THE CASE STUDY ON REPRESENTATION OF TEMPORAL LOGIC

Shweta Jain
School of computer science and engineering
VIT UNIVERSITY
Vellore, India
nrprashant123@gmail.com

Shrikant Ujjainkar
School of computer science and engineering
VIT UNIVERSITY
Vellore, India
shrikant41@gmail.com

Prashant Mandloi
School of computer science and engineering
VIT UNIVERSITY
Vellore, India
nrprashant123@gmail.com

RA. K. Saravanguru
School of computer science and engineering
VIT UNIVERSITY
Vellore, India
sarophd@gmail.com

Abstract
The temporal logic is system of rules and symbols for represent the terms related to time. In this paper, the case study has been done for representing the temporal logic in Timed I/O Automata and Ontology. Also, the case study of Java messaging service has been done for representing the time in message passing using publish and subscribe. Timed I/O Automata is used for representing the timing constraints. Tempo is a simple formal language for modelling distributed systems as collections of interacting state machines called Timed Input/output Automata. Dealing with information that changes over time is a critical problem for practical Knowledge Representation languages (OWL, RDF). These languages are based on binary relationships between concepts. Time can be represented using protégé tool. The case study is on Veromodo’s toolkit TEMPO and protégé plug-in CHRONOS.

Keywords- Tempo, Chronos, Ontology, I/O Automata, Publish/Subscribe, Messaging Queue.

1. Introduction

Temporal Logic has its origins in philosophy where it was used to analyze the structure or topology of time. Philosophers introduced special temporal operators such as 0 henceforth
and 0 eventually, for the analysis of connectives in language. As a valuable tool for analyzing the topology of time, various types or semantics can be given to the temporal operators. The Timed I/O Automata mathematical framework is an extension of the classical I/O Automata framework, which was developed many years ago in the theoretical distributed algorithms research community. I/O Automata are very simple interacting asynchronous state machines, without any support for describing timing features. Although they are simple, I/O Automata provide a rich set of capabilities for modelling and analyzing distributed algorithms. I/O Automata support description of many properties that distributed algorithms are required to satisfy, and mathematical proofs that the algorithms in fact satisfy their required properties. These proofs are based on methods such as invariant assertions and compositional reasoning. I/O Automata also support representation of algorithms at different levels of abstraction, and proofs of consistency relationships between algorithm representations at different levels. Because of these capabilities, I/O Automata have been used fairly extensively for modelling and analyzing asynchronous distributed algorithms, and even for proving impossibility results about computability in asynchronous distributed settings.

Ontologies are used to capture knowledge about some domain of interest. The most recent development in standard ontology languages is OWL from the World Wide Web Consortium (W3C). Properties in OWL are of two types, data properties and object properties. Data properties, which relate an individual to an XML Schema data type and Object properties, which are relations on individuals. Both property types are binary. This feature of OWL makes representation of temporal relations very difficult because a temporal relation is represented as a ternary relation between two instances and the interval during which the relation holds, which is not permitted in OWL.

The Java Message Service (JMS) is a specification developed by Sun Microsystems that provides a common way for Java applications to access each other. The JMS specification defines a set of interfaces and associated semantics that allow Java applications to create, send, receive, and read messages. The specification does not define how messages need to be transported with in a particular implementation, known as a JMS Provider. This clear separation of concerns was essential in order to allow vendors of existing messaging products to support the JMS specification [12]. Furthermore, some JMS vendors provide multiple message-transportation implementations within the same product thereby providing the user with the ability to select the most appropriate transport technology for a particular deployment. An example is the Arjuna Message Service [13], which provides both server-based and server-less message transport technologies that can be mixed within the same application.

