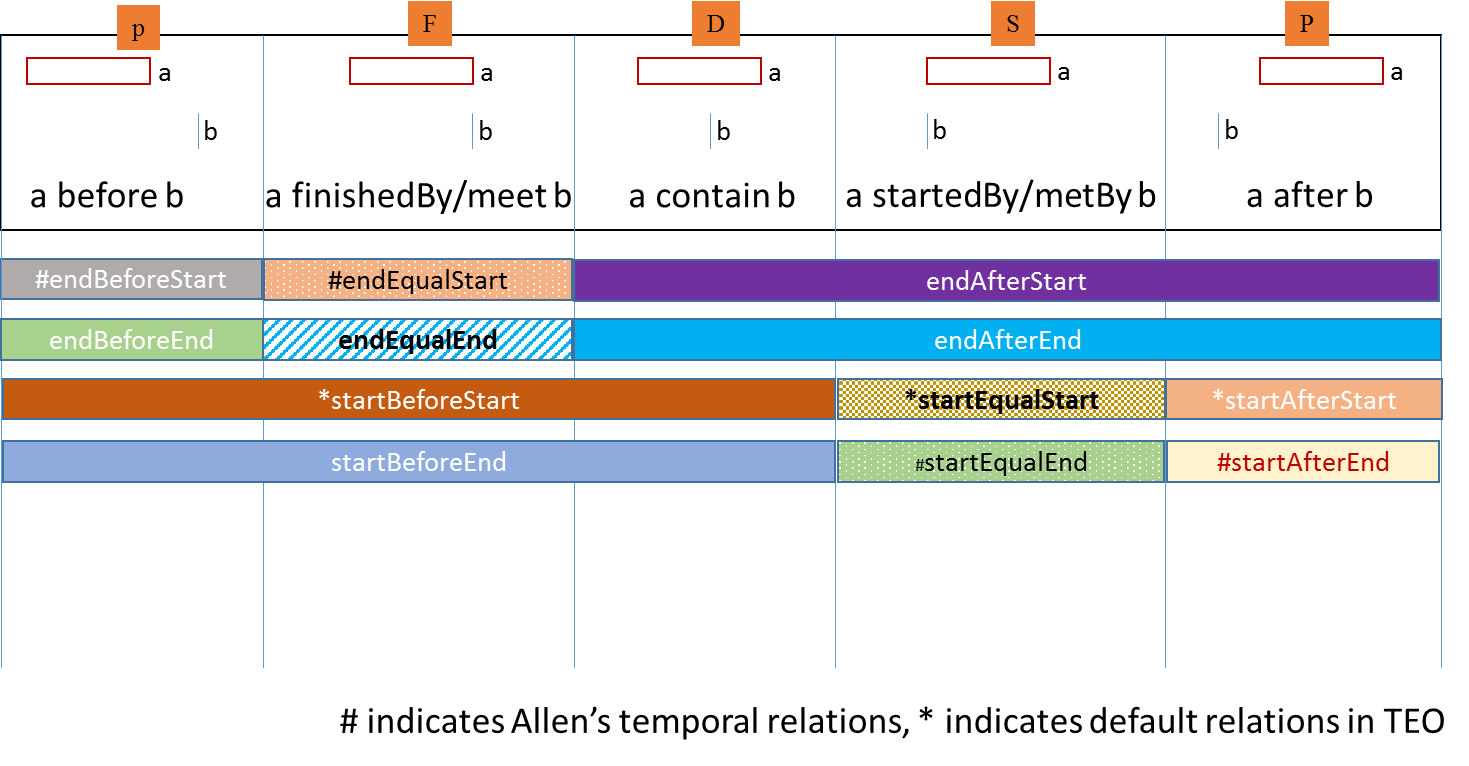
The following chart exhibits the mapping between interval relations and point relations.