2. Related Work

In [1], the main objective of this survey paper is to inform a reader about the theoretical properties (or capabilities) of timed automata which are (or might be) useful for real-time model driven development. To achieve this goal, this paper presents a survey on semantics, decision problems, and variants of timed automata. In [2], this paper addresses the generation of complete abstractions of polynomial dynamical systems by timed automata. For the proposed abstraction, the state space is divided into cells by sublevel sets of functions.
In [3], this paper presents a methodology and tool that enables a formal reasoning about the correctness of reactive/control software written in Simulink/Stateflow through its translation into an automaton model. In [4], this paper, Timed Automata is an extension to the automata theoretic approach for the modeling of real time systems that introduces time into the classical automata. In [5], this research proposal they discuss three of these issues: representation round tripping, concept drift, and contextual knowledge. We advocate an integrated approach to solve them, and present some preliminary results. In [6], this paper is intended to serve as a comprehensive introduction to the emerging field concerned with the design and use of ontologies. They have observed that disparate backgrounds, languages, tools, and techniques are a major barrier to effective communication among people, organisations, or software systems.

In [7], this paper they have discussed the current trends in ontology building and its future challenges, namely, the new issues for building ontologies for the Semantic Web. In [8] this, they first describe the motivation for OWL in terms of its requirements, and the resulting non-trivial relation with RDF Schema and then describe the various language elements of OWL in some detail. In [9], paper focuses on real-world scenarios in which messaging may be used, and provides a detailed performance comparison between two JMS implementations: The Java System Message Queue and the IBM WebSphere MQ products. In [10], paper focuses on investigating the capacity of the high performance JMS server implementation ActiveMQ. The authors of this paper had determined that the message throughput in the presence of filters and showed that filtering reduces the performance significantly.

3. Case Study 1: CHRONOS, a Protégé Plugin

Temporal relations are those which changes over time. Although the syntax of temporal ontologies have been defined by W3C, but it is very difficult to develop a temporal ontology manually. To solve this problem a plugin for protégé was developed in 2011 known as Chronos. Chronos allows us to create and edit temporal ontology. It handles the temporal ontology as static ontology. Chronos reduces the overhead involved in the creating the temporal ontology. It has following desirable features:
1. Chronos is easy to use, handles temporal ontologies as the static ones, allowing the user not to be familiar with the peculiarities of the underlying temporal representation model.
2. Supports restriction adding and checking restriction on temporal properties (e.g. A student cannot study in two different university at the same time), classes (e.g., a branch cannot accommodate more than 90 student at the same time) and on individuals.
3. Supports reasoning over the temporal ontologies using the standard Pellet reasoner for Protege.
4. Chronos interface is consistent with the layout of the default Protege tabs.

3.1 Temporal Representation
Dealing with information that changes over time is a critical problem for practical Knowledge Representation languages (OWL, RDF). These languages are based on binary relationships between concepts.

Time can be represented using various Models is discussed as follows:

1. Reification is a general purpose technique for representing n-ary relations using languages that only permit binary relations. An n-ary relation is represented as an object, which is the
subject of \( n + 1 \) triple. Those triples have as objects the participants of the \( n \)-ary relation plus one triple has as object the predicate of the property. In our example, supposing that we want to represent the statement "John has joined a university Stanford from 2000 to 2005" is expressed as studiedin (John, Stanford, t). Where t represents the interval '2000 to 2005'. Using reification, this relationship is represented as a new object with John, Stanford, t and studiedin being objects of properties. A major disadvantage of reification is that it offers limited reasoning capabilities. This is a result of representing the predicate of the relation as an object of a property thus the OWL semantics over the property is no longer applicable.

Figure 1: Reification

2. \( N \)-ary Relations is a general purpose technique for representing \( n \)-ary relations. In \( N \)-ary Relations representation model, the static entities are considered to participate in events. A temporal property property between two individuals (e.g. Employee works for Company) holds as long as that event endures. The \( n \)-ary property is represented as a class rather than as a property. Instances of such classes correspond to instances of the relation. Additional properties provide binary links to each argument of the relation. In contrast to reification, the \( n \)-ary relation is not represented as the object of a property but as two properties each related with the new object.