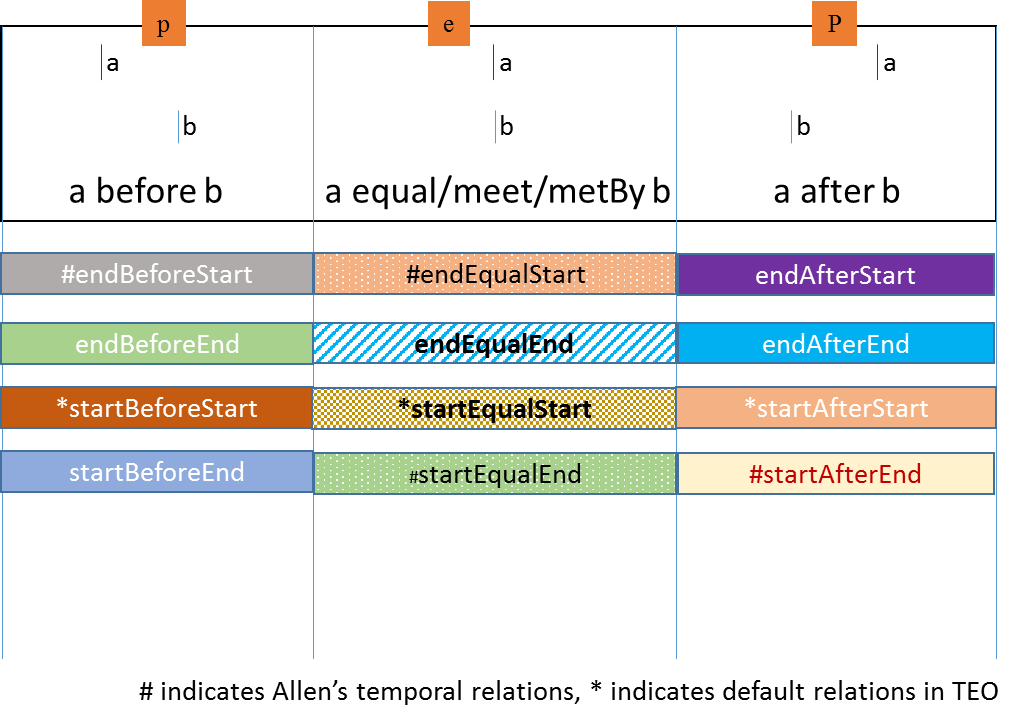
If the event is a TimeInstant event, we assume its start time equals its end Time. Hence we have another 3 figures below.



The subject event (source event ***a***) is a TimeInstant event



The object event (target event ***b***) is a TimeInstant event



Both events are TimeInstant events

Now we present a sample annotation of Event2 as follows. According to this annotation, Event2 is a named individual of TEO temporal event (teo:TEO\_0000025). It starts before the start of another event, namely, it has a property startBeforeStart (teo:TEO\_0000152) with the value being Event1. Furthermore, this startBeforeStart property is associated with a “property axiom annotation” which specifies the time-offset is Duration1.

:Event2 rdf:type teo:TEO\_0000025 ,

owl:NamedIndividual ;

rdfs:comment "event 2"@en ;

teo:TEO\_0000152 :Event1 .

[ rdf:type owl:Axiom ;

owl:annotatedProperty teo:TEO\_0000152 ;

teo:hasTimeOffset :Duration1 ;

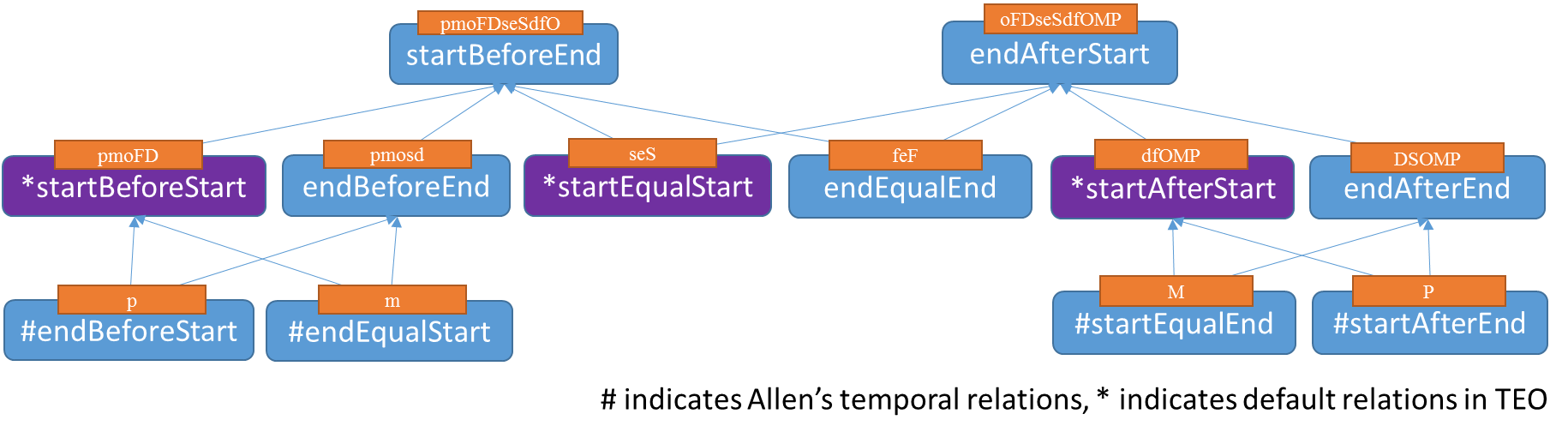
owl:annotatedTarget :Event1 ;

owl:annotatedSource :Event2

] .

2.2. TEOReasoner

After parsing the dataset into the ontology model, we then start the reasoning process. This process begins with a pre-reasoning step that calls Pellet to reason some basic class/property axioms. For temporal relations, it should add more triples according to the super/sub property hierarchy, transitivity axioms of property and equivalent property axioms. The following figure shows a property hierarchy. For example, if we parsed a triple “Even1 before Event2” we are now able to know that “Event1 endBeforeStart Event2”, “Event1 startBeforeStart Event2”, “Event1 endBeforeEnd Event2”, and “Event1 startBeforeEnd Event2”. The complete property axioms about temporal relations in TEO are given in the following list.



**Property Axioms - Pellet**

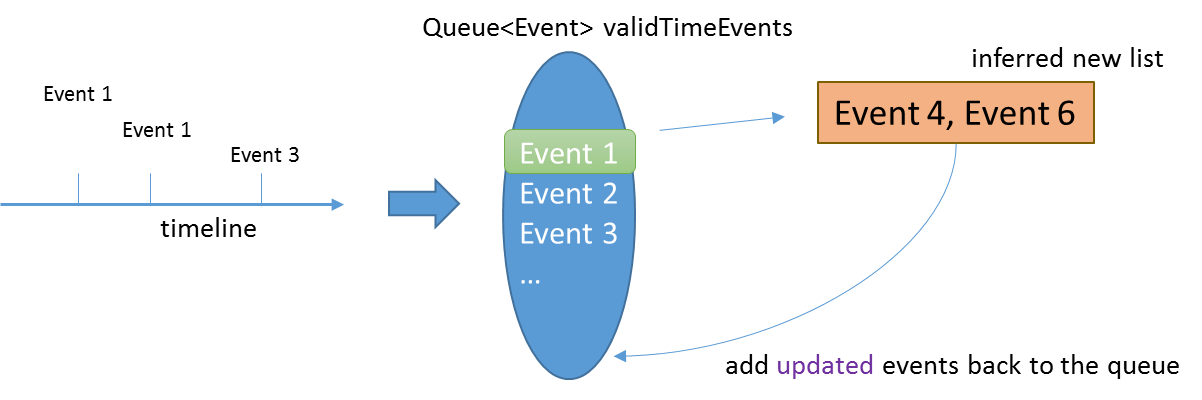
* *before* inverseOf *after*
* *meet* inverseOf *metBy*
* *overlap* inverseOf *overlappedBy*
* *start* inverseOf *startedby*
* *finish* inverseOf *finished*
* *contain* inverseOf *during*
* *equal* symmetric/transitive
* *before* = *endBeforeStart (TEO)*
* *after* = *startAfterEnd (TEO)*
* *before* transitive
* *after* transitive
* *meet* = *endEqualStart (TEO)*
* *metBy* = *startEqualEnd (TEO)*
* *endBeforeStart* inverseOf *startAfterEnd*
* *endBeforeEnd* inverseOf *endAfterEnd*
* *startBeforeEnd* inverseOf *endAfterStart*
* *startBeforeStart* inverseOf *startAfterStart*
* *endBeforeStart* transitive
* *startAfterEnd* transitive
* *startBeforeStart* transitive
* *startAfterStart* transitive
* *endBeforeEnd* transitive
* *endAfterEnd* transitive
* *startEqualStart* symmetric/transitive
* *endEqualEnd* symmetric/transitive