Figure 2: \( N \)-ary Relation

3.2 Properties and individuals

3.2.1 Object Properties
The representation of object property using \( N \)-ary representation requires some changes to be introduced in static properties. In this representation an instance of event class in introduced between the subject and object of the static properties. The domain and the range of the static property is also change. For example the Domain of the static property "studiedin" becomes Event or Student and the range become Event or University.
3.2.2 DataType Properties
These properties are handled by Chronos in a similar way to object properties. The dynamic data property's domain is the union of the static property's domain and the Event class. The main difference with the object properties is that the range cannot be the union of a data type and the Event class. To overcome this problem, Chronos create an object property named by the data property, and followed by “OP”. This object property will relate the static data property's domain to the Event class. This is also made a sub-property of the participatesIn object property. The data property with the modified domain will connect the event to the data type. As in object property conversion, the Event is related to the Interval with the during object property. An example of a converted data property is illustrated in Figure 4 the examples show how would the temporal data property hasprojectperiod have been represented, according to the approach.

3.2.3 Individuals
They represent objects in the domain of interest. For example, `John' is an individual of the Class `Person'. The statement “John has studied in Stanford from 1920 to 1998” does not require a temporal property for its representation. However, when a property evolves over time, such as in the statement John has studied in Stanford from 1920 to 1998; the property studied in is a temporal property that holds during a specific time interval. When a property is converted to temporal, all the triples that contain this property are converted too. For each triple in the ontology, a new instance of the Event class is created and introduced between the subject and the object, as explained in subsections. This event individual is connected to a TimeInterval instance with the during object property. The TimeInterval individual is related to two instant individuals, one that represents the starting point of the interval and one that represents the ending point of the interval. Each of these Instant individuals is connected to a dateTime data type with the data property inXSDDateTime.
3.2.4 Cardinality

3.2.4.1 owl:someValuesFrom

The value constraint owl:someValuesFrom is a built-in OWL property that links a restriction class to a class description or a data range. A restriction containing an owl:someValuesFrom constraint describes a class of all individuals for which at least one value of the property concerned is an instance of the class description or a data value in the data range. The restriction, in the case that a temporal property is concerned, is used to describe a class of all individuals for which at least one value of the property concerned is an Event individual that is connected with the concerned property to an instance of the class description or a data value in the data range.

For example, a restriction on a class `university' could be:

\[ \text{hasJoined some university} \]

If the object property hasJoined is temporal, the restriction becomes:

\[ \text{hasJoined some(Event and hasJoined some university)} \]

3.2.4.2 owl:hasValue

The value constraint owl:hasValue is a built-in OWL property that links a restriction class to a value V, which can be either an individual or a data value. A restriction containing a owl:hasValue constraint describes a class of all individuals for which the property concerned has at least one value semantically equal to V (it may have other values as well). This temporal constraint is applied with the addition of an SWRL rule that to the ontology. For example, an owl:hasValue restriction on a class `Student' is:

\[ \text{hasJoined value stanford} \]

where `stanford' is an individual of the class university. If the object property hasJoined is temporal, the restriction becomes:

\[ \text{hasJoined some(Event and hasJoined value stanford)} \]

This temporal constraint is applied with the addition of an SWRL rule that to the ontology. For The SWRL rule that would be added to the ontology to apply the temporal constraint would be:

\[
\text{student(?x)^participatesIn(?x, ?e)^Event(?e) \rightarrow hasJoined(?e,stanford)^hasJoined(?x, ?e)}
\]

Meaning that for each event that an individual of the class `student' participates in, that student individual is also related to the `Stanford' with the temporal object property hasJoined.

Examples
• For determining the Which event like term end exam or midterm exam in college has to start at a given date and time then for this DLquery will be given as follows:
  hastostartOP some(Event and (during some(ProperInterval and (intervalAfter some(DateTimeInterval and (hasBeginning some (Instant and inXSDDateTime value "2013-10-01T12:00:00"^^dateTime))))))

• If we need to determine which student has joined the college that a given date and time then DLquery will be as follows:
  hasjoined some (Event and during some (ProperInterval and hasBeginning some (Instant and inXSDDateTime value "2013-08-02T10:00:00"^^dateTime)))

4. Case Study 2: Veromodo toolkit TEMPO

I/O Automata support description of many properties that distributed algorithms are required to satisfy, and mathematical proofs that the algorithms in fact satisfy their required properties. These proofs are based on methods such as invariant assertions and compositional reasoning. I/O Automata also support representation of algorithms at different levels of abstraction, and proofs of consistency relationships between algorithm representations at different levels. Because of these capabilities, I/O Automata have been used fairly extensively for modelling and analyzing asynchronous distributed algorithms, and even for proving impossibility results about computability in asynchronous distributed settings.