Besides property axioms, this pre-reasoning step also handles property axiom annotations (an OWL2 feature) of time-offsets, which is not supported by Pellet. For example, if we know “Even1 startBefore Event2 with timeOffset 2 years”, we should also infer that “Event2 startAfterStart Event1 with timeOffset 2 years”.

After this pre-reasoning step, we start two major reasoning phases – reasonValidTime() and reasonTemporalRelations(). The output ontology model from the parser turns to be the input of the reasonValidTime method, then the reasonTemporalRelations method.

2.2.1. reasonValidTime

This reasoning step tries to infer absolution temporal information, such as a fixed date, a start time instant or a timestamp based on point relations and the time-offsets. It pins those events with valid time information onto the timeline and use them to infer other events. The implementation can be represented as the following figure.



The algorithm maintains a queue of events that have valid time information and iterates through these events to infer new valid time information for other events. If an event is updated with new time information, the reasoner adds updated events back to the queue and continue the iteration. For example, if we have Event1 starts at 10am today, and Event1 has a point relation “Event1 startBeforeStart Event4 with timeOffset 5 hours”, we are able to infer that Event4 starts at 3pm today. If Event4 is updated with such new information, it should be added back to the queue for further reasoning, otherwise, we just ignore this event. As a result, the queue will finally shrink to an empty list and terminate this reasoning process. The following figure shows a testing example.

Event 1 instant: TimeInstant1 (10:30:45, 5/15/2009)

Event 2 instant: TimeInstant2 (10am, 5/20/2010)

Event 3 interval: TimeInterval1 (hasStartTime: TimeInstant2 (10am, 5/20/201),

hasDuration: Duration1 (1Y5M10D))

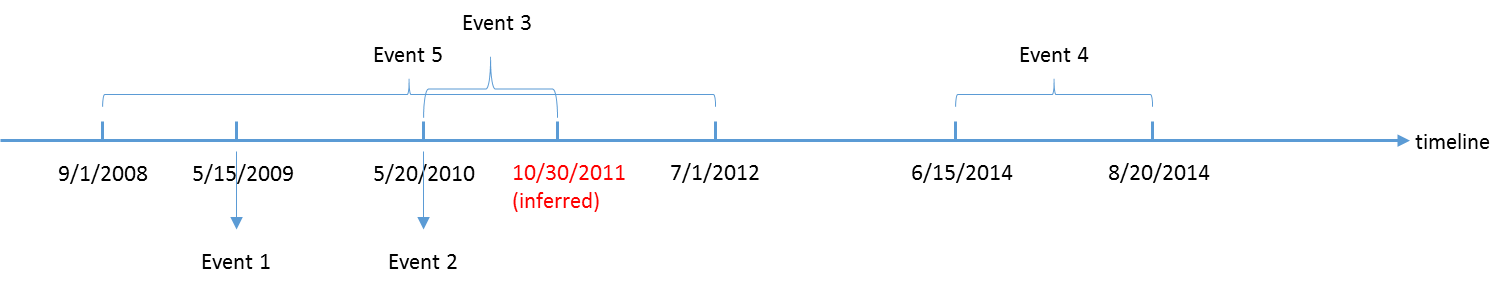
Event 4 interval: TimeInterval2 (hasStartTime: TimeInsant3 (10:30:00, 5/15/2014),

hasEndTime: TimeInstant4 (August 20, 2014))