However, ordinary I/O Automata cannot be used to describe distributed algorithms that use time explicitly, for example, those that use timeouts or schedule events periodically. And they do not provide explicit support for describing timing constraints such as bounds on message delay or clock rates. Moreover, without support for timing, I/O Automata could not be used for other applications such as practical communication protocols. These limitations led to the development of Timed I/O Automata, which include new features most notably, trajectories specially designed for describing timing aspects of systems.

Like ordinary I/O Automata, Timed I/O Automata are simple interacting state machines. They have a well-developed, elegant theory, which is presented in a separate monograph. Like I/O Automata, Timed I/O Automata provide a rich set of capabilities for system modelling and analysis. Methods used for analyzing TIOAs are essentially the same as those used for ordinary I/O automata: invariant assertions, compositional reasoning, and correspondences between levels of abstraction. However, all of these methods needed to be modified somewhat from their counterparts for I/O Automata, to take into account the timing of events.

Figure 6 illustrates a simple communication channel that can be modelled as a Timed I/O Automaton. Arrows in the figure represent external discrete actions, through which the channel automaton can interact with its environment. The incoming arrow represents the input action, send(m), by means of which the environment can inject a message m into the channel. The outgoing arrow represents the output action receive(m), by means of which the channel can deliver a message m to its environment.
The next section includes related work after which then we discussed operation on timed I/O automata in next section. The leader election algorithm in tempo is discussed in further section. Finally, the paper is concluded.

The most important operation provided for TIOAs is parallel composition, by which individual TIOAs can be combined to produce a model for a larger timed system. The model for the composed system describes interactions among the components, which involve joint participation in discrete transitions. All TIOAs in a composition participate in trajectories concurrently, allowing the same amount of time to pass. Composition requires certain "compatibility" conditions, namely, that no output action is an output of more than one automaton, and that no hidden action of any automaton is shared with any other automaton. On the other hand, an output of one automaton is allowed to be an input of any number of other automata, and an input may be shared by any number of automata. Notice that all communication between TIOAs is by means of discrete actions. TIOA does not provide directly for other forms of communication, such as shared variable communication. If you want to use TIOAs to model a system of processes communicating by means of shared variable, you have two options: (1) we can model the entire system of processes plus variables as a single automaton, for the Fischer mutual exclusion algorithm, (2) we can model the shared variables as automata, with inputs representing invocations of operations and outputs representing responses. Then the entire system of processes and objects can be modelled as a composition of automata.

TIOA also does not support continuous communication, for example, transmission of a continuous signal between two components. The more general Hybrid I/O Automata modelling framework allows both discrete and continuous communication among components. The composition operation respects traces; for example, if A1 implements A2 then the composition of A1 and B implements the composition of A2 and B. Composition also satisfies projection and pasting results, which are fundamental for compositional design and verification of systems: a trace of a composition of TIOAs "projects" to give traces of the individual TIOAs, and traces of components are "pastable" to give traces of the composition.

Finally, the TIOA framework provides a hiding operation for TIOAs, by which some output actions become reclassified as hidden. This implies that, when the new automaton is composed with other automata, these newly-hidden actions are no longer available for communication with the other TIOAs.

Leader Election Algorithm is a simple distributed algorithm in which several processes attempt to coordinate so that one of their number is distinguished as the "leader", at any point in time. We assume that processes may fail and recover. Since we would like the leader to be a non-failed process, the identity of the leader may have to change from time to time, during the operation of the system. The Distributed Algorithms research literature contains many examples of leader election algorithms. The particular algorithm we consider here is based on...
one used in a proposed fault-tolerant extension of the DHCP IP-address-assignment protocol.

In that setting, leader election is used to choose the server that is currently responsible for managing a particular IP address. This application requires that the leader-election algorithm satisfy a kind of "mutual exclusion" property: it should never allow two processes to believe, at the same time, that they are the leader.