Event 5 interval: TimeInterval3 (hasStartTime: TimeInstant5 (Sept 1, 2008),

hasEndTime: TimeInstant6 (July 1, 2012),

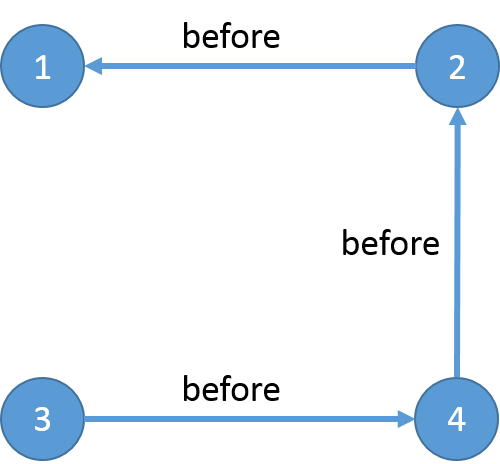
hasDuration: Duration2 (4Y));



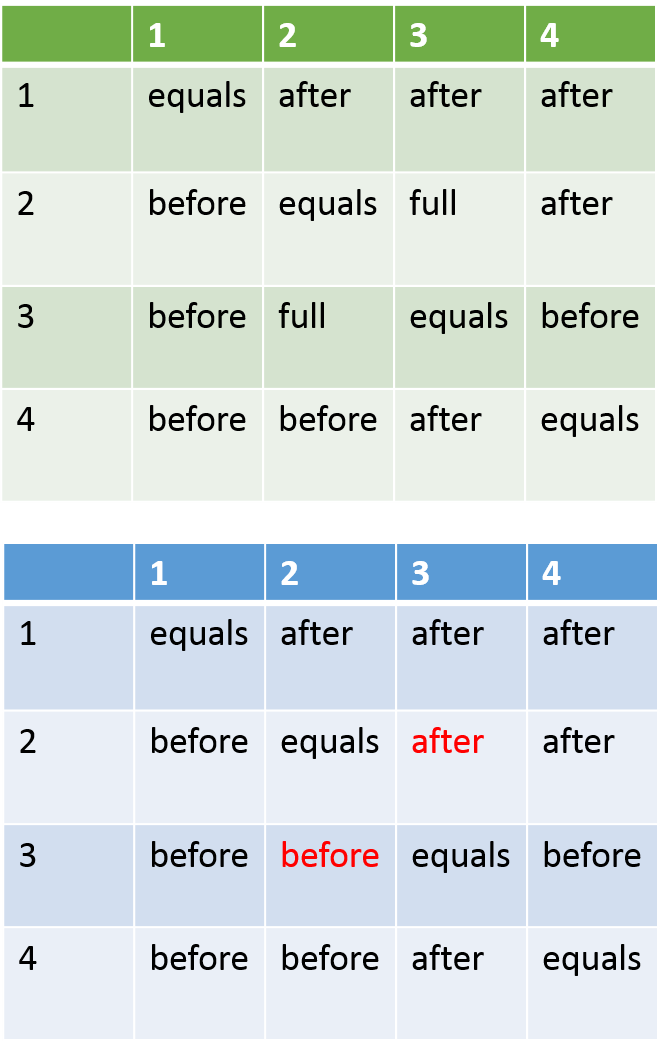
The completion of valid time information facilitates many queries, such as the query to retrieve the duration of an interval event and the query to get the duration between two time instants. It also helps to add extra temporal relations back to the graph. For example, if in the given annotation dataset we don’t have any temporal relation between Event\_m and Event\_n. But after the reasonValidTime process, we may know that “Event\_m ends at 10:00am today” while “Event\_n starts at 1:00pm today”, then we are able to add one temporal relation “before” between them. These new relations could be useful in the next reasoning step – reasonTemporalRelations.

2.2.2. reasonTemporalRelation

We can treat events and their temporal relations as a directed graph (see the following figure), after collecting all explicit relations between nodes, we hope to reason all potential relations in the graph and figure out relations between any pair of events. This reasonTemporalRelation process is indeed a graph completion task.



In another view, we want to fill in the matrix below.



To achieve this goal, we adopt the Allen’s Interval Algebra and its reasoning algorithm. The algorithm is based on the following transitivity matrix. For example, if we know:

:Event1 p (before) :Event2

:Event2 o (overlaps) :Event3

We are able to know that:

:Event1 p (before) :Event3

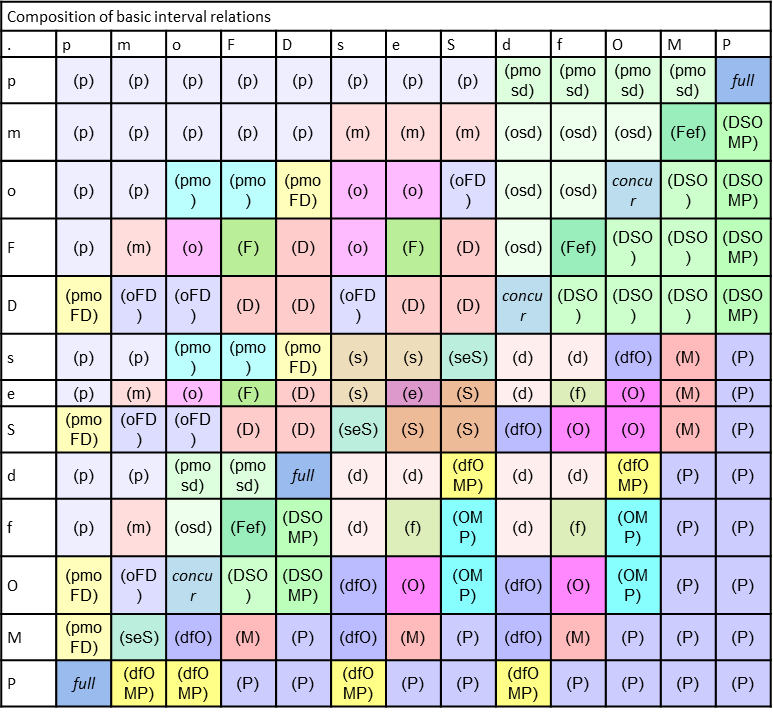


Figure from [7]

The most naïve reasoning algorithm was proposed by Allen in his initial paper [1]. The pseudo code of the algorithm is attached below. In our Java implementation, we used the Allen’s temporal reasoner library and adapted it a little to handle Point Relations as well.



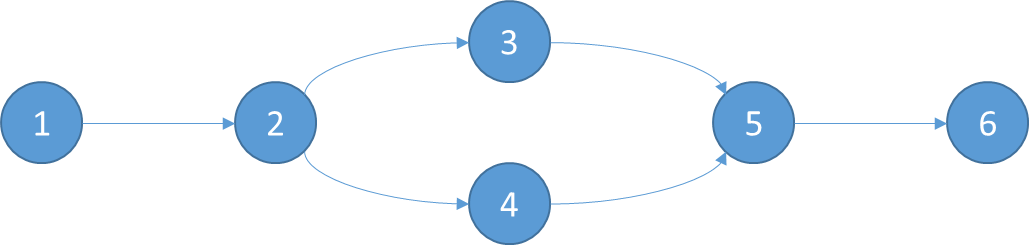
However, this naïve algorithm has its deficiency. Although it does not import inconsistency itself, it cannot detect the existing inconsistency in the graph so that it is not able to guarantee the global consistency eventually. We will improve the algorithm in the future.

2.3. TEOQuerier

We implemented our query APIs in the component and it directly interacts with end users. Currently, we have realized the following list of APIs:

1. List<Event> findEvents(String searchText);
2. Date getEventFeature(Event event);
3. Event getEventByIRIStr(String IRIStr);
4. Duration getDuration(Event intervalEvent);
5. Duration getDurationBetweenEvents(Event event1, Event event2, Granularity granularity);
6. List<Short> getTemporalRelation(Event event1, Event event2, Granularity granularity);
7. List<Event> getEventsTimeline();

The first 6 APIs are intuitive, we want to discuss how to generate the event timeline in details.



In our Java implementation, we only consider the start time of all events when computing the timeline. First, we figure out the point relations of start time instants between any pair of two events and obtain a Directed Acyclic Graph (DAG) as above. Then we are able to mode this problem as a topology ordering question. We implemented a classic algorithm with O(N+E) complexity. One thing we want to point out is that there might be equivalent nodes in this graph, for example, node 6 and node 7 (not included in the figure) may start at the same time then we should group them together in the final timeline. In fact, we used Disjoint Set to group equivalent nodes and selected one representative node for topology ordering computing.