The algorithm which is described uses a separate failure-detection service, which provides information to the processes about which processes are currently alive. In practice, the failure-detection service would itself be implemented using another distributed algorithm, using a strategy based on timeouts. However, we present it abstractly here, as a single global service automaton. The processes are assumed to have local clocks, which are reliable and increase at rate 1. Less synchronized clocks could also be used, with slight adjustments in bounds. We assume that, when a process recovers, it resumes its computation with its entire variable restored to their initial values. The only exception to this rule is the local clock, which we assume remains reliable. The processes use time for other purposes besides failure detection; in particular, a process must wait a specified amount of time after recovery before it can decide that it is the leader.

This leader-election algorithm is a typical example of a distributed algorithm in which a collection of processes coordinate by using a shared service. It is atypical in that the processes do not communicate in any other way except through the shared service; usually, the processes would also communicate via point-to-point or broadcast channels. We could also model such channels as TIOAs. Like that, many concurrent algorithms can be implemented in tempo. The sample program of tempo for Alarm is as follow:

```plaintext
let legalTime(hour, minute) : Nat,Nat -> Bool = minute < 60 ∧ hour < 24;

automaton Alarm
signature
  input  showTime(hour, minute: Nat) where legalTime(hour, minute),
  setAlarm(hour, minute: Nat) where legalTime(hour, minute),
  toggleAlarm
output ring
states
  alarmTime: Nat := 0;
  turnedOn: Bool := false;
  ringNow: Bool := false;
transitions
  input setAlarm(hour, minute)
    eff alarmTime := (60*hour) + minute;
  input showTime(hour, minute)
    eff ringNow := turnedOn ∧ alarmTime = (60*hour) + minute;
  input toggleAlarm
    eff turnedOn := ~turnedOn;
```

© 2013, [http://www.journalofcomputerscience.com](http://www.journalofcomputerscience.com) - TIJCSA All Rights Reserved
output ring
pre ringNow;

5. Case Study 3: JMS Publish/Subscribe

The JMS API includes support for the two most common enterprise-messaging models namely, message queuing and publish/subscribe. The message queuing model provides one-to-one delivery semantics. Clients send messages to and receive messages from queues. There may be multiple sender and receivers associated with a given queue but each message sent by a sender is consumed by exactly one receiver. This means that if multiple receivers are associated with a queue, some sort of arbitration is required by the JMS Provider to decide which one will receive a given message. The publish/subscribe model complements the queuing model in that it provides one-to-many delivery semantics. Clients publish messages to and receive messages from topics. Each message may be consumed by zero, one or more subscribers.

Publish-and-subscribe [11] systems handle messages intended for multiple receivers. Publish-and-subscribe systems send messages to a destination called a topic. An example of an application using publish-and-subscribe would be a manufacturer that needs to communicate schedule and quantity changes from its demand forecasting system to suppliers. The manufacturer would use the publish-and-subscribe system to send the messages simultaneously to all vendors supplying components for a particular product.

![Publish-and-Subscribe Messaging](image)

The JMS is an API for accessing enterprise messaging systems. The JMS specification defines an interface but does not itself define an implementation. The specification is vendor neutral; it sets requirements but does not dictate how they are to be implemented. This means that it is up to the vendor to add facilities, services, or enhancements not included or defined explicitly in the specification.

6. Results and Conclusion

The Case studies has been carried out in efficient manner and are implemented in various platforms. The case study of temporal logics in ontologies has been performed in protégé.
The case study of timed I/O automata has been performed in Tempo IDE. The case study of JMS is been performed in Eclipse using Jboss server. The implementation is done in all three case studies. The snapshots of CHRONOS are as follows,

![Figure 8: The Class hierarchy and Individual](image1)

![Figure 9: The Object Property and Graphics Representation](image2)

The Veromodo toolkit Tempo has been used to implement the alarm clock. The snapshots are as follows,

![Figure 10: The Automata of Alarm Clock](image3)
Figure 11: The Final Output as Java Program of Alarm Clock

The JMS Publish/Subscribe is implemented in Eclipse using Jboss server. The snapshots are as follows,

Figure 12: The Java Code of Producer and Consumer
REFERENCES