2.4. TEOCalendarAnalyzer

We need this component to handle Special Calendar Elements defined in TEO, such as federal holidays and some customized special dates. The basic requirement of this component is to achieve two aspects of analysis: 1) to annotate a given date with an appropriate label according to which special dates it belongs to; 2) to translate Special Calendar Element definitions in TEO (as a TEO class) into human understandable rules, such as “the 5th day in May”, and enumerate specific dates if user further specifies the year. Currently, the class template of a Special Calendar Element consist of 4 properties: occurDay, occurMonth, occurWeek and occurYear.

To achieve the first requirement, we take advantage of the Pellet reasoner which is able to infer class membership for an ontology. Given a date, we calculate its occurDay, occurMonth, occurWeek and occurYear with the help of the Java build-in Calendar class and create an individual with these properties. Then by calling Pellet we are able to get the Class this individual belongs to. For example, if the input date is 07/07/2014, we can create an individual as “occurMonth: 6 (July), occurYear: 2014, occurDay: 4 (monthDay), occurDay: 6 (weekday), occurWeek: 1”. When calling the Pellet reasoner we are able to output the holiday class this date belongs to which is

“<http://informatics.mayo.edu/TEO.owl#TEO\_0000069>: "Independence Day"”.

To realize the second requirement, we need to parse the class definition from the TEO first. This task can be difficult when the definition is complicated. A simple definition can be as the “Independence Day” as follows:

(occurMonth some July)

and (occurDay some 'Day 4 of a month')

and (occurYear some int[>= "1775"^^int])

But a complicated definition can be as the “Inauguration Day” as follows:

(((occurDay some Monday)

and (occurDay some 'Day 21 of a month'))

or ((occurDay some 'Day 20 of a month')

and (occurDay some

(Monday

or Tuesday

or Wednesday

or Thursday

or Friday

or Saturday))))

and (occurMonth some January)

and (occurYear some int[>= "1937"^^int])

In fact, the second example shows two possible rules of the “Inauguration Day” which is connected with OR operator. Therefore, to parse the definition, we decide to represent the logic expression in disjunctive normal form (DNF) as the beginning step, then treat each disjunctive component as a rule. Let’s take the “Inauguration Day” as an example, suppose we replace each atomic rule (occurDay, occurWeek, occurMonth and occurYear) with the abbreviation. Then the original rule looks like:

( ( ( R1 & R2 ) | ( R3 & R34 ) ) & R5 & R6 ) .

We transfer it into DNF as:

R1 & R2 & R5 & R6 | R3 & R4 & R5 & R6

Then we are able to parse it into two sub-rules as:

* Rule1:

{[Year: (1937, Integer.Max)] // year >= 1937

[Month: (0)] // month = January

[MonthDay: (21)] // 21st day of the month

[Week: (null)] // monthWeek = null

[WeekDay: (2)]} // weekday = Monday

* Rule2:

{[Year: (1937, Integer.Max)] // year >= 1937

[Month: (0)] // month = January

[MonthDay: (20)] // 20th day of the month

[Week: (null)] // monthWeek = null

[WeekDay: (2, 3, 4, 5, 6, 7)]} // weekday = Mon | Tue | Wed | Thr | Fri | Sat

Therefore if we further constraint the year as 2014 we can get the “Inauguration Day” is 01/20/2014, or if we set the year as 2019 we can get the date “01/21/2019”.

**References:**

1. OWLAPI, http://owlapi.sourceforge.net/
2. Pellet, http://clarkparsia.com/pellet/
3. Jena, https://jena.apache.org/
4. PrettytimeNLP, http://ocpsoft.org/prettytime/nlp/
5. Allen’s Temporal Relation Reasoner, https://code.google.com/p/allenintervalrelationships/
6. Allen, James F. "Maintaining knowledge about temporal intervals". Communications of the ACM 26(11) pp.832-843, Nov. 1983
7. Allen’s Interval Algebra, http://www.ics.uci.edu/~alspaugh/cls/shr/allen.